Matplotlib

Release 3.3.3

John Hunter
Darren Dale
Eric Firing
Michael Droettboom
and the matplotlib development team

December 22, 2020

CONTENTS

Part I

User's Guide

CHAPTER

ONE

INSTALLATION GUIDE

Note: If you wish to contribute to the project, it's recommended you *install the latest development version*.

Contents

1.1 Installing an official release

Matplotlib and its dependencies are available as wheel packages for macOS, Windows and Linux distributions:

```
python -m pip install -U pip
python -m pip install -U matplotlib
```

If this command results in Matplotlib being compiled from source and there's trouble with the compilation, you can add `--prefer-binary` to select the newest version of Matplotlib for which there is a precompiled wheel for your OS and Python.

Note: The following backends work out of the box: Agg, ps, pdf, svg

Python is typically shipped with tk bindings which are used by TkAgg.

For support of other GUI frameworks, LaTeX rendering, saving animations and a larger selection of file formats, you need to install *additional dependencies*.

Although not required, we suggest also installing `IPython` for interactive use. To easily install a complete Scientific Python stack, see *Scientific Python Distributions* below.

1.1.1 Test data

The wheels (`*.whl`) on the PyPI download page do not contain test data or example code.

If you want to try the many demos that come in the Matplotlib source distribution, download the `*.tar.gz` file and look in the `examples` subdirectory.

To run the test suite:

- extract the `lib/matplotlib/tests` or `lib/mpl_toolkits/tests` directories from the source distribution.
- install test dependencies: pytest, MiKTeX, GhostScript, ffmpeg, avconv, ImageMagick, and Inkscape.
- run `python -mpytest`.

1.2 Third-party distributions of Matplotlib

1.2.1 Scientific Python Distributions

Anaconda and ActiveState are excellent choices that "just work" out of the box for Windows, macOS and common Linux platforms. WinPython is an option for Windows users. All of these distributions include Matplotlib and *lots* of other useful (data) science tools.

1.2.2 Linux: using your package manager

If you are on Linux, you might prefer to use your package manager. Matplotlib is packaged for almost every major Linux distribution.

- Debian / Ubuntu: `sudo apt-get install python3-matplotlib`
- Fedora: `sudo dnf install python3-matplotlib`
- Red Hat: `sudo yum install python3-matplotlib`
- Arch: `sudo pacman -S python-matplotlib`

1.3 Installing from source

If you are interested in contributing to Matplotlib development, running the latest source code, or just like to build everything yourself, it is not difficult to build Matplotlib from source. Grab the latest *tar.gz* release file from the PyPI files page, or if you want to develop Matplotlib or just need the latest bugfixed version, grab the latest git version, and see *Install from source*.

Matplotlib can be installed from the source directory with a simple

```
python -m pip install .
```

We provide a setup.cfg file which you can use to customize the build process. For example, which default backend to use, whether some of the optional libraries that Matplotlib ships with are installed, and so on. This file will be particularly useful to those packaging Matplotlib.

1.3.1 Dependencies

Matplotlib requires the following dependencies:

- Python (>= 3.6)
- NumPy (>= 1.15)
- setuptools
- cycler (>= 0.10.0)
- dateutil (>= 2.1)
- kiwisolver (>= 1.0.0)
- Pillow (>= 6.2)
- pyparsing (>=2.0.3)

Optionally, you can also install a number of packages to enable better user interface toolkits. See *What is a backend?* for more details on the optional Matplotlib backends and the capabilities they provide.

- Tk (>= 8.3, != 8.6.0 or 8.6.1): for the Tk-based backends.
- PyQt4 (>= 4.6) or PySide (>= 1.0.3)[1]: for the Qt4-based backends.
- PyQt5 or PySide2: for the Qt5-based backends.
- PyGObject: for the GTK3-based backends[2].
- wxPython (>= 4)[3]: for the wx-based backends.
- pycairo (>= 1.11.0) or cairocffi (>= 0.8): for the GTK3 and/or cairo-based backends.
- Tornado: for the WebAgg backend.

For better support of animation output format and image file formats, LaTeX, etc., you can install the following:

- ffmpeg: for saving movies.
- ImageMagick: for saving animated gifs.
- LaTeX (with cm-super) and GhostScript (>=9.0) : for rendering text with LaTeX.
- fontconfig (>= 2.7): for detection of system fonts on Linux.

1.3.2 FreeType and Qhull

Matplotlib depends on FreeType (>= 2.3), a font rendering library, and on Qhull (>= 2015.2), a library for computing triangulations. By default (except on AIX) Matplotlib downloads and builds its own copy of FreeType (this is necessary to run the test suite, because different versions of FreeType rasterize characters differently), and uses its own copy of Qhull.

To force Matplotlib to use a copy of FreeType or Qhull already installed in your system, create a `setup.cfg` file with the following contents:

```
[libs]
system_freetype = true
system_qhull = true
```

before running `python -m pip install .`.

In this case, you need to install the FreeType and Qhull library and headers. This can be achieved using a package manager, e.g. for FreeType:

[1] PySide cannot be pip-installed on Linux (but can be conda-installed).

[2] If using pip (and not conda), PyGObject must be built from source; see https://pygobject.readthedocs.io/en/latest/devguide/dev_environ.html.

[3] If using pip (and not conda) on Linux, wxPython wheels must be manually downloaded from https://wxpython.org/pages/downloads/.

```
# Pick ONE of the following:
sudo apt install libfreetype6-dev  # Debian/Ubuntu
sudo dnf install freetype-devel  # Fedora
brew install freetype  # macOS with Homebrew
conda install freetype  # conda, any OS
```

(adapt accordingly for Qhull).

On Linux and macOS, it is also recommended to install pkg-config, a helper tool for locating FreeType:

```
# Pick ONE of the following:
sudo apt install pkg-config  # Debian/Ubuntu
sudo dnf install pkgconf  # Fedora
brew install pkg-config  # macOS with Homebrew
conda install pkg-config  # conda
# Or point the PKG_CONFIG environment variable to the path to pkg-config:
export PKG_CONFIG=...
```

If not using pkg-config (in particular on Windows), you may need to set the include path (to the library headers) and link path (to the libraries) explicitly, if they are not in standard locations. This can be done using standard environment variables -- on Linux and OSX:

```
export CFLAGS='-I/directory/containing/ft2build.h'
export LDFLAGS='-L/directory/containing/libfreetype.so'
```

and on Windows:

```
set CL=/IC:\directory\containing\ft2build.h
set LINK=/LIBPATH:C:\directory\containing\freetype.lib
```

Note: Matplotlib always uses its own copies of the following libraries:

- `Agg`: the Anti-Grain Geometry C++ rendering engine;
- `ttconv`: a TrueType font utility.

1.3.3 Building on Windows

Compiling Matplotlib (or any other extension module, for that matter) requires Visual Studio 2015 or later.

If you are building your own Matplotlib wheels (or sdists), note that any DLLs that you copy into the source tree will be packaged too.

1.3.4 Conda packages

The conda packaging scripts for Matplotlib are available at https://github.com/conda-forge/matplotlib-feedstock.

CHAPTER

TWO

TUTORIALS

This page contains more in-depth guides for using Matplotlib. It is broken up into beginner, intermediate, and advanced sections, as well as sections covering specific topics.

For shorter examples, see our examples page. You can also find external resources and a FAQ in our user guide.

2.1 Introductory

These tutorials cover the basics of creating visualizations with Matplotlib, as well as some best-practices in using the package effectively.

2.1.1 Usage Guide

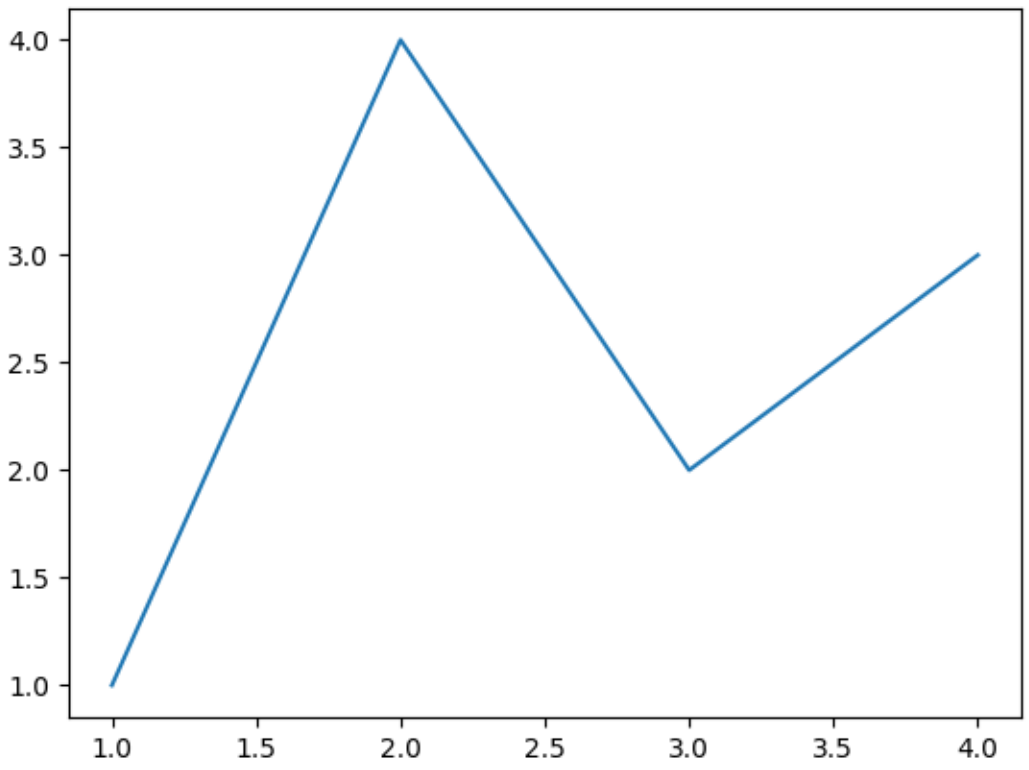
This tutorial covers some basic usage patterns and best-practices to help you get started with Matplotlib.

```
import matplotlib.pyplot as plt
import numpy as np
```

A simple example

Matplotlib graphs your data on *Figure*s (i.e., windows, Jupyter widgets, etc.), each of which can contain one or more *Axes* (i.e., an area where points can be specified in terms of x-y coordinates (or theta-r in a polar plot, or x-y-z in a 3D plot, etc.). The most simple way of creating a figure with an axes is using *pyplot.subplots*. We can then use *Axes.plot* to draw some data on the axes:

```
fig, ax = plt.subplots()  # Create a figure containing a single axes.
ax.plot([1, 2, 3, 4], [1, 4, 2, 3])  # Plot some data on the axes.
```



Out:

```
[<matplotlib.lines.Line2D object at 0x7f53fb3ae670>]
```

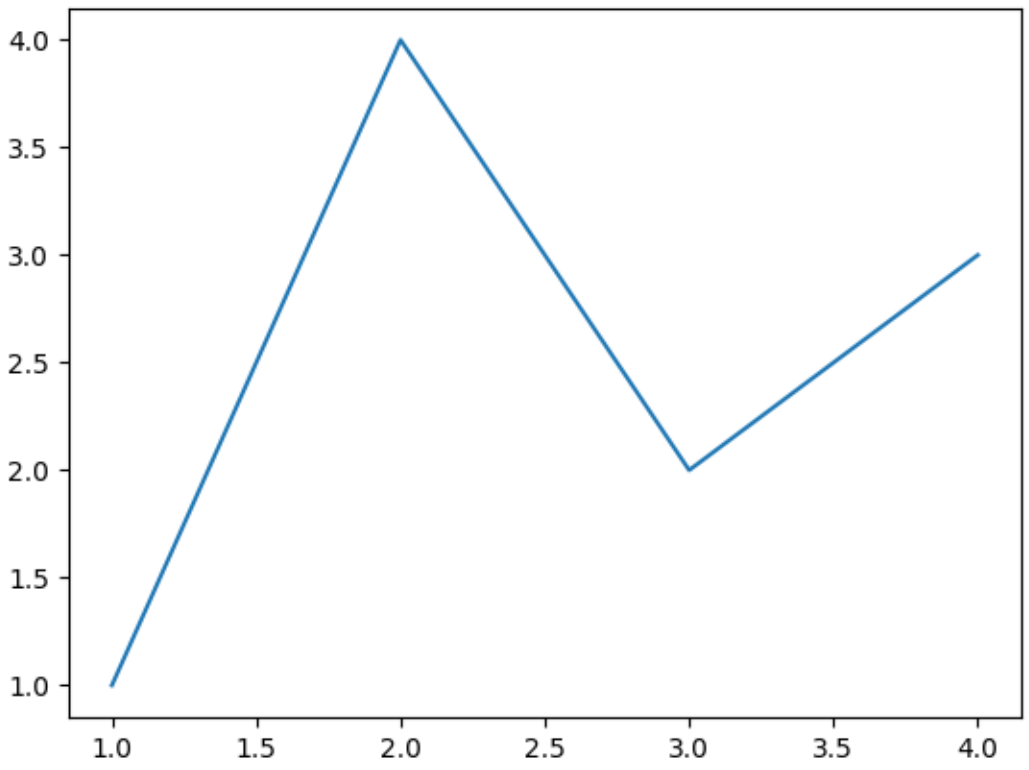
Many other plotting libraries or languages do not require you to explicitly create an axes. For example, in MATLAB, one can just do

```
plot([1, 2, 3, 4], [1, 4, 2, 3])  % MATLAB plot.
```

and get the desired graph.

In fact, you can do the same in Matplotlib: for each *Axes* graphing method, there is a corresponding function in the *matplotlib.pyplot* module that performs that plot on the "current" axes, creating that axes (and its parent figure) if they don't exist yet. So the previous example can be written more shortly as

```
plt.plot([1, 2, 3, 4], [1, 4, 2, 3])  # Matplotlib plot.
```

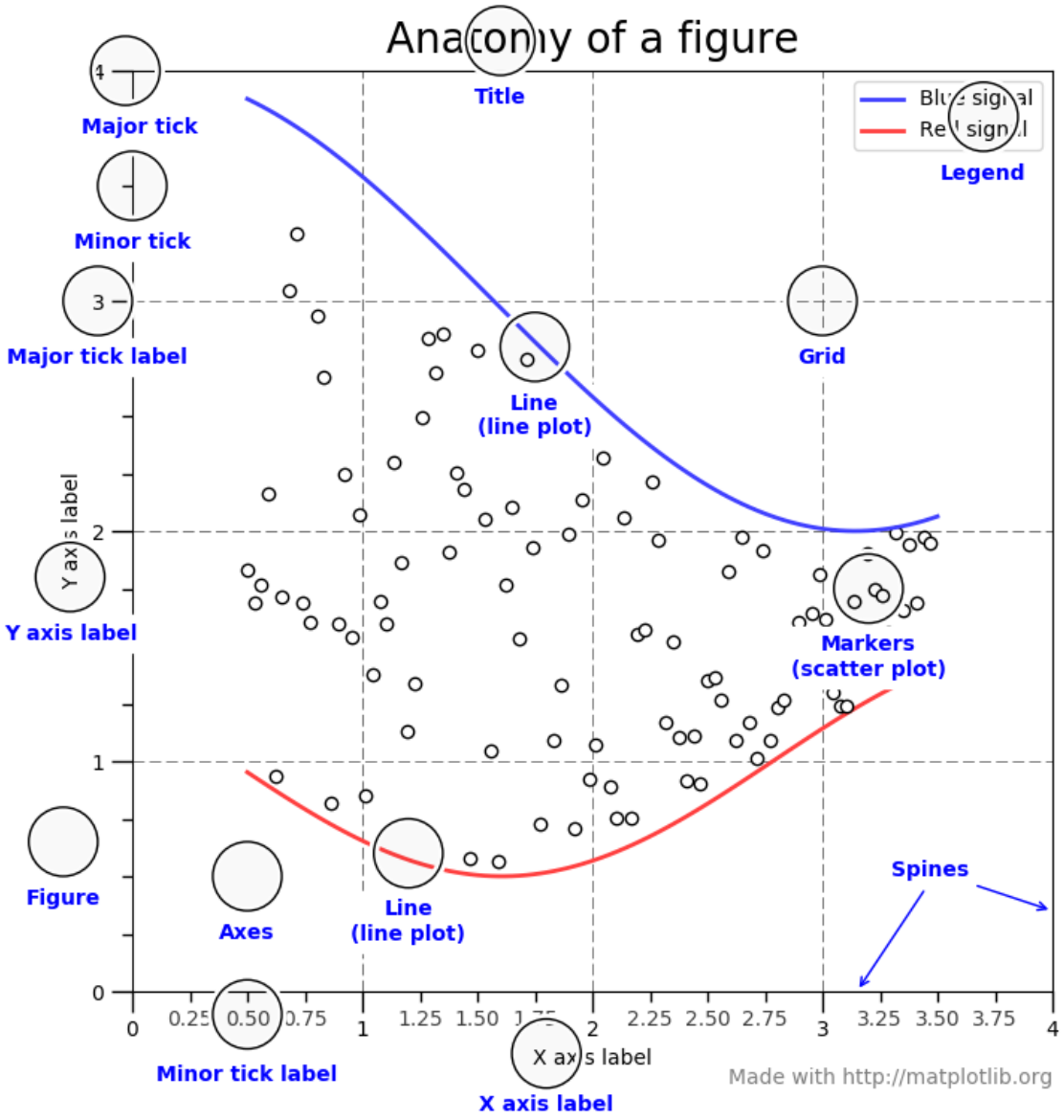


Out:

```
[<matplotlib.lines.Line2D object at 0x7f53ff82dc70>]
```

Parts of a Figure

Now, let's have a deeper look at the components of a Matplotlib figure.



`Figure`

The **whole** figure. The figure keeps track of all the child *`Axes`*, a smattering of 'special' artists (titles, figure legends, etc), and the **canvas**. (Don't worry too much about the canvas, it is crucial as it is the object that actually does the drawing to get you your plot, but as the user it is more-or-less invisible to you). A figure can contain any number of *`Axes`*, but will typically have at least one.

The easiest way to create a new figure is with pyplot:

```
fig = plt.figure()  # an empty figure with no Axes
fig, ax = plt.subplots()  # a figure with a single Axes
fig, axs = plt.subplots(2, 2)  # a figure with a 2x2 grid of Axes
```

It's convenient to create the axes together with the figure, but you can also add axes later on, allowing for more complex axes layouts.

Axes

This is what you think of as 'a plot', it is the region of the image with the data space. A given figure can contain many Axes, but a given *Axes* object can only be in one *Figure*. The Axes contains two (or three in the case of 3D) *Axis* objects (be aware of the difference between **Axes** and **Axis**) which take care of the data limits (the data limits can also be controlled via the *axes.Axes.set_xlim()* and *axes.Axes.set_ylim()* methods). Each *Axes* has a title (set via *set_title()*), an x-label (set via *set_xlabel()*), and a y-label set via *set_ylabel()*).

The *Axes* class and its member functions are the primary entry point to working with the OO interface.

Axis

These are the number-line-like objects. They take care of setting the graph limits and generating the ticks (the marks on the axis) and ticklabels (strings labeling the ticks). The location of the ticks is determined by a *Locator* object and the ticklabel strings are formatted by a *Formatter*. The combination of the correct *Locator* and *Formatter* gives very fine control over the tick locations and labels.

Artist

Basically everything you can see on the figure is an artist (even the *Figure*, *Axes*, and *Axis* objects). This includes *Text* objects, *Line2D* objects, *collections* objects, *Patch* objects ... (you get the idea). When the figure is rendered, all of the artists are drawn to the **canvas**. Most Artists are tied to an Axes; such an Artist cannot be shared by multiple Axes, or moved from one to another.

Types of inputs to plotting functions

All of plotting functions expect `numpy.array` or `numpy.ma.masked_array` as input. Classes that are 'array-like' such as `pandas` data objects and `numpy.matrix` may or may not work as intended. It is best to convert these to `numpy.array` objects prior to plotting.

For example, to convert a `pandas.DataFrame`

```
a = pandas.DataFrame(np.random.rand(4, 5), columns = list('abcde'))
a_asarray = a.values
```

and to convert a `numpy.matrix`

```
b = np.matrix([[1, 2], [3, 4]])
b_asarray = np.asarray(b)
```

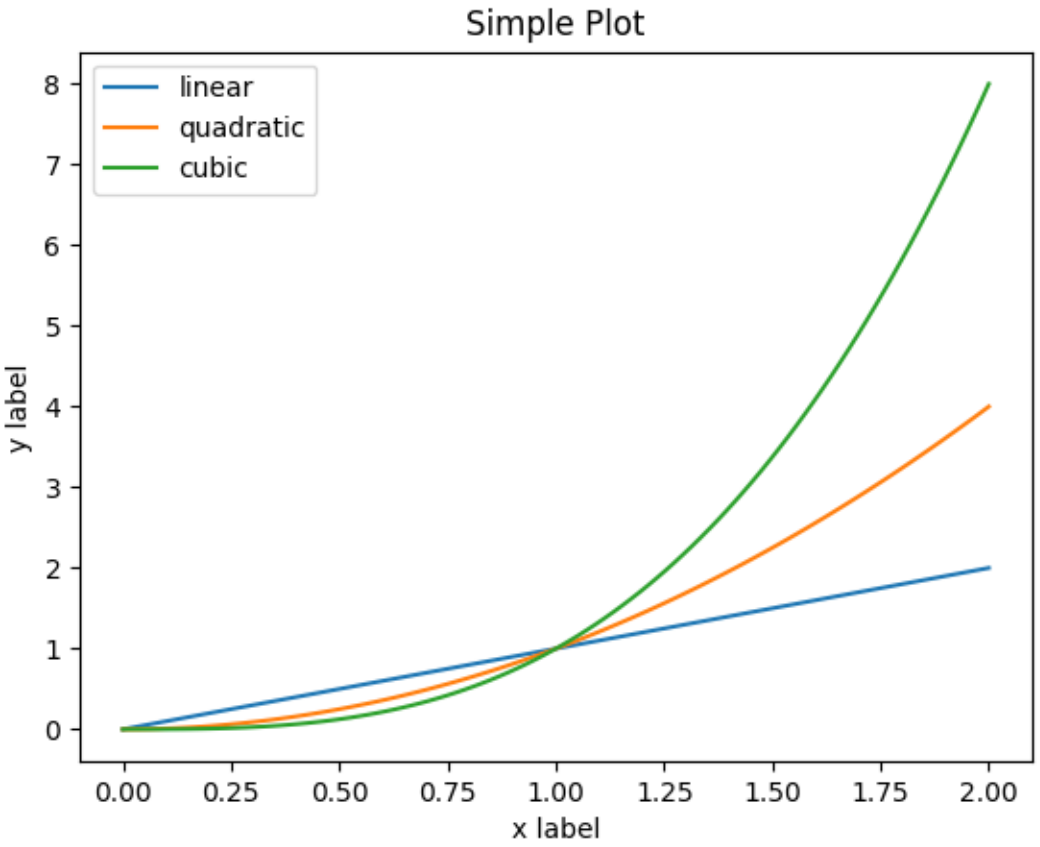
The object-oriented interface and the pyplot interface

As noted above, there are essentially two ways to use Matplotlib:

- Explicitly create figures and axes, and call methods on them (the "object-oriented (OO) style").
- Rely on pyplot to automatically create and manage the figures and axes, and use pyplot functions for plotting.

So one can do (OO-style)

```
x = np.linspace(0, 2, 100)

# Note that even in the OO-style, we use `.pyplot.figure` to create the figure.
fig, ax = plt.subplots()  # Create a figure and an axes.
ax.plot(x, x, label='linear')  # Plot some data on the axes.
ax.plot(x, x**2, label='quadratic')  # Plot more data on the axes...
ax.plot(x, x**3, label='cubic')  # ... and some more.
ax.set_xlabel('x label')  # Add an x-label to the axes.
ax.set_ylabel('y label')  # Add a y-label to the axes.
ax.set_title("Simple Plot")  # Add a title to the axes.
ax.legend()  # Add a legend.
```

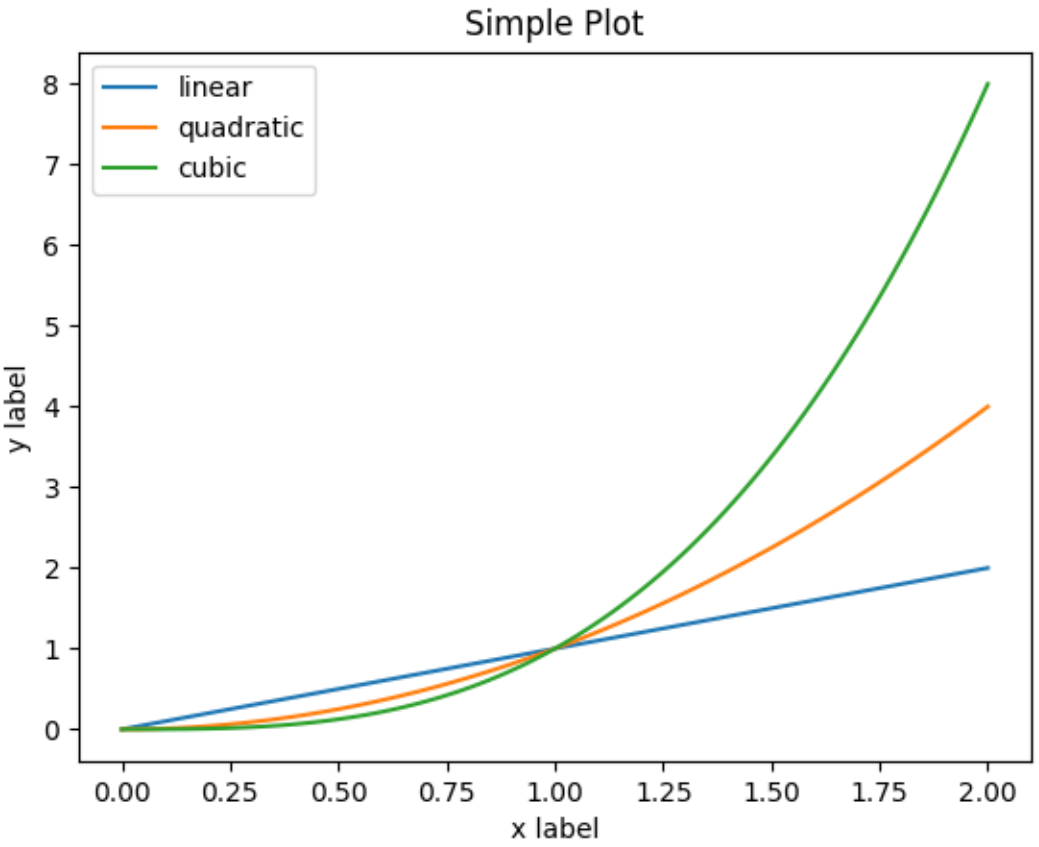


Out:

```
<matplotlib.legend.Legend object at 0x7f53fec96bb0>
```

or (pyplot-style)

```
x = np.linspace(0, 2, 100)

plt.plot(x, x, label='linear')  # Plot some data on the (implicit) axes.
plt.plot(x, x**2, label='quadratic')  # etc.
plt.plot(x, x**3, label='cubic')
plt.xlabel('x label')
plt.ylabel('y label')
plt.title("Simple Plot")
plt.legend()
```



Out:

```
<matplotlib.legend.Legend object at 0x7f53fedee070>
```

Actually there is a third approach, for the case where you are embedding Matplotlib in a GUI application, which completely drops pyplot, even for figure creation. We won't discuss it here; see the corresponding section in the gallery for more info (user_interfaces).

Matplotlib's documentation and examples use both the OO and the pyplot approaches (which are equally powerful), and you should feel free to use either (however, it is preferable pick one of them and stick to it, instead of mixing them). In general, we suggest to restrict pyplot to interactive plotting (e.g., in a Jupyter notebook), and to prefer the OO-style for non-interactive plotting (in functions and scripts that are intended to be reused as part of a larger project).

Note: In older examples, you may find examples that instead used the so-called `pylab` interface, via `from pylab import *`. This star-import imports everything both from pyplot and from `numpy`, so that one could do

```
x = linspace(0, 2, 100)
```

(continues on next page)

(continued from previous page)

```
plot(x, x, label='linear')
...
```

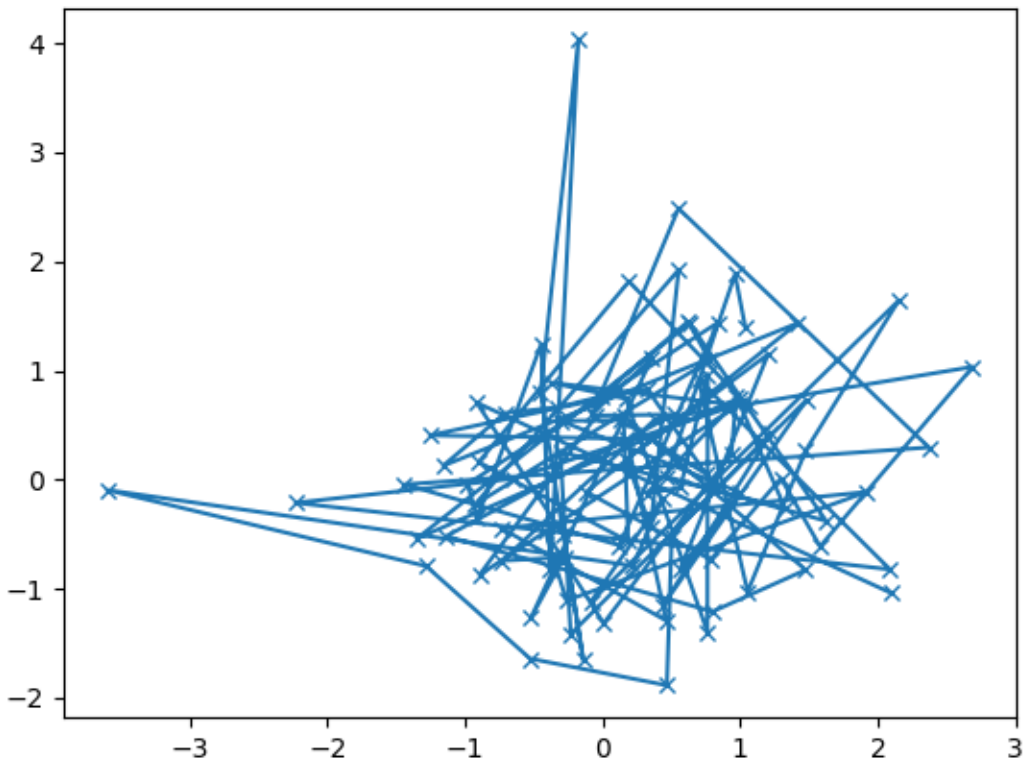
for an even more MATLAB-like style. This approach is strongly discouraged nowadays and deprecated; it is only mentioned here because you may still encounter it in the wild.

Typically one finds oneself making the same plots over and over again, but with different data sets, which leads to needing to write specialized functions to do the plotting. The recommended function signature is something like:

```
def my_plotter(ax, data1, data2, param_dict):
    """
    A helper function to make a graph

    Parameters
    ----------
    ax : Axes
        The axes to draw to

    data1 : array
       The x data

    data2 : array
       The y data

    param_dict : dict
       Dictionary of kwargs to pass to ax.plot

    Returns
    -------
    out : list
        list of artists added
    """
    out = ax.plot(data1, data2, **param_dict)
    return out
```

which you would then use as:

```
data1, data2, data3, data4 = np.random.randn(4, 100)
fig, ax = plt.subplots(1, 1)
my_plotter(ax, data1, data2, {'marker': 'x'})
```

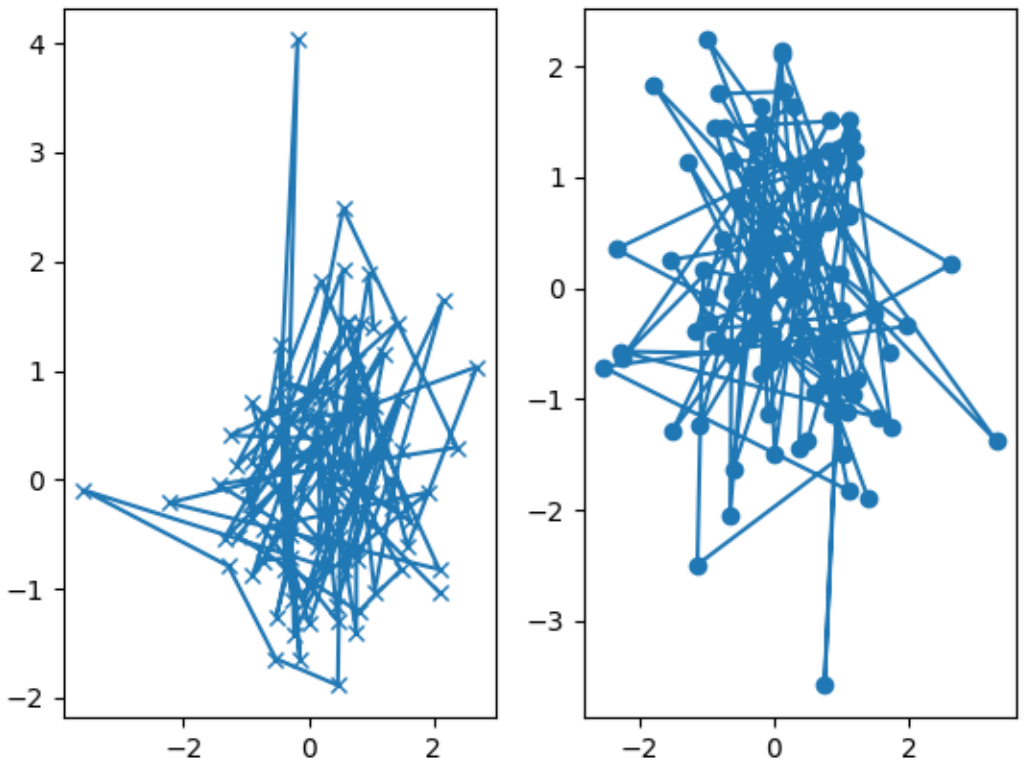


Out:

```
[<matplotlib.lines.Line2D object at 0x7f53fac8e730>]
```

or if you wanted to have 2 sub-plots:

```
fig, (ax1, ax2) = plt.subplots(1, 2)
my_plotter(ax1, data1, data2, {'marker': 'x'})
my_plotter(ax2, data3, data4, {'marker': 'o'})
```



Out:

```
[<matplotlib.lines.Line2D object at 0x7f540066d6d0>]
```

For these simple examples this style seems like overkill, however once the graphs get slightly more complex it pays off.

Backends

What is a backend?

A lot of documentation on the website and in the mailing lists refers to the "backend" and many new users are confused by this term. matplotlib targets many different use cases and output formats. Some people use matplotlib interactively from the python shell and have plotting windows pop up when they type commands. Some people run Jupyter notebooks and draw inline plots for quick data analysis. Others embed matplotlib into graphical user interfaces like wxpython or pygtk to build rich applications. Some people use matplotlib in batch scripts to generate postscript images from numerical simulations, and still others run web application servers to dynamically serve up graphs.

To support all of these use cases, matplotlib can target different outputs, and each of these capabilities is called a backend; the "frontend" is the user facing code, i.e., the plotting code, whereas the "backend" does all the hard work behind-the-scenes to make the figure. There are two types of backends: user interface backends (for use in pygtk, wxpython, tkinter, qt4, or macosx; also referred to as "interactive backends") and hardcopy backends to make image files (PNG, SVG, PDF, PS; also referred to as "non-interactive backends").

Selecting a backend

There are three ways to configure your backend:

1. The `rcParams["backend"]` (default: `'agg'`) parameter in your `matplotlibrc` file
2. The *MPLBACKEND* environment variable
3. The function *matplotlib.use()*

A more detailed description is given below.

If multiple of these are configurations are present, the last one from the list takes precedence; e.g. calling *matplotlib.use()* will override the setting in your `matplotlibrc`.

If no backend is explicitly set, Matplotlib automatically detects a usable backend based on what is available on your system and on whether a GUI event loop is already running. On Linux, if the environment variable *DISPLAY* is unset, the "event loop" is identified as "headless", which causes a fallback to a noninteractive backend (agg).

Here is a detailed description of the configuration methods:

1. Setting `rcParams["backend"]` (default: `'agg'`) in your `matplotlibrc` file:

   ```
   backend : qt5agg   # use pyqt5 with antigrain (agg) rendering
   ```

   See also *Customizing Matplotlib with style sheets and rcParams*.

2. Setting the *MPLBACKEND* environment variable:

   You can set the environment variable either for your current shell or for a single script.

   On Unix:

   ```
   > export MPLBACKEND=qt5agg
   > python simple_plot.py

   > MPLBACKEND=qt5agg python simple_plot.py
   ```

   On Windows, only the former is possible:

```
> set MPLBACKEND=qt5agg
> python simple_plot.py
```

Setting this environment variable will override the `backend` parameter in *any* `matplotlibrc`, even if there is a `matplotlibrc` in your current working directory. Therefore, setting *MPLBACKEND* globally, e.g. in your `.bashrc` or `.profile`, is discouraged as it might lead to counter-intuitive behavior.

3. If your script depends on a specific backend you can use the function *matplotlib.use()*:

```
import matplotlib
matplotlib.use('qt5agg')
```

This should be done before any figure is created; otherwise Matplotlib may fail to switch the backend and raise an ImportError.

Using *use* will require changes in your code if users want to use a different backend. Therefore, you should avoid explicitly calling *use* unless absolutely necessary.

The builtin backends

By default, Matplotlib should automatically select a default backend which allows both interactive work and plotting from scripts, with output to the screen and/or to a file, so at least initially you will not need to worry about the backend. The most common exception is if your Python distribution comes without `tkinter` and you have no other GUI toolkit installed; this happens on certain Linux distributions, where you need to install a Linux package named `python-tk` (or similar).

If, however, you want to write graphical user interfaces, or a web application server (*How to use Matplotlib in a web application server*), or need a better understanding of what is going on, read on. To make things a little more customizable for graphical user interfaces, matplotlib separates the concept of the renderer (the thing that actually does the drawing) from the canvas (the place where the drawing goes). The canonical renderer for user interfaces is `Agg` which uses the Anti-Grain Geometry C++ library to make a raster (pixel) image of the figure; it is used by the `Qt5Agg`, `Qt4Agg`, `GTK3Agg`, `wxAgg`, `TkAgg`, and `macosx` backends. An alternative renderer is based on the Cairo library, used by `Qt5Cairo`, `Qt4Cairo`, etc.

For the rendering engines, one can also distinguish between vector or raster renderers. Vector graphics languages issue drawing commands like "draw a line from this point to this point" and hence are scale free, and raster backends generate a pixel representation of the line whose accuracy depends on a DPI setting.

Here is a summary of the matplotlib renderers (there is an eponymous backend for each; these are *non-interactive backends*, capable of writing to a file):

Renderer	Filetypes	Description
AGG	png	raster graphics -- high quality images using the Anti-Grain Geometry engine
PDF	pdf	vector graphics -- Portable Document Format
PS	ps, eps	vector graphics -- Postscript output
SVG	svg	vector graphics -- Scalable Vector Graphics
PGF	pgf, pdf	vector graphics -- using the pgf package
Cairo	png, ps, pdf, svg	raster or vector graphics -- using the Cairo library

To save plots using the non-interactive backends, use the `matplotlib.pyplot.savefig('filename')` method.

And here are the user interfaces and renderer combinations supported; these are *interactive backends*, capable of displaying to the screen and of using appropriate renderers from the table above to write to a file:

Backend	Description
Qt5Agg	Agg rendering in a *Qt5* canvas (requires PyQt5). This backend can be activated in IPython with `%matplotlib qt5`.
ipympl	Agg rendering embedded in a Jupyter widget. (requires ipympl). This backend can be enabled in a Jupyter notebook with `%matplotlib ipympl`.
GTK3Agg	Agg rendering to a *GTK* 3.x canvas (requires PyGObject, and pycairo or cairocffi). This backend can be activated in IPython with `%matplotlib gtk3`.
macosx	Agg rendering into a Cocoa canvas in OSX. This backend can be activated in IPython with `%matplotlib osx`.
TkAgg	Agg rendering to a *Tk* canvas (requires TkInter). This backend can be activated in IPython with `%matplotlib tk`.
nbAgg	Embed an interactive figure in a Jupyter classic notebook. This backend can be enabled in Jupyter notebooks via `%matplotlib notebook`.
WebAgg	On `show()` will start a tornado server with an interactive figure.
GTK3Cairo	Cairo rendering to a *GTK* 3.x canvas (requires PyGObject, and pycairo or cairocffi).
Qt4Agg	Agg rendering to a *Qt4* canvas (requires PyQt4 or `pyside`). This backend can be activated in IPython with `%matplotlib qt4`.
wxAgg	Agg rendering to a *wxWidgets* canvas (requires wxPython 4). This backend can be activated in IPython with `%matplotlib wx`.

Note: The names of builtin backends case-insensitive; e.g., 'Qt5Agg' and 'qt5agg' are equivalent.

ipympl

The Jupyter widget ecosystem is moving too fast to support directly in Matplotlib. To install ipympl

```
pip install ipympl
jupyter nbextension enable --py --sys-prefix ipympl
```

or

```
conda install ipympl -c conda-forge
```

See jupyter-matplotlib for more details.

How do I select PyQt4 or PySide?

The *QT_API* environment variable can be set to either `pyqt` or `pyside` to use `PyQt4` or `PySide`, respectively.

Since the default value for the bindings to be used is `PyQt4`, Matplotlib first tries to import it, if the import fails, it tries to import `PySide`.

Using non-builtin backends

More generally, any importable backend can be selected by using any of the methods above. If `name.of.the.backend` is the module containing the backend, use `module://name.of.the.backend` as the backend name, e.g. `matplotlib.use('module://name.of.the.backend')`.

What is interactive mode?

Use of an interactive backend (see *What is a backend?*) permits--but does not by itself require or ensure--plotting to the screen. Whether and when plotting to the screen occurs, and whether a script or shell session continues after a plot is drawn on the screen, depends on the functions and methods that are called, and on a state variable that determines whether matplotlib is in "interactive mode". The default Boolean value is set by the `matplotlibrc` file, and may be customized like any other configuration parameter (see *Customizing Matplotlib with style sheets and rcParams*). It may also be set via *matplotlib.interactive()*, and its value may be queried via *matplotlib.is_interactive()*. Turning interactive mode on and off in the middle of a stream of plotting commands, whether in a script or in a shell, is rarely needed and potentially confusing, so in the following we will assume all plotting is done with interactive mode either on or off.

Note: Major changes related to interactivity, and in particular the role and behavior of *show()*, were made in the transition to matplotlib version 1.0, and bugs were fixed in 1.0.1. Here we describe the version 1.0.1 behavior for the primary interactive backends, with the partial exception of *macosx*.

Interactive mode may also be turned on via *matplotlib.pyplot.ion()*, and turned off via *matplotlib.pyplot.ioff()*.

Note: Interactive mode works with suitable backends in ipython and in the ordinary python shell, but it does *not* work in the IDLE IDE. If the default backend does not support interactivity, an interactive backend can be explicitly activated using any of the methods discussed in *What is a backend?*.

Interactive example

From an ordinary python prompt, or after invoking ipython with no options, try this:

```
import matplotlib.pyplot as plt
plt.ion()
plt.plot([1.6, 2.7])
```

This will pop up a plot window. Your terminal prompt will remain active, so that you can type additional commands such as:

```
plt.title("interactive test")
plt.xlabel("index")
```

On most interactive backends, the figure window will also be updated if you change it via the object-oriented interface. E.g. get a reference to the *Axes* instance, and call a method of that instance:

```
ax = plt.gca()
ax.plot([3.1, 2.2])
```

If you are using certain backends (like `macosx`), or an older version of matplotlib, you may not see the new line added to the plot immediately. In this case, you need to explicitly call *draw()* in order to update the plot:

```
plt.draw()
```

Non-interactive example

Start a fresh session as in the previous example, but now turn interactive mode off:

```
import matplotlib.pyplot as plt
plt.ioff()
plt.plot([1.6, 2.7])
```

Nothing happened--or at least nothing has shown up on the screen (unless you are using *macosx* backend, which is anomalous). To make the plot appear, you need to do this:

```
plt.show()
```

Now you see the plot, but your terminal command line is unresponsive; *pyplot.show() blocks* the input of additional commands until you manually kill the plot window.

What good is this--being forced to use a blocking function? Suppose you need a script that plots the contents of a file to the screen. You want to look at that plot, and then end the script. Without some blocking command such as `show()`, the script would flash up the plot and then end immediately, leaving nothing on the screen.

In addition, non-interactive mode delays all drawing until `show()` is called; this is more efficient than redrawing the plot each time a line in the script adds a new feature.

Prior to version 1.0, `show()` generally could not be called more than once in a single script (although sometimes one could get away with it); for version 1.0.1 and above, this restriction is lifted, so one can write a script like this:

```
import numpy as np
import matplotlib.pyplot as plt

plt.ioff()
for i in range(3):
    plt.plot(np.random.rand(10))
    plt.show()
```

which makes three plots, one at a time. I.e. the second plot will show up, once the first plot is closed.

Summary

In interactive mode, pyplot functions automatically draw to the screen.

When plotting interactively, if using object method calls in addition to pyplot functions, then call *draw()* whenever you want to refresh the plot.

Use non-interactive mode in scripts in which you want to generate one or more figures and display them before ending or generating a new set of figures. In that case, use *show()* to display the figure(s) and to block execution until you have manually destroyed them.

Performance

Whether exploring data in interactive mode or programmatically saving lots of plots, rendering performance can be a painful bottleneck in your pipeline. Matplotlib provides a couple ways to greatly reduce rendering time at the cost of a slight change (to a settable tolerance) in your plot's appearance. The methods available to reduce rendering time depend on the type of plot that is being created.

Line segment simplification

For plots that have line segments (e.g. typical line plots, outlines of polygons, etc.), rendering performance can be controlled by `rcParams["path.simplify"]` (default: `True`) and `rcParams["path.simplify_threshold"]` (default: `0.111111111111`), which can be defined e.g. in the `matplotlibrc` file (see *Customizing Matplotlib with style sheets and rcParams* for more information about the `matplotlibrc` file). `rcParams["path.simplify"]` (default: `True`) is a boolean indicating whether or not line segments are simplified at all. `rcParams["path.simplify_threshold"]` (default: `0.111111111111`) controls how much line segments are simplified; higher thresholds result in quicker rendering.

The following script will first display the data without any simplification, and then display the same data with simplification. Try interacting with both of them:

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl

# Setup, and create the data to plot
y = np.random.rand(100000)
y[50000:] *= 2
y[np.geomspace(10, 50000, 400).astype(int)] = -1
mpl.rcParams['path.simplify'] = True

mpl.rcParams['path.simplify_threshold'] = 0.0
plt.plot(y)
plt.show()

mpl.rcParams['path.simplify_threshold'] = 1.0
plt.plot(y)
plt.show()
```

Matplotlib currently defaults to a conservative simplification threshold of `1/9`. If you want to change your default settings to use a different value, you can change your `matplotlibrc` file. Alternatively, you could create a new style for interactive plotting (with maximal simplification) and another style for publication quality plotting (with minimal simplification) and activate them as necessary. See *Customizing Matplotlib with style sheets and rcParams* for instructions on how to perform these actions.

The simplification works by iteratively merging line segments into a single vector until the next line segment's perpendicular distance to the vector (measured in display-coordinate space) is greater than the `path.simplify_threshold` parameter.

Note: Changes related to how line segments are simplified were made in version 2.1. Rendering time will still be improved by these parameters prior to 2.1, but rendering time for some kinds of data will be vastly improved in versions 2.1 and greater.

Marker simplification

Markers can also be simplified, albeit less robustly than line segments. Marker simplification is only available to *Line2D* objects (through the `markevery` property). Wherever *Line2D* construction parameters are passed through, such as *matplotlib.pyplot.plot()* and *matplotlib.axes.Axes.plot()*, the `markevery` parameter can be used:

```
plt.plot(x, y, markevery=10)
```

The markevery argument allows for naive subsampling, or an attempt at evenly spaced (along the *x* axis) sampling. See the /gallery/lines_bars_and_markers/markevery_demo for more information.

Splitting lines into smaller chunks

If you are using the Agg backend (see *What is a backend?*), then you can make use of `rcParams["agg.path.chunksize"]` (default: `0`) This allows you to specify a chunk size, and any lines with greater than that many vertices will be split into multiple lines, each of which has no more than `agg.path.chunksize` many vertices. (Unless `agg.path.chunksize` is zero, in which case there is no chunking.) For some kind of data, chunking the line up into reasonable sizes can greatly decrease rendering time.

The following script will first display the data without any chunk size restriction, and then display the same data with a chunk size of 10,000. The difference can best be seen when the figures are large, try maximizing the GUI and then interacting with them:

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
mpl.rcParams['path.simplify_threshold'] = 1.0

# Setup, and create the data to plot
y = np.random.rand(100000)
y[50000:] *= 2
y[np.geomspace(10, 50000, 400).astype(int)] = -1
mpl.rcParams['path.simplify'] = True

mpl.rcParams['agg.path.chunksize'] = 0
plt.plot(y)
plt.show()
```

(continues on next page)

(continued from previous page)

```
mpl.rcParams['agg.path.chunksize'] = 10000
plt.plot(y)
plt.show()
```

Legends

The default legend behavior for axes attempts to find the location that covers the fewest data points (`loc='best'`). This can be a very expensive computation if there are lots of data points. In this case, you may want to provide a specific location.

Using the *fast* style

The *fast* style can be used to automatically set simplification and chunking parameters to reasonable settings to speed up plotting large amounts of data. It can be used simply by running:

```
import matplotlib.style as mplstyle
mplstyle.use('fast')
```

It is very light weight, so it plays nicely with other styles, just make sure the fast style is applied last so that other styles do not overwrite the settings:

```
mplstyle.use(['dark_background', 'ggplot', 'fast'])
```

Total running time of the script: (0 minutes 2.252 seconds)

2.1.2 Pyplot tutorial

An introduction to the pyplot interface.

Intro to pyplot

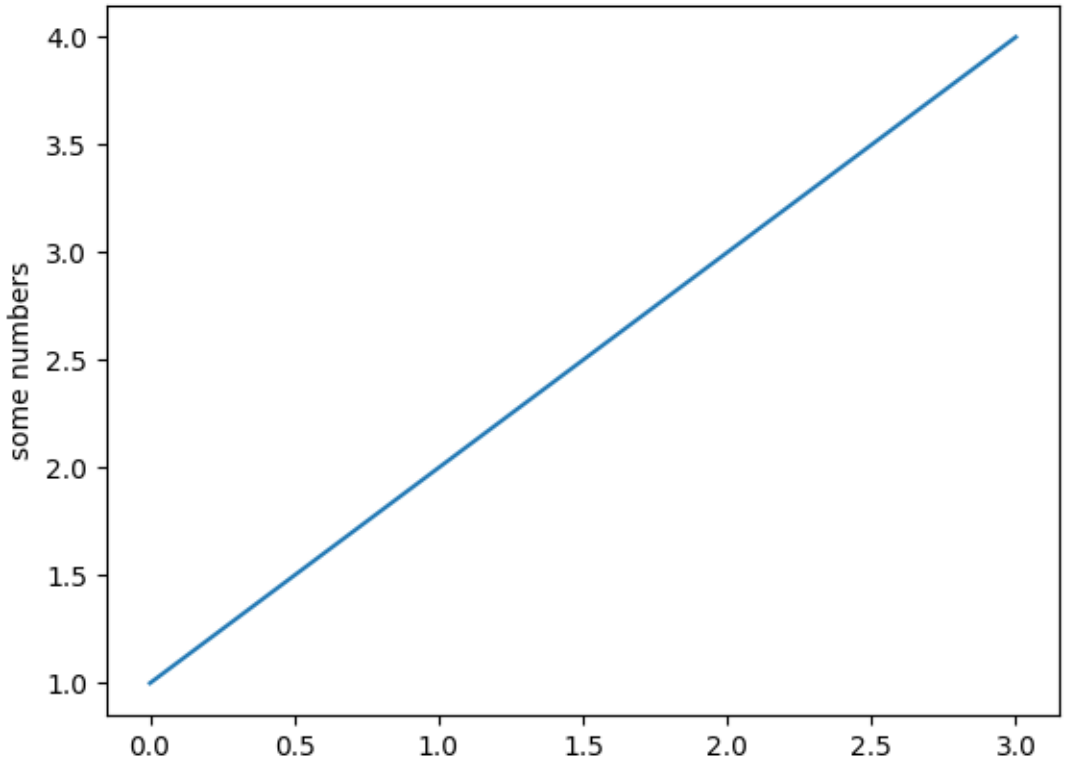
matplotlib.pyplot is a collection of functions that make matplotlib work like MATLAB. Each `pyplot` function makes some change to a figure: e.g., creates a figure, creates a plotting area in a figure, plots some lines in a plotting area, decorates the plot with labels, etc.

In *matplotlib.pyplot* various states are preserved across function calls, so that it keeps track of things like the current figure and plotting area, and the plotting functions are directed to the current axes (please note that "axes" here and in most places in the documentation refers to the *axes part of a figure* and not the strict mathematical term for more than one axis).

Note: the pyplot API is generally less-flexible than the object-oriented API. Most of the function calls you see here can also be called as methods from an `Axes` object. We recommend browsing the tutorials and examples to see how this works.

Generating visualizations with pyplot is very quick:

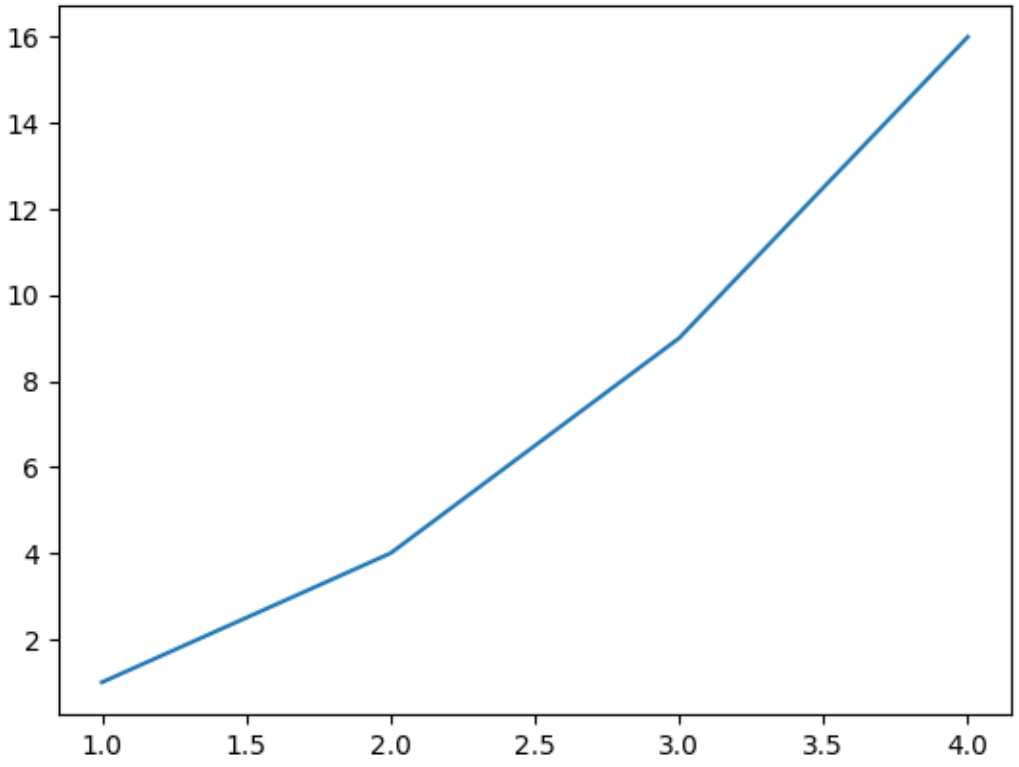
```
import matplotlib.pyplot as plt
plt.plot([1, 2, 3, 4])
plt.ylabel('some numbers')
plt.show()
```



You may be wondering why the x-axis ranges from 0-3 and the y-axis from 1-4. If you provide a single list or array to *plot*, matplotlib assumes it is a sequence of y values, and automatically generates the x values for you. Since python ranges start with 0, the default x vector has the same length as y but starts with 0. Hence the x data are `[0, 1, 2, 3]`.

plot is a versatile function, and will take an arbitrary number of arguments. For example, to plot x versus y, you can write:

```
plt.plot([1, 2, 3, 4], [1, 4, 9, 16])
```



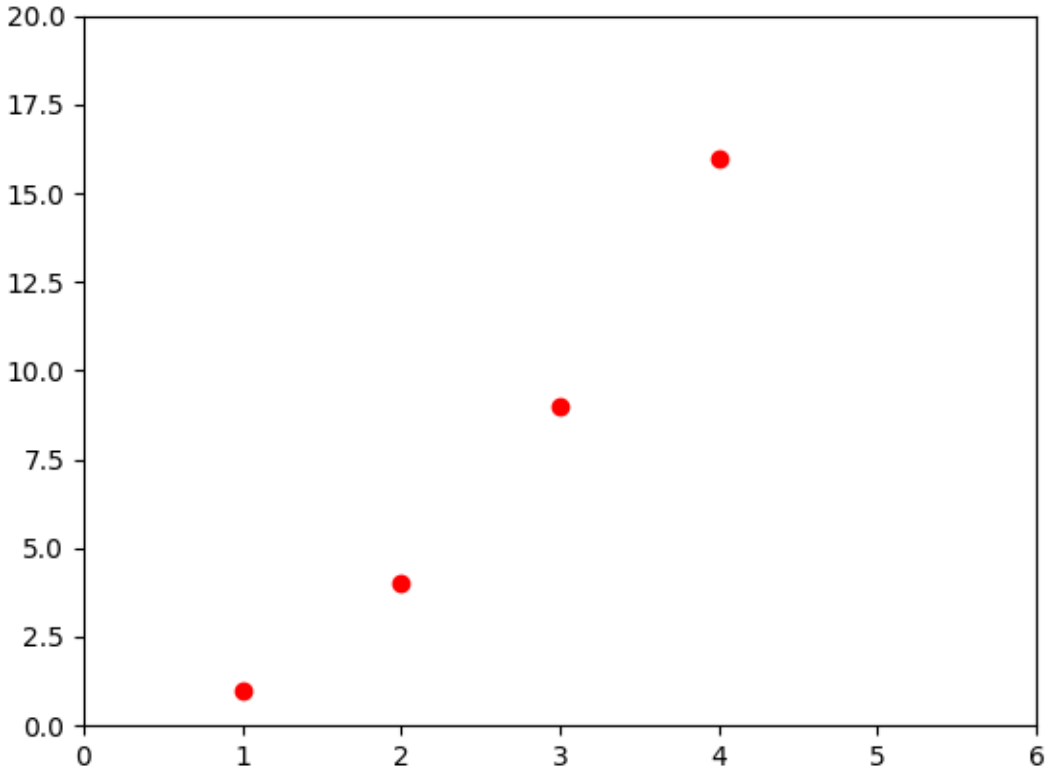
Out:

```
[<matplotlib.lines.Line2D object at 0x7f53fecaa520>]
```

Formatting the style of your plot

For every x, y pair of arguments, there is an optional third argument which is the format string that indicates the color and line type of the plot. The letters and symbols of the format string are from MATLAB, and you concatenate a color string with a line style string. The default format string is 'b-', which is a solid blue line. For example, to plot the above with red circles, you would issue

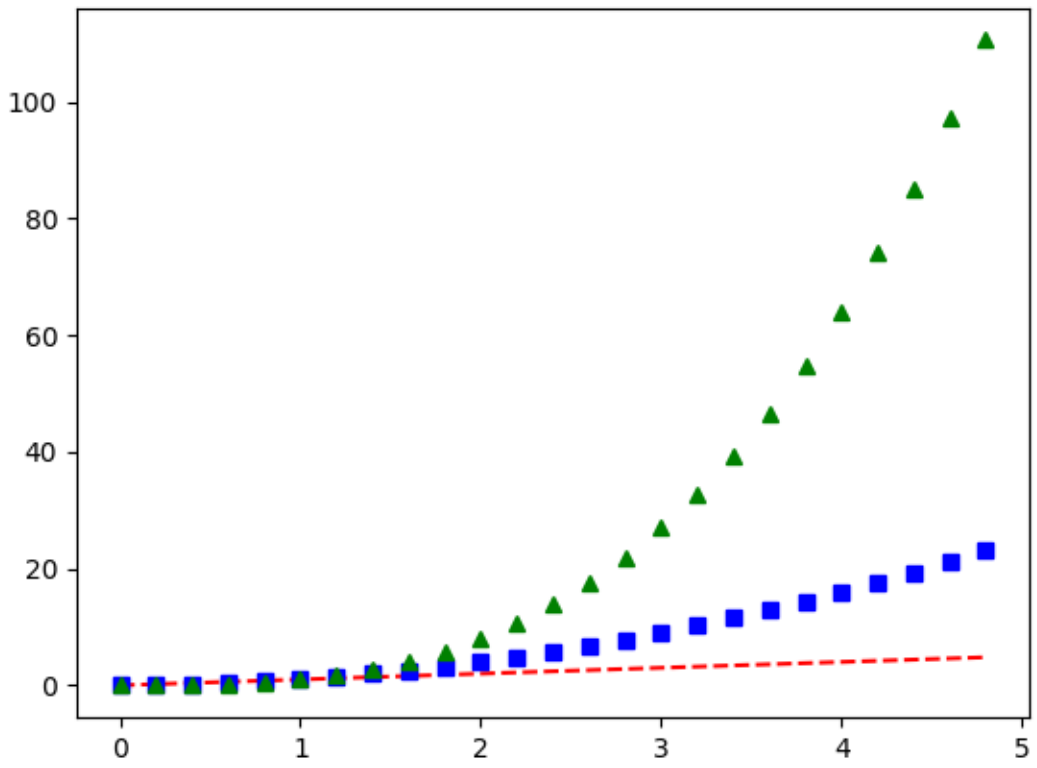
```
plt.plot([1, 2, 3, 4], [1, 4, 9, 16], 'ro')
plt.axis([0, 6, 0, 20])
plt.show()
```



See the *plot* documentation for a complete list of line styles and format strings. The *axis* function in the example above takes a list of `[xmin, xmax, ymin, ymax]` and specifies the viewport of the axes.

If matplotlib were limited to working with lists, it would be fairly useless for numeric processing. Generally, you will use numpy arrays. In fact, all sequences are converted to numpy arrays internally. The example below illustrates plotting several lines with different format styles in one function call using arrays.

```
import numpy as np

# evenly sampled time at 200ms intervals
t = np.arange(0., 5., 0.2)

# red dashes, blue squares and green triangles
plt.plot(t, t, 'r--', t, t**2, 'bs', t, t**3, 'g^')
plt.show()
```

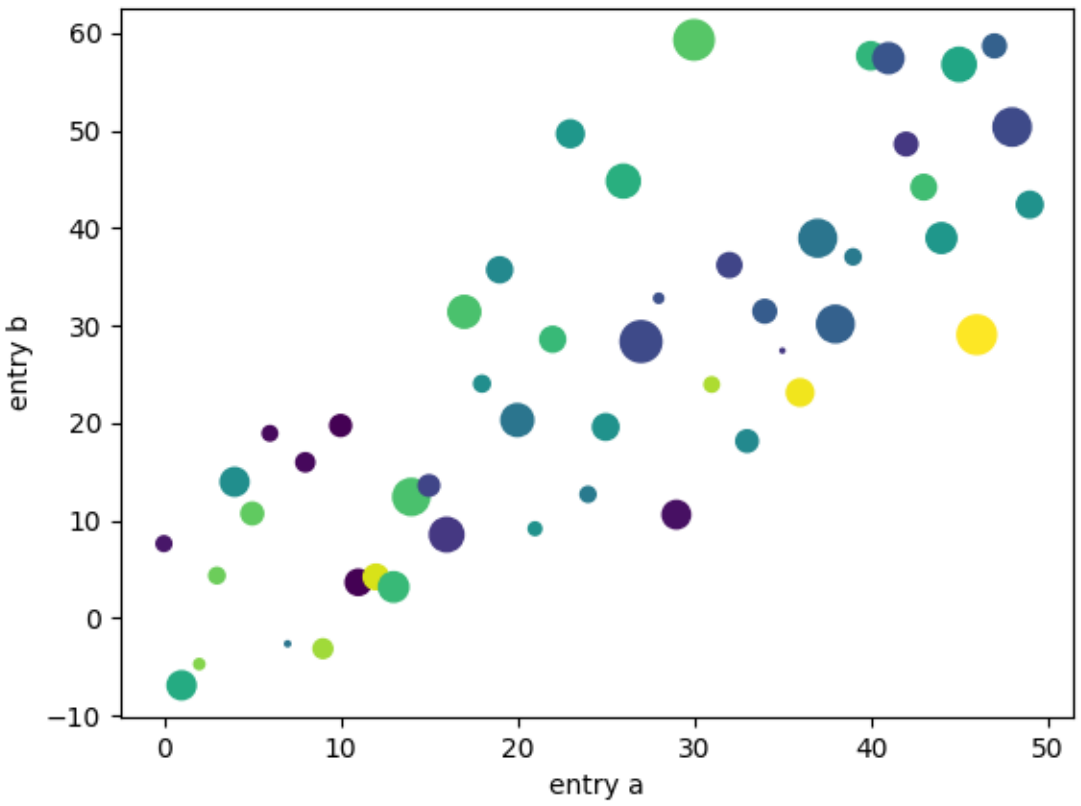


Plotting with keyword strings

There are some instances where you have data in a format that lets you access particular variables with strings. For example, with `numpy.recarray` or `pandas.DataFrame`.

Matplotlib allows you provide such an object with the `data` keyword argument. If provided, then you may generate plots with the strings corresponding to these variables.

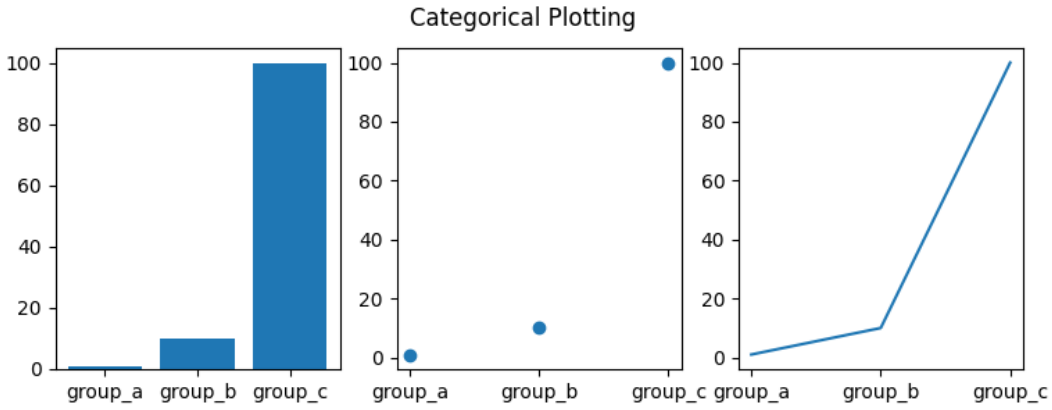
```
data = {'a': np.arange(50),
        'c': np.random.randint(0, 50, 50),
        'd': np.random.randn(50)}
data['b'] = data['a'] + 10 * np.random.randn(50)
data['d'] = np.abs(data['d']) * 100

plt.scatter('a', 'b', c='c', s='d', data=data)
plt.xlabel('entry a')
plt.ylabel('entry b')
plt.show()
```



Plotting with categorical variables

It is also possible to create a plot using categorical variables. Matplotlib allows you to pass categorical variables directly to many plotting functions. For example:

```
names = ['group_a', 'group_b', 'group_c']
values = [1, 10, 100]

plt.figure(figsize=(9, 3))

plt.subplot(131)
plt.bar(names, values)
plt.subplot(132)
plt.scatter(names, values)
plt.subplot(133)
plt.plot(names, values)
plt.suptitle('Categorical Plotting')
plt.show()
```



Controlling line properties

Lines have many attributes that you can set: linewidth, dash style, antialiased, etc; see *matplotlib.lines.Line2D*. There are several ways to set line properties

- Use keyword args:

```
plt.plot(x, y, linewidth=2.0)
```

- Use the setter methods of a `Line2D` instance. `plot` returns a list of `Line2D` objects; e.g., `line1, line2 = plot(x1, y1, x2, y2)`. In the code below we will suppose that we have only one line so that the list returned is of length 1. We use tuple unpacking with `line,` to get the first element of that list:

```
line, = plt.plot(x, y, '-')
line.set_antialiased(False) # turn off antialiasing
```

- Use *setp*. The example below uses a MATLAB-style function to set multiple properties on a list of lines. `setp` works transparently with a list of objects or a single object. You can either use python keyword arguments or MATLAB-style string/value pairs:

```
lines = plt.plot(x1, y1, x2, y2)
# use keyword args
plt.setp(lines, color='r', linewidth=2.0)
# or MATLAB style string value pairs
plt.setp(lines, 'color', 'r', 'linewidth', 2.0)
```

Here are the available *Line2D* properties.

Property	Value Type
alpha	float
animated	[True | False]
antialiased or aa	[True | False]
clip_box	a matplotlib.transform.Bbox instance

continues on next page

Table 1 – continued from previous page

Property	Value Type
clip_on	[True | False]
clip_path	a Path instance and a Transform instance, a Patch
color or c	any matplotlib color
contains	the hit testing function
dash_capstyle	['butt' | 'round' | 'projecting']
dash_joinstyle	['miter' | 'round' | 'bevel']
dashes	sequence of on/off ink in points
data	(np.array xdata, np.array ydata)
figure	a matplotlib.figure.Figure instance
label	any string
linestyle or ls	['-' | '--' | '-.' | ':' | 'steps' | ...]
linewidth or lw	float value in points
marker	['+' | ',' | '.' | '1' | '2' | '3' | '4']
markeredgecolor or mec	any matplotlib color
markeredgewidth or mew	float value in points
markerfacecolor or mfc	any matplotlib color
markersize or ms	float
markevery	[None | integer | (startind, stride)]
picker	used in interactive line selection
pickradius	the line pick selection radius
solid_capstyle	['butt' | 'round' | 'projecting']
solid_joinstyle	['miter' | 'round' | 'bevel']
transform	a matplotlib.transforms.Transform instance
visible	[True | False]
xdata	np.array
ydata	np.array
zorder	any number

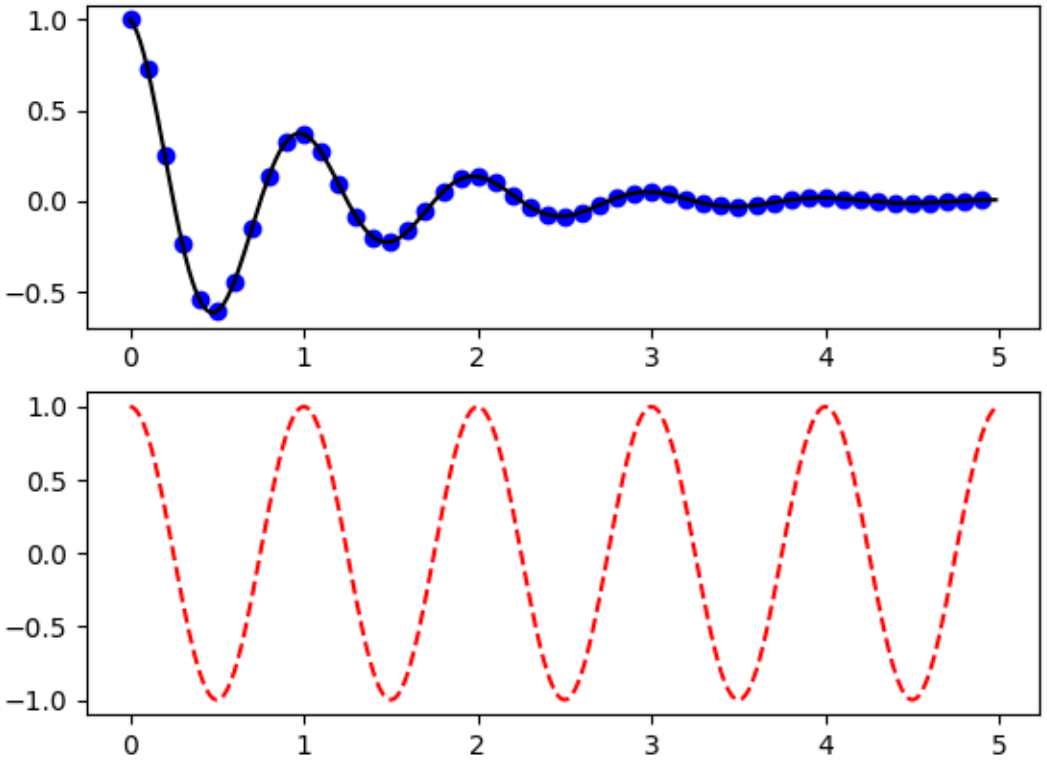
To get a list of settable line properties, call the *setp* function with a line or lines as argument

```
In [69]: lines = plt.plot([1, 2, 3])

In [70]: plt.setp(lines)
  alpha: float
  animated: [True | False]
  antialiased or aa: [True | False]
  ...snip
```

Working with multiple figures and axes

MATLAB, and *pyplot*, have the concept of the current figure and the current axes. All plotting functions apply to the current axes. The function *gca* returns the current axes (a *matplotlib.axes.Axes* instance), and *gcf* returns the current figure (a *matplotlib.figure.Figure* instance). Normally, you don't have to worry about this, because it is all taken care of behind the scenes. Below is a script to create two subplots.

```
def f(t):
    return np.exp(-t) * np.cos(2*np.pi*t)

t1 = np.arange(0.0, 5.0, 0.1)
t2 = np.arange(0.0, 5.0, 0.02)

plt.figure()
plt.subplot(211)
plt.plot(t1, f(t1), 'bo', t2, f(t2), 'k')

plt.subplot(212)
plt.plot(t2, np.cos(2*np.pi*t2), 'r--')
plt.show()
```



The *figure* call here is optional because `figure(1)` will be created by default, just as

a `subplot(111)` will be created by default if you don't manually specify any axes. The *subplot* call specifies `numrows, numcols, plot_number` where `plot_number` ranges from 1 to `numrows*numcols`. The commas in the `subplot` call are optional if `numrows*numcols<10`. So `subplot(211)` is identical to `subplot(2, 1, 1)`.

You can create an arbitrary number of subplots and axes. If you want to place an axes manually, i.e., not on a rectangular grid, use *axes*, which allows you to specify the location as `axes([left, bottom, width, height])` where all values are in fractional (0 to 1) coordinates. See /gallery/subplots_axes_and_figures/axes_demo for an example of placing axes manually and /gallery/subplots_axes_and_figures/subplot_demo for an example with lots of subplots.

You can create multiple figures by using multiple *figure* calls with an increasing figure number. Of course, each figure can contain as many axes and subplots as your heart desires:

```
import matplotlib.pyplot as plt
plt.figure(1)                # the first figure
plt.subplot(211)             # the first subplot in the first figure
plt.plot([1, 2, 3])
plt.subplot(212)             # the second subplot in the first figure
plt.plot([4, 5, 6])


plt.figure(2)                # a second figure
plt.plot([4, 5, 6])          # creates a subplot(111) by default

plt.figure(1)                # figure 1 current; subplot(212) still current
plt.subplot(211)             # make subplot(211) in figure1 current
plt.title('Easy as 1, 2, 3') # subplot 211 title
```

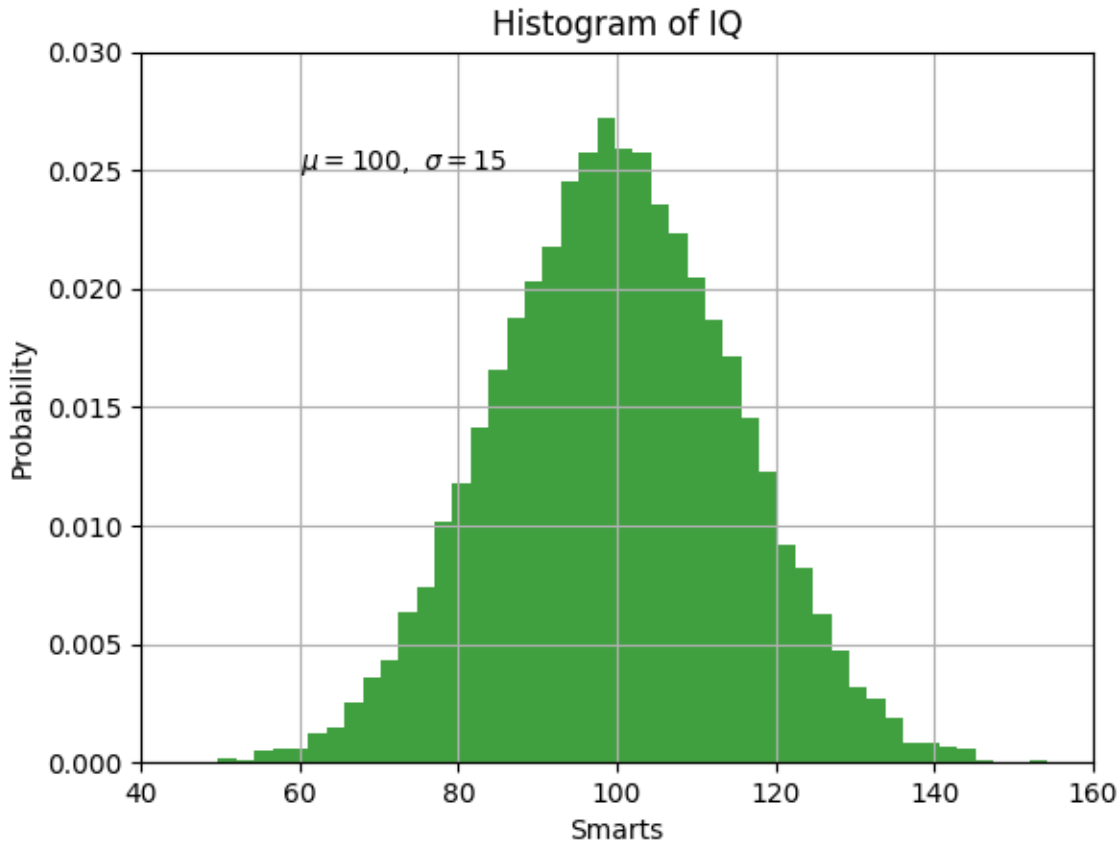
You can clear the current figure with *clf* and the current axes with *cla*. If you find it annoying that states (specifically the current image, figure and axes) are being maintained for you behind the scenes, don't despair: this is just a thin stateful wrapper around an object oriented API, which you can use instead (see *Artist tutorial*)

If you are making lots of figures, you need to be aware of one more thing: the memory required for a figure is not completely released until the figure is explicitly closed with *close*. Deleting all references to the figure, and/or using the window manager to kill the window in which the figure appears on the screen, is not enough, because pyplot maintains internal references until *close* is called.

Working with text

text can be used to add text in an arbitrary location, and *xlabel*, *ylabel* and *title* are used to add text in the indicated locations (see *Text in Matplotlib Plots* for a more detailed example)

```
mu, sigma = 100, 15
x = mu + sigma * np.random.randn(10000)

# the histogram of the data
n, bins, patches = plt.hist(x, 50, density=1, facecolor='g', alpha=0.75)


plt.xlabel('Smarts')
plt.ylabel('Probability')
plt.title('Histogram of IQ')
plt.text(60, .025, r'$\mu=100,\ \sigma=15$')
plt.axis([40, 160, 0, 0.03])
plt.grid(True)
plt.show()
```



All of the *text* functions return a *matplotlib.text.Text* instance. Just as with with lines above, you can customize the properties by passing keyword arguments into the text

functions or using *setp*:

```
t = plt.xlabel('my data', fontsize=14, color='red')
```

These properties are covered in more detail in *Text properties and layout*.

Using mathematical expressions in text

matplotlib accepts TeX equation expressions in any text expression. For example to write the expression $\sigma_i = 15$ in the title, you can write a TeX expression surrounded by dollar signs:

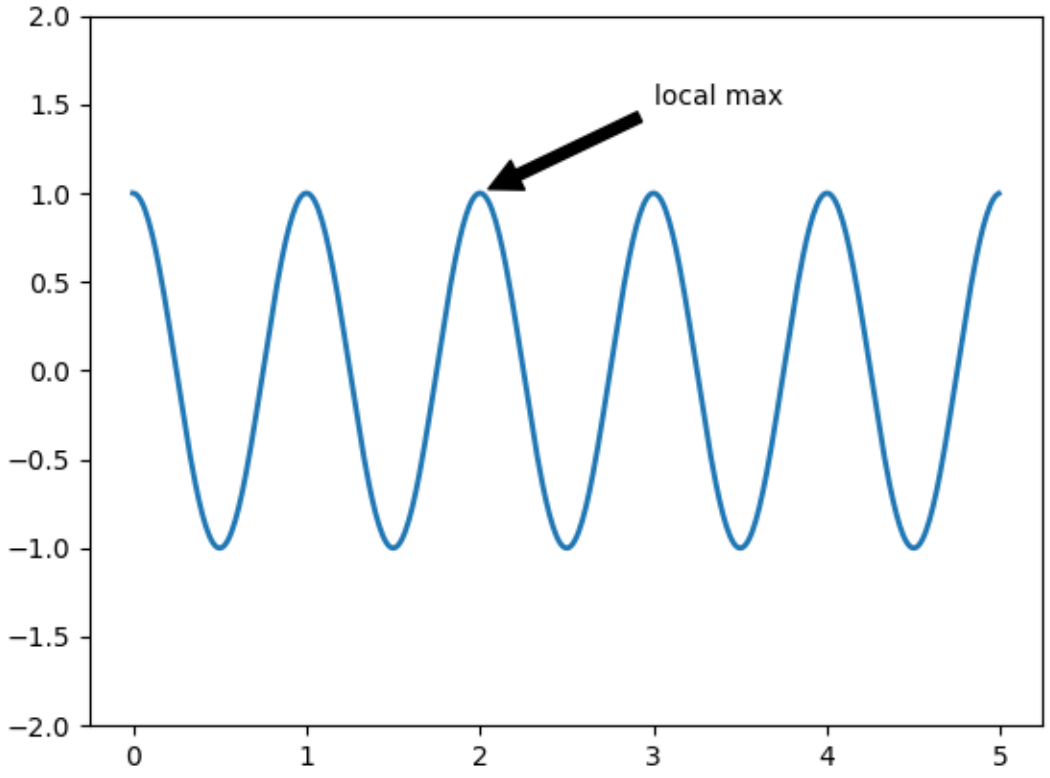
```
plt.title(r'$\sigma_i=15$')
```

The `r` preceding the title string is important -- it signifies that the string is a *raw* string and not to treat backslashes as python escapes. matplotlib has a built-in TeX expression parser and layout engine, and ships its own math fonts -- for details see *Writing mathematical expressions*. Thus you can use mathematical text across platforms without requiring a TeX installation. For those who have LaTeX and dvipng installed, you can also use LaTeX to format your text and incorporate the output directly into your display figures or saved postscript -- see *Text rendering With LaTeX*.

Annotating text

The uses of the basic *text* function above place text at an arbitrary position on the Axes. A common use for text is to annotate some feature of the plot, and the *annotate* method provides helper functionality to make annotations easy. In an annotation, there are two points to consider: the location being annotated represented by the argument `xy` and the location of the text `xytext`. Both of these arguments are `(x, y)` tuples.

```
ax = plt.subplot(111)

t = np.arange(0.0, 5.0, 0.01)
s = np.cos(2*np.pi*t)
line, = plt.plot(t, s, lw=2)

plt.annotate('local max', xy=(2, 1), xytext=(3, 1.5),
             arrowprops=dict(facecolor='black', shrink=0.05),
             )

plt.ylim(-2, 2)
plt.show()
```



In this basic example, both the `xy` (arrow tip) and `xytext` locations (text location) are in data coordinates. There are a variety of other coordinate systems one can choose -- see *Basic annotation* and *Advanced Annotations* for details. More examples can be found in /gallery/text_labels_and_annotations/annotation_demo.

Logarithmic and other nonlinear axes

matplotlib.pyplot supports not only linear axis scales, but also logarithmic and logit scales. This is commonly used if data spans many orders of magnitude. Changing the scale of an axis is easy:

```
plt.xscale('log')
```

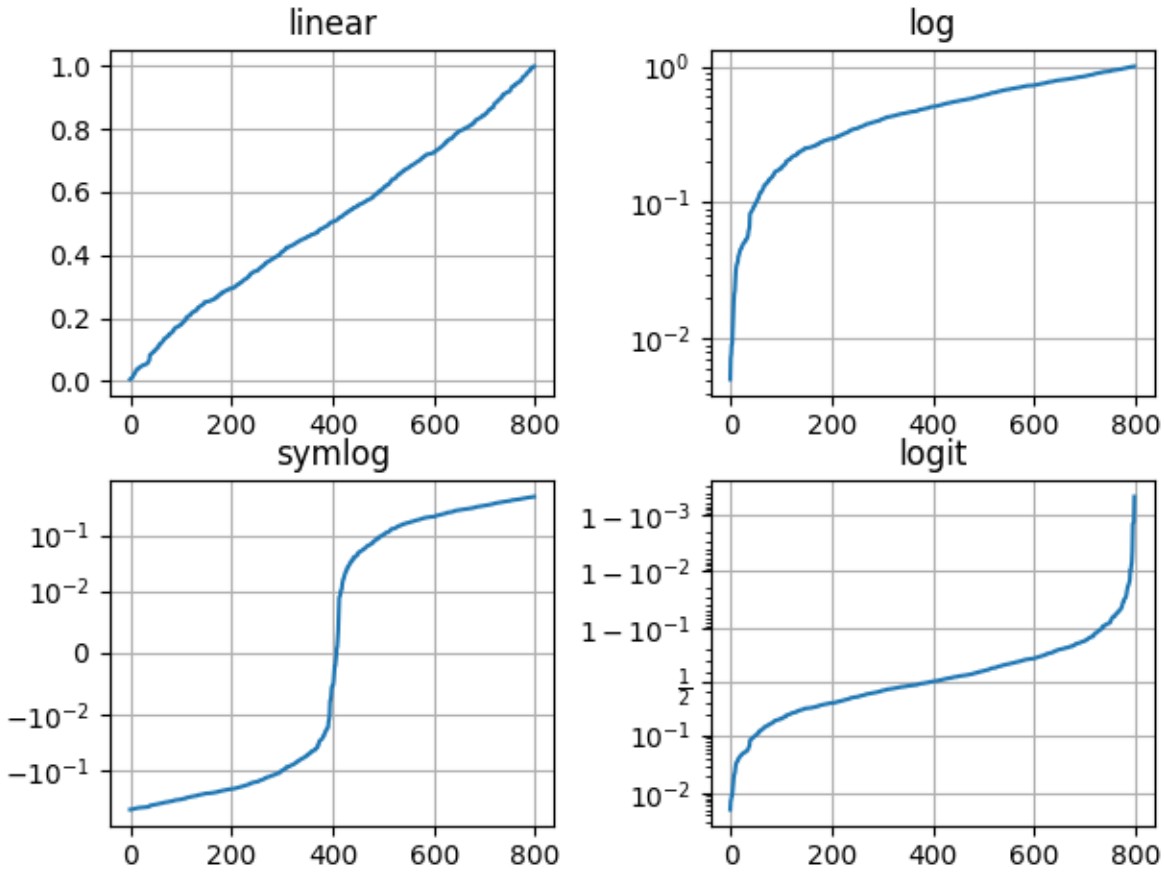
An example of four plots with the same data and different scales for the y axis is shown below.

```
# Fixing random state for reproducibility
np.random.seed(19680801)

# make up some data in the open interval (0, 1)
y = np.random.normal(loc=0.5, scale=0.4, size=1000)
```

(continues on next page)

(continued from previous page)

```
y = y[(y > 0) & (y < 1)]
y.sort()
x = np.arange(len(y))

# plot with various axes scales
plt.figure()

# linear
plt.subplot(221)
plt.plot(x, y)
plt.yscale('linear')
plt.title('linear')
plt.grid(True)

# log
plt.subplot(222)
plt.plot(x, y)
plt.yscale('log')
plt.title('log')
plt.grid(True)

# symmetric log
plt.subplot(223)
plt.plot(x, y - y.mean())
plt.yscale('symlog', linthresh=0.01)
plt.title('symlog')
plt.grid(True)

# logit
plt.subplot(224)
plt.plot(x, y)
plt.yscale('logit')
plt.title('logit')
plt.grid(True)
# Adjust the subplot layout, because the logit one may take more space
# than usual, due to y-tick labels like "1 - 10^{-3}"
plt.subplots_adjust(top=0.92, bottom=0.08, left=0.10, right=0.95, hspace=0.25,
                    wspace=0.35)

plt.show()
```



It is also possible to add your own scale, see *Developer's guide for creating scales and transformations* for details.

Total running time of the script: (0 minutes 3.482 seconds)

2.1.3 Sample plots in Matplotlib

Here you'll find a host of example plots with the code that generated them.

Line Plot

Here's how to create a line plot with text labels using `plot()`.

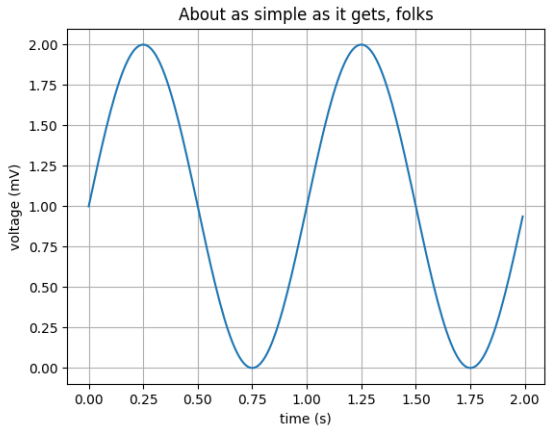


Fig. 1: Simple Plot

Multiple subplots in one figure

Multiple axes (i.e. subplots) are created with the *subplot()* function:

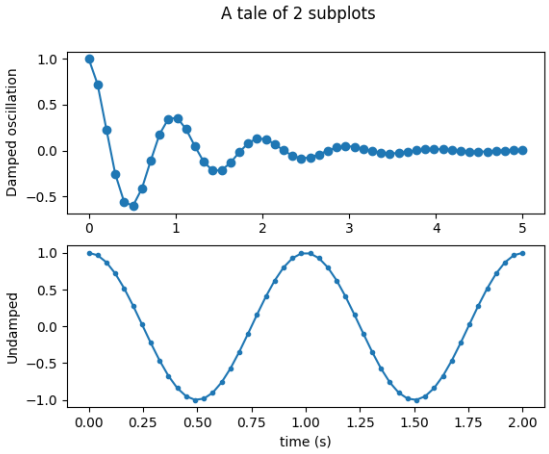


Fig. 2: Subplot

Images

Matplotlib can display images (assuming equally spaced horizontal dimensions) using the *imshow()* function.

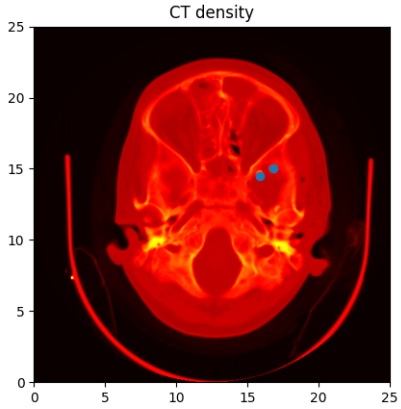


Fig. 3: Example of using *imshow()* to display a CT scan

Contouring and pseudocolor

The *pcolormesh()* function can make a colored representation of a two-dimensional array, even if the horizontal dimensions are unevenly spaced. The *contour()* function is another way to represent the same data:

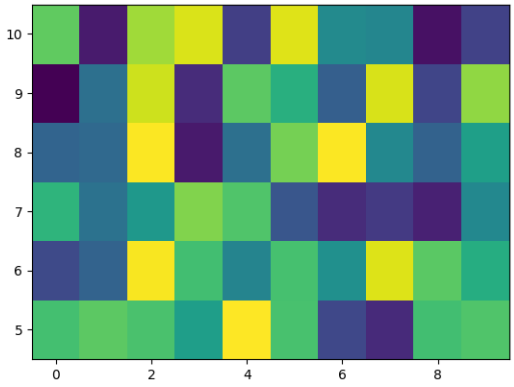


Fig. 4: Example comparing *pcolormesh()* and *contour()* for plotting two-dimensional data

Histograms

The *hist()* function automatically generates histograms and returns the bin counts or probabilities:

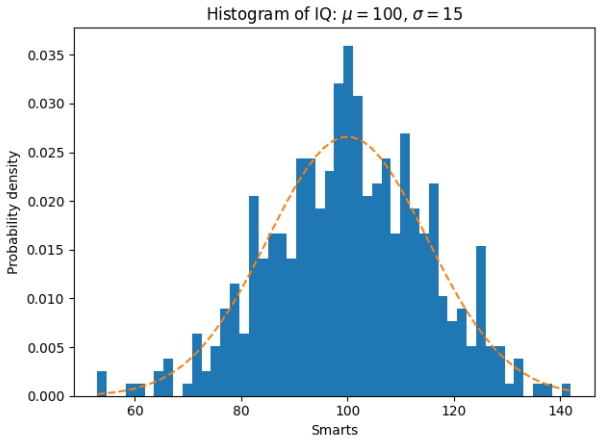


Fig. 5: Histogram Features

Paths

You can add arbitrary paths in Matplotlib using the *matplotlib.path* module:

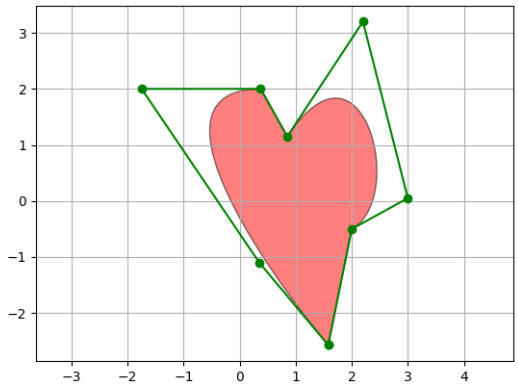


Fig. 6: Path Patch

Three-dimensional plotting

The mplot3d toolkit (see *Getting started* and mplot3d-examples-index) has support for simple 3d graphs including surface, wireframe, scatter, and bar charts.

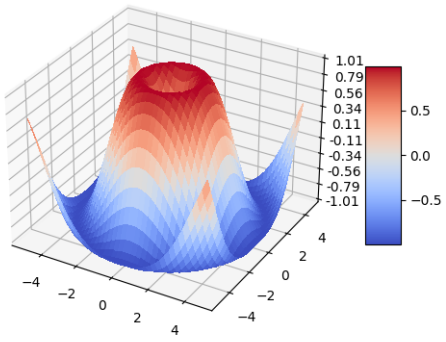


Fig. 7: Surface3d

Thanks to John Porter, Jonathon Taylor, Reinier Heeres, and Ben Root for the *mplot3d* toolkit. This toolkit is included with all standard Matplotlib installs.

Streamplot

The *streamplot()* function plots the streamlines of a vector field. In addition to simply plotting the streamlines, it allows you to map the colors and/or line widths of streamlines to a separate parameter, such as the speed or local intensity of the vector field.

This feature complements the *quiver()* function for plotting vector fields. Thanks to Tom Flannaghan and Tony Yu for adding the streamplot function.

Ellipses

In support of the Phoenix mission to Mars (which used Matplotlib to display ground tracking of spacecraft), Michael Droettboom built on work by Charlie Moad to provide an extremely accurate 8-spline approximation to elliptical arcs (see *Arc*), which are insensitive to zoom level.

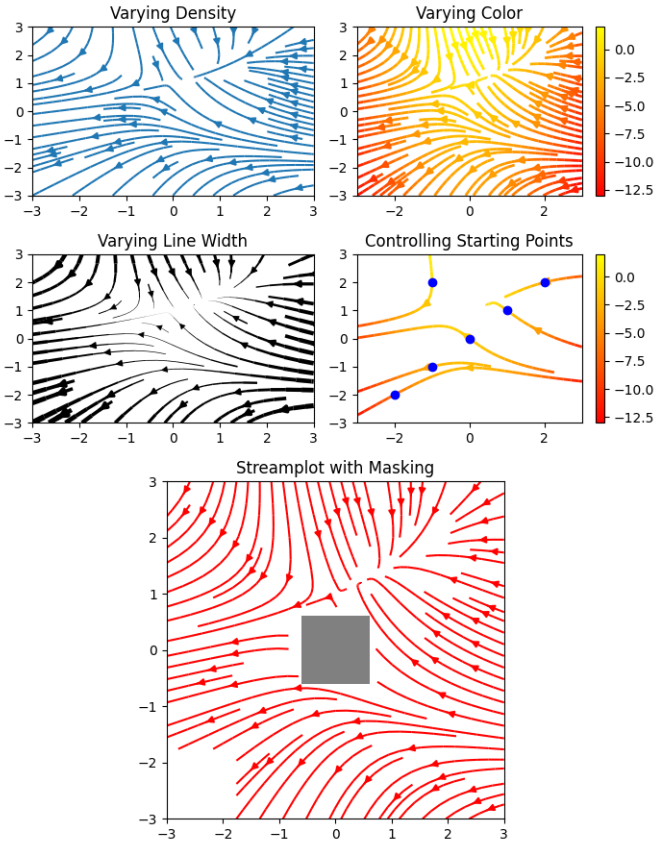


Fig. 8: Streamplot with various plotting options.

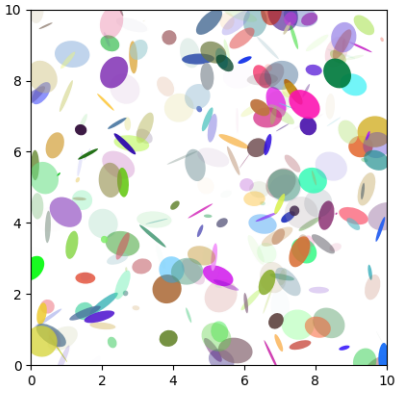


Fig. 9: Ellipse Demo

Bar charts

Use the *bar()* function to make bar charts, which includes customizations such as error bars:

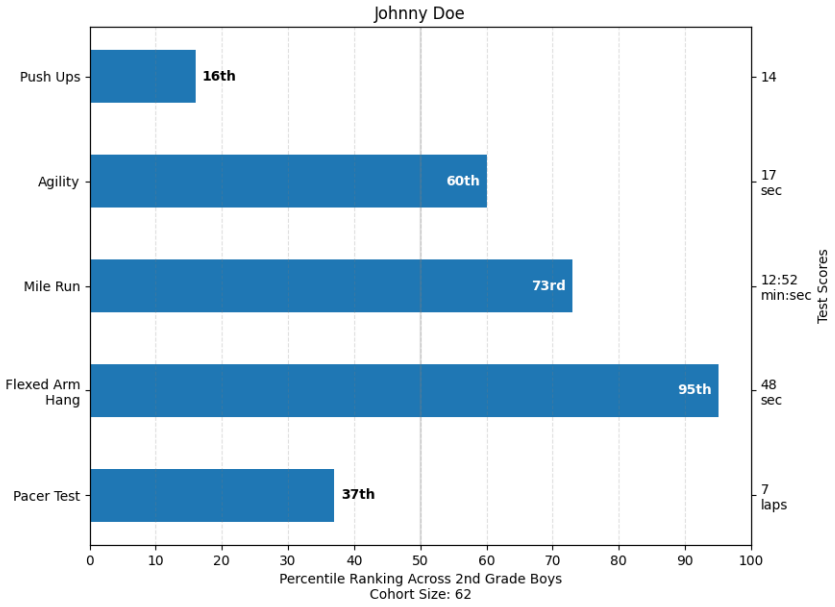


Fig. 10: Barchart Demo

You can also create stacked bars (bar_stacked.py), or horizontal bar charts (barh.py).

Pie charts

The *pie()* function allows you to create pie charts. Optional features include auto-labeling the percentage of area, exploding one or more wedges from the center of the pie, and a shadow effect. Take a close look at the attached code, which generates this figure in just a few lines of code.

Tables

The *table()* function adds a text table to an axes.

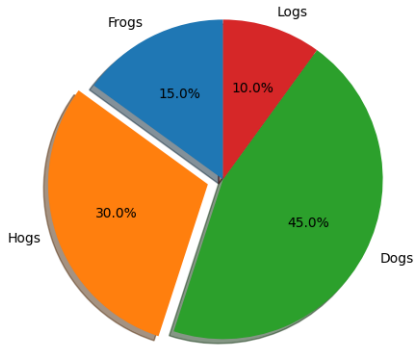


Fig. 11: Pie Features

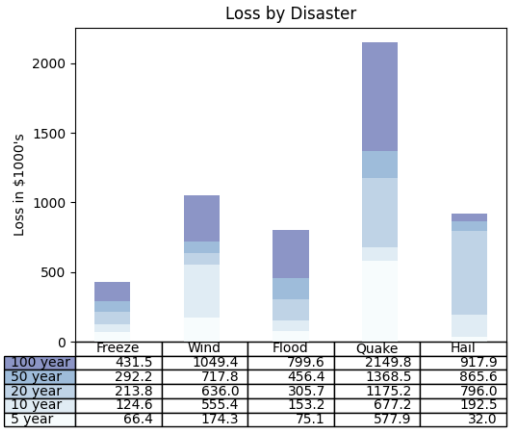


Fig. 12: Table Demo

Scatter plots

The *scatter()* function makes a scatter plot with (optional) size and color arguments. This example plots changes in Google's stock price, with marker sizes reflecting the trading volume and colors varying with time. Here, the alpha attribute is used to make semitransparent circle markers.

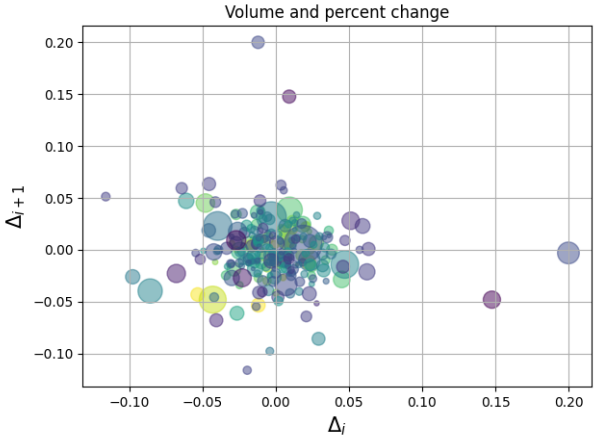


Fig. 13: Scatter Demo2

GUI widgets

Matplotlib has basic GUI widgets that are independent of the graphical user interface you are using, allowing you to write cross GUI figures and widgets. See *matplotlib.widgets* and the widget examples.

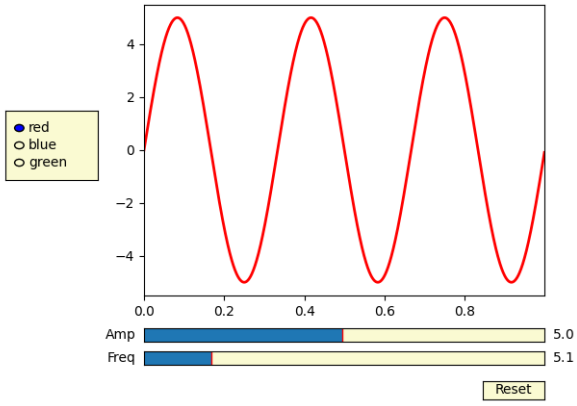


Fig. 14: Slider and radio-button GUI.

Filled curves

The *fill()* function lets you plot filled curves and polygons:

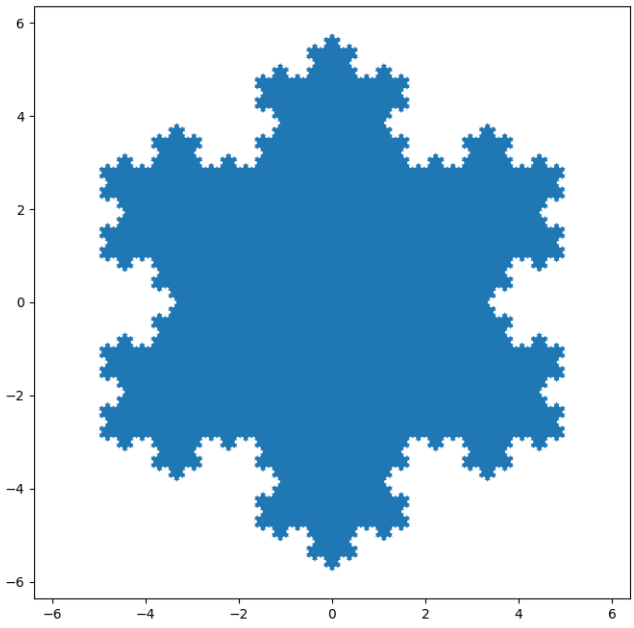


Fig. 15: Fill

Thanks to Andrew Straw for adding this function.

Date handling

You can plot timeseries data with major and minor ticks and custom tick formatters for both.

See *matplotlib.ticker* and *matplotlib.dates* for details and usage.

Log plots

The *semilogx()*, *semilogy()* and *loglog()* functions simplify the creation of logarithmic plots.

Thanks to Andrew Straw, Darren Dale and Gregory Lielens for contributions log-scaling infrastructure.

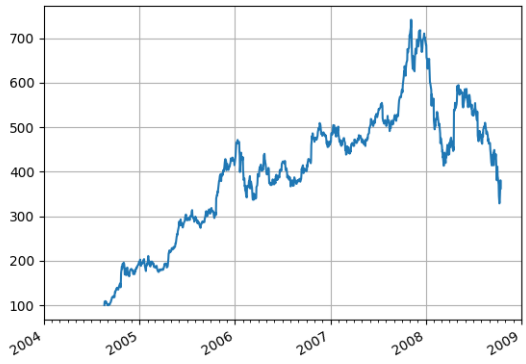


Fig. 16: Date

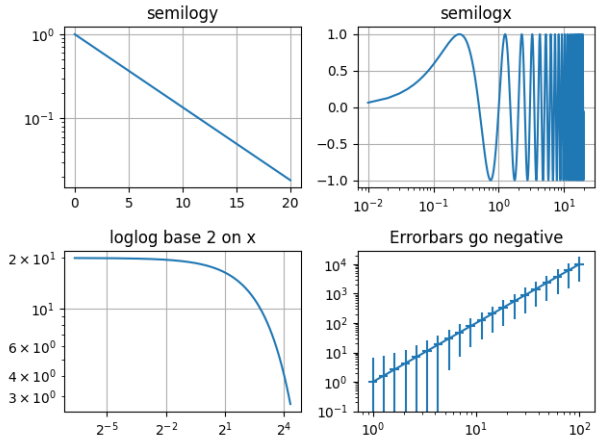


Fig. 17: Log Demo

Polar plots

The *polar()* function generates polar plots.

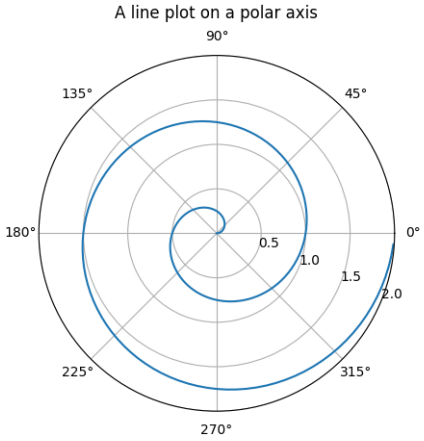


Fig. 18: Polar Demo

Legends

The *legend()* function automatically generates figure legends, with MATLAB-compatible legend-placement functions.

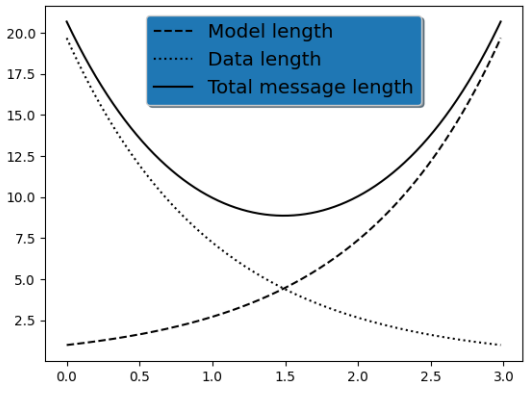


Fig. 19: Legend

Thanks to Charles Twardy for input on the legend function.

TeX-notation for text objects

Below is a sampling of the many TeX expressions now supported by Matplotlib's internal mathtext engine. The mathtext module provides TeX style mathematical expressions using FreeType and the DejaVu, BaKoMa computer modern, or STIX fonts. See the *matplotlib.mathtext* module for additional details.

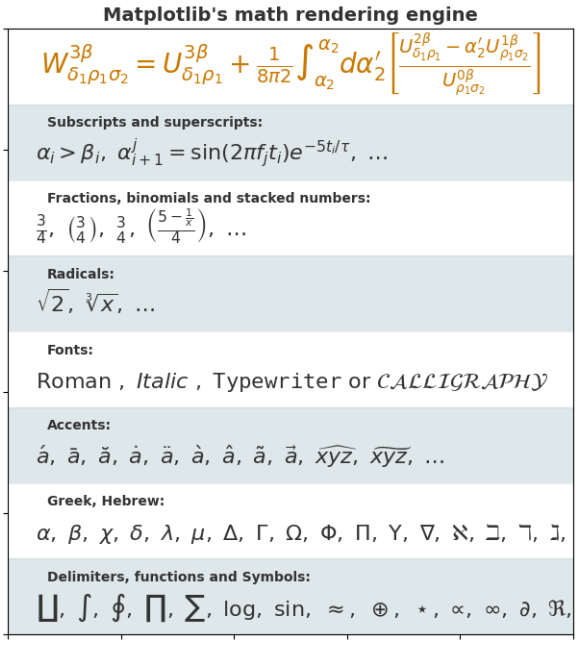


Fig. 20: Mathtext Examples

Matplotlib's mathtext infrastructure is an independent implementation and does not require TeX or any external packages installed on your computer. See the tutorial at *Writing mathematical expressions*.

Native TeX rendering

Although Matplotlib's internal math rendering engine is quite powerful, sometimes you need TeX. Matplotlib supports external TeX rendering of strings with the *usetex* option.

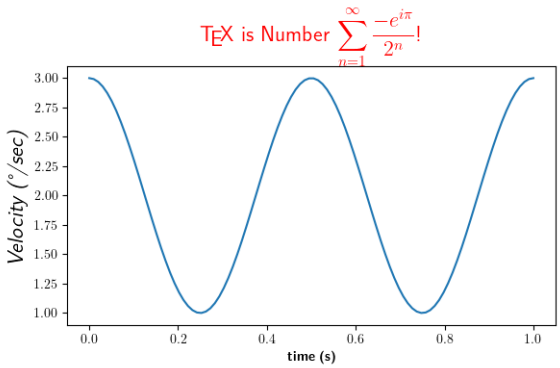
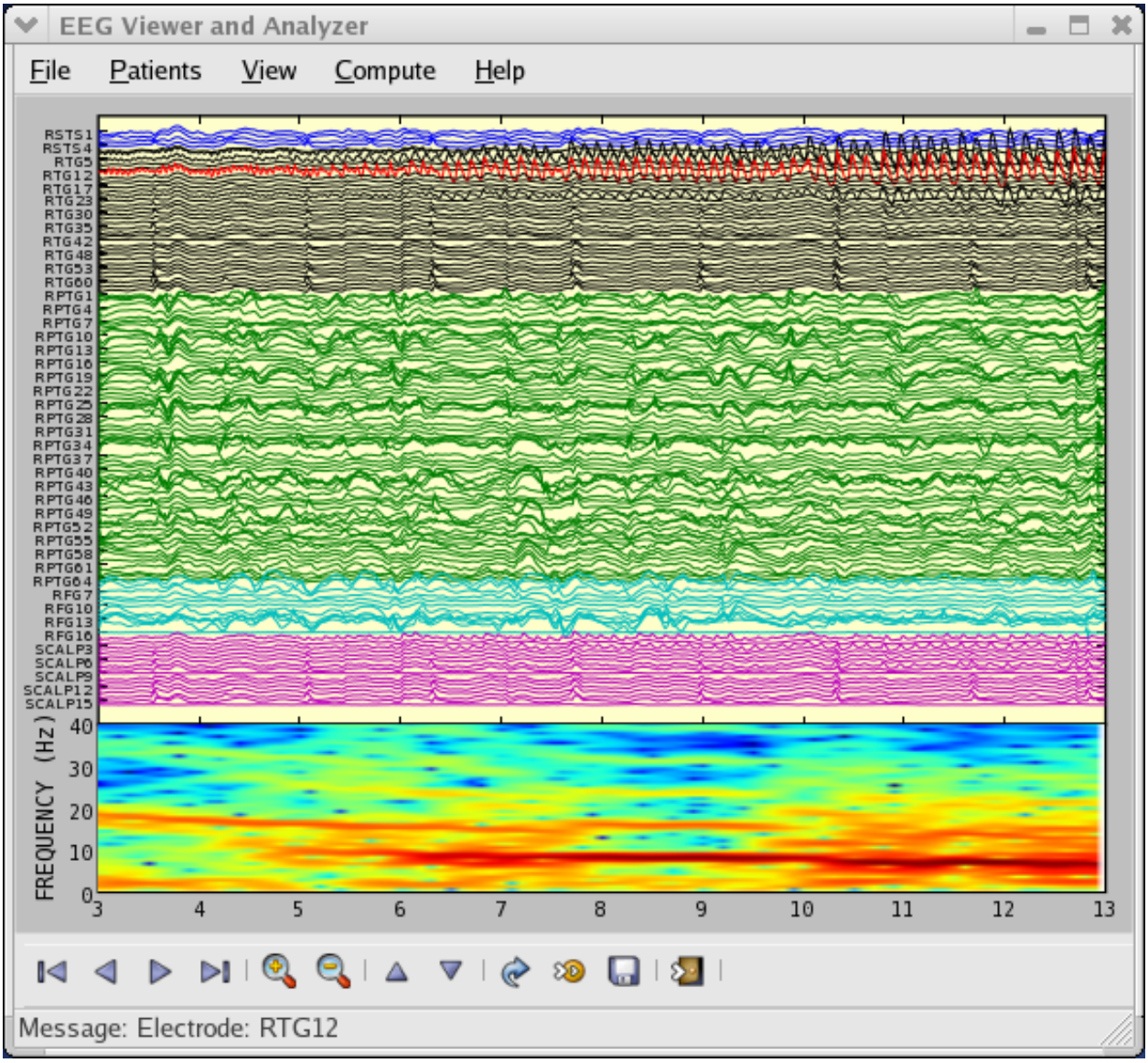


Fig. 21: Tex Demo

EEG GUI

You can embed Matplotlib into pygtk, wx, Tk, or Qt applications. Here is a screenshot of an EEG viewer called pbrain.



The lower axes uses *specgram()* to plot the spectrogram of one of the EEG channels.

For examples of how to embed Matplotlib in different toolkits, see:

- /gallery/user_interfaces/embedding_in_gtk3_sgskip
- /gallery/user_interfaces/embedding_in_wx2_sgskip
- /gallery/user_interfaces/mpl_with_glade3_sgskip
- /gallery/user_interfaces/embedding_in_qt_sgskip
- /gallery/user_interfaces/embedding_in_tk_sgskip

XKCD-style sketch plots

Just for fun, Matplotlib supports plotting in the style of xkcd.

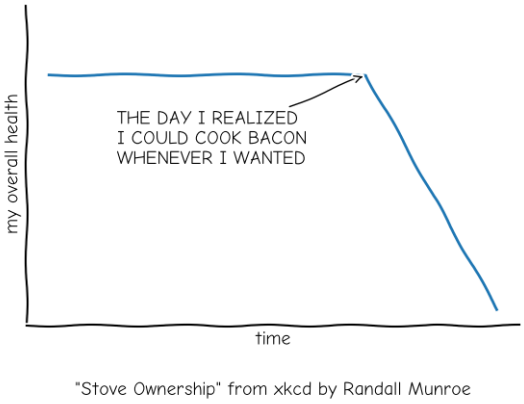
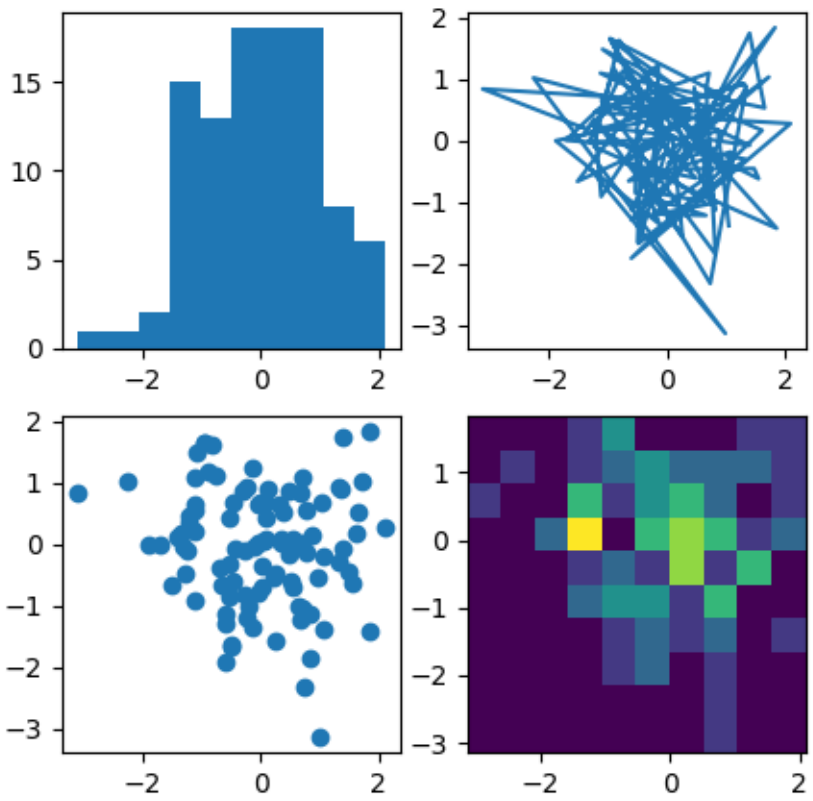


Fig. 22: xkcd

Subplot example

Many plot types can be combined in one figure to create powerful and flexible representations of data.




```
import matplotlib.pyplot as plt
import numpy as np

np.random.seed(19680801)
data = np.random.randn(2, 100)

fig, axs = plt.subplots(2, 2, figsize=(5, 5))
axs[0, 0].hist(data[0])
axs[1, 0].scatter(data[0], data[1])
axs[0, 1].plot(data[0], data[1])
axs[1, 1].hist2d(data[0], data[1])

plt.show()
```

2.1.4 Image tutorial

A short tutorial on plotting images with Matplotlib.

Startup commands

First, let's start IPython. It is a most excellent enhancement to the standard Python prompt, and it ties in especially well with Matplotlib. Start IPython either directly at a shell, or with the Jupyter Notebook (where IPython as a running kernel).

With IPython started, we now need to connect to a GUI event loop. This tells IPython where (and how) to display plots. To connect to a GUI loop, execute the **%matplotlib** magic at your IPython prompt. There's more detail on exactly what this does at IPython's documentation on GUI event loops.

If you're using Jupyter Notebook, the same commands are available, but people commonly use a specific argument to the %matplotlib magic:

```
In [1]: %matplotlib inline
```

This turns on inline plotting, where plot graphics will appear in your notebook. This has important implications for interactivity. For inline plotting, commands in cells below the cell that outputs a plot will not affect the plot. For example, changing the color map is not possible from cells below the cell that creates a plot. However, for other backends, such as Qt5, that open a separate window, cells below those that create the plot will change the plot - it is a live object in memory.

This tutorial will use Matplotlib's imperative-style plotting interface, pyplot. This interface maintains global state, and is very useful for quickly and easily experimenting with various plot settings. The alternative is the object-oriented interface, which is also very powerful, and generally more suitable for large application development. If you'd like to learn about the object-oriented interface, a great place to start is our *Usage guide*. For now, let's get on with the imperative-style approach:

```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
```

Importing image data into Numpy arrays

Matplotlib relies on the Pillow library to load image data.

Here's the image we're going to play with:

It's a 24-bit RGB PNG image (8 bits for each of R, G, B). Depending on where you get your data, the other kinds of image that you'll most likely encounter are RGBA images, which allow for transparency, or single-channel grayscale (luminosity) images. You can right click on it and choose "Save image as" to download it to your computer for the rest of this tutorial.

And here we go...

```
img = mpimg.imread('../../doc/_static/stinkbug.png')
print(img)
```

Out:

```
[[[0.40784314 0.40784314 0.40784314]
  [0.40784314 0.40784314 0.40784314]
  [0.40784314 0.40784314 0.40784314]
  ...
  [0.42745098 0.42745098 0.42745098]
  [0.42745098 0.42745098 0.42745098]
  [0.42745098 0.42745098 0.42745098]]

 [[0.4117647  0.4117647  0.4117647 ]
  [0.4117647  0.4117647  0.4117647 ]
```

(continues on next page)

(continued from previous page)

```
  [0.4117647  0.4117647  0.4117647 ]
  ...
  [0.42745098 0.42745098 0.42745098]
  [0.42745098 0.42745098 0.42745098]
  [0.42745098 0.42745098 0.42745098]]

 [[0.41960785 0.41960785 0.41960785]
  [0.41568628 0.41568628 0.41568628]
  [0.41568628 0.41568628 0.41568628]
  ...
  [0.43137255 0.43137255 0.43137255]
  [0.43137255 0.43137255 0.43137255]
  [0.43137255 0.43137255 0.43137255]]

 ...

 [[0.4392157  0.4392157  0.4392157 ]
  [0.43529412 0.43529412 0.43529412]
  [0.43137255 0.43137255 0.43137255]
  ...
  [0.45490196 0.45490196 0.45490196]
  [0.4509804  0.4509804  0.4509804 ]
  [0.4509804  0.4509804  0.4509804 ]]

 [[0.44313726 0.44313726 0.44313726]
  [0.44313726 0.44313726 0.44313726]
  [0.4392157  0.4392157  0.4392157 ]
  ...
  [0.4509804  0.4509804  0.4509804 ]
  [0.44705883 0.44705883 0.44705883]
  [0.44705883 0.44705883 0.44705883]]

 [[0.44313726 0.44313726 0.44313726]
  [0.4509804  0.4509804  0.4509804 ]
  [0.4509804  0.4509804  0.4509804 ]
  ...
  [0.44705883 0.44705883 0.44705883]
  [0.44705883 0.44705883 0.44705883]
  [0.44313726 0.44313726 0.44313726]]]
```

Note the dtype there - float32. Matplotlib has rescaled the 8 bit data from each channel to floating point data between 0.0 and 1.0. As a side note, the only datatype that Pillow can work with is uint8. Matplotlib plotting can handle float32 and uint8, but image reading/writing for any format other than PNG is limited to uint8 data. Why 8 bits? Most displays can only render 8 bits per channel worth of color gradation. Why can they only render 8 bits/channel? Because that's about all the human eye can see. More here (from a photography standpoint): Luminous Landscape bit depth tutorial.

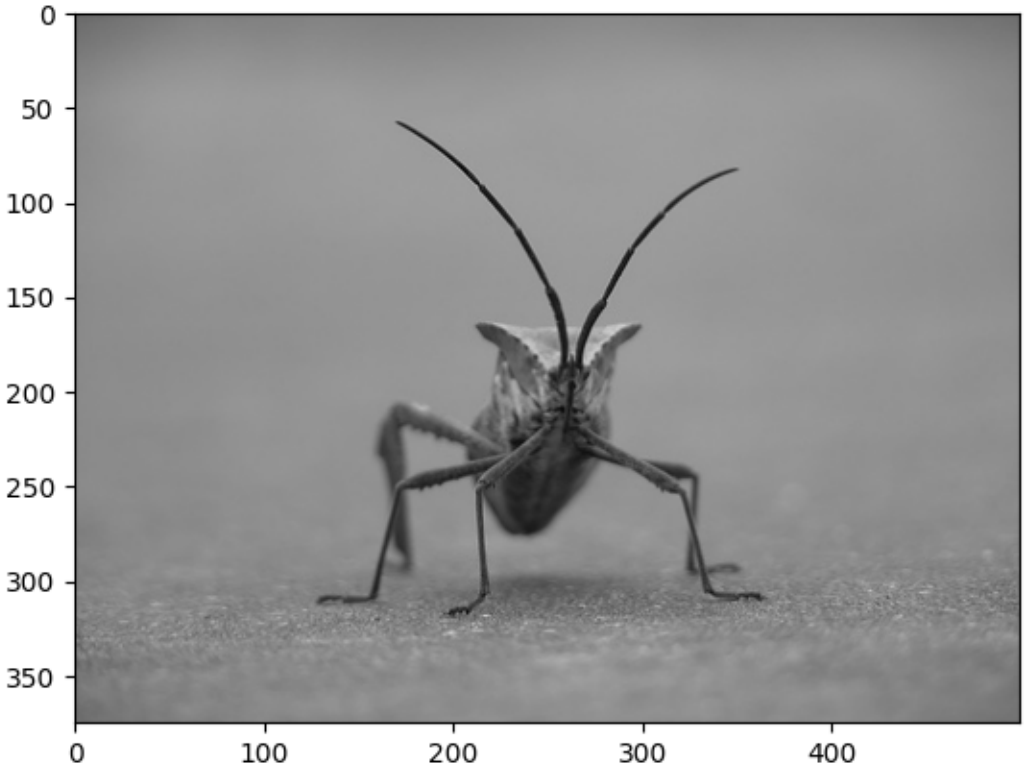
Each inner list represents a pixel. Here, with an RGB image, there are 3 values. Since it's a black and white image, R, G, and B are all similar. An RGBA (where A is alpha, or transparency), has 4 values per inner list, and a simple luminance image just has

one value (and is thus only a 2-D array, not a 3-D array). For RGB and RGBA images, Matplotlib supports float32 and uint8 data types. For grayscale, Matplotlib supports only float32. If your array data does not meet one of these descriptions, you need to rescale it.

Plotting numpy arrays as images

So, you have your data in a numpy array (either by importing it, or by generating it). Let's render it. In Matplotlib, this is performed using the *imshow()* function. Here we'll grab the plot object. This object gives you an easy way to manipulate the plot from the prompt.

```
imgplot = plt.imshow(img)
```



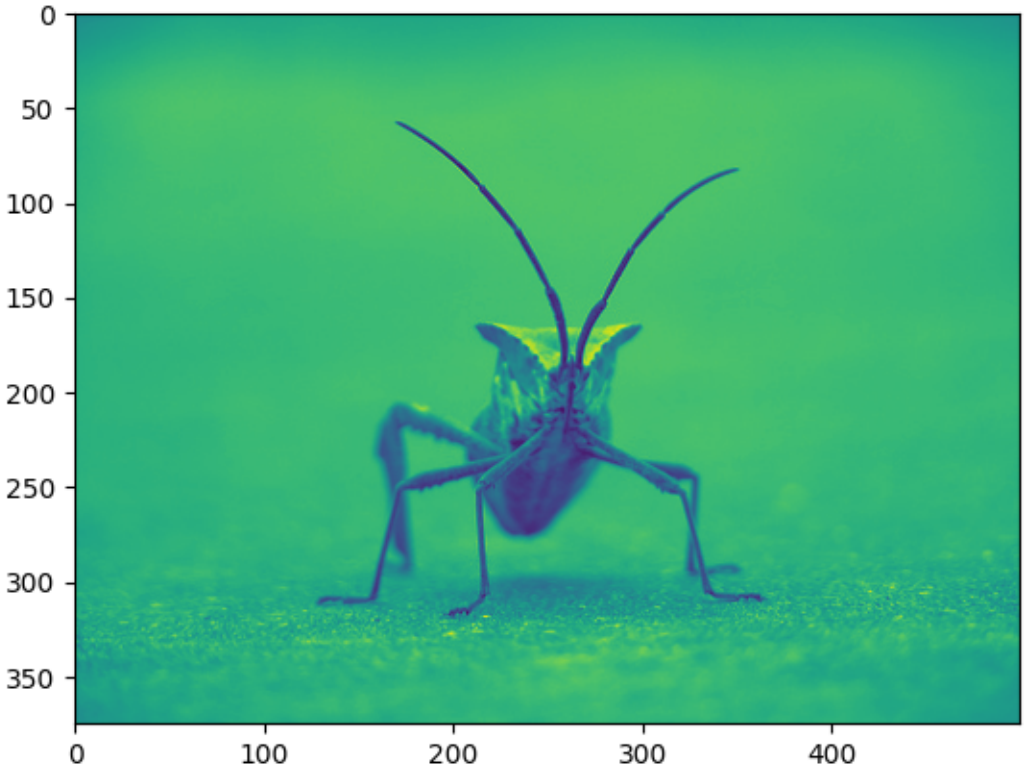
You can also plot any numpy array.

Applying pseudocolor schemes to image plots

Pseudocolor can be a useful tool for enhancing contrast and visualizing your data more easily. This is especially useful when making presentations of your data using projectors - their contrast is typically quite poor.

Pseudocolor is only relevant to single-channel, grayscale, luminosity images. We currently have an RGB image. Since R, G, and B are all similar (see for yourself above or in your data), we can just pick one channel of our data:

```
lum_img = img[:, :, 0]

# This is array slicing.  You can read more in the `Numpy tutorial
# <https://docs.scipy.org/doc/numpy/user/quickstart.html>`_.

plt.imshow(lum_img)
```



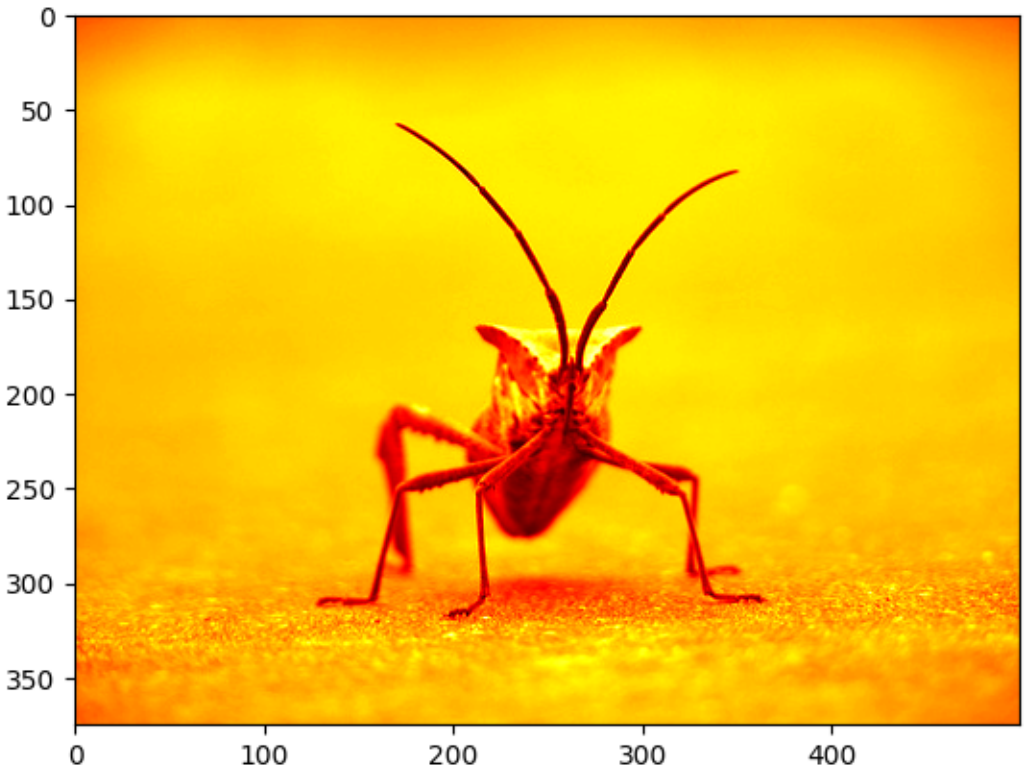
Out:

```
<matplotlib.image.AxesImage object at 0x7f53fb3eb640>
```

Now, with a luminosity (2D, no color) image, the default colormap (aka lookup table, LUT), is applied. The default is called viridis. There are plenty of others to choose

from.

```
plt.imshow(lum_img, cmap="hot")
```

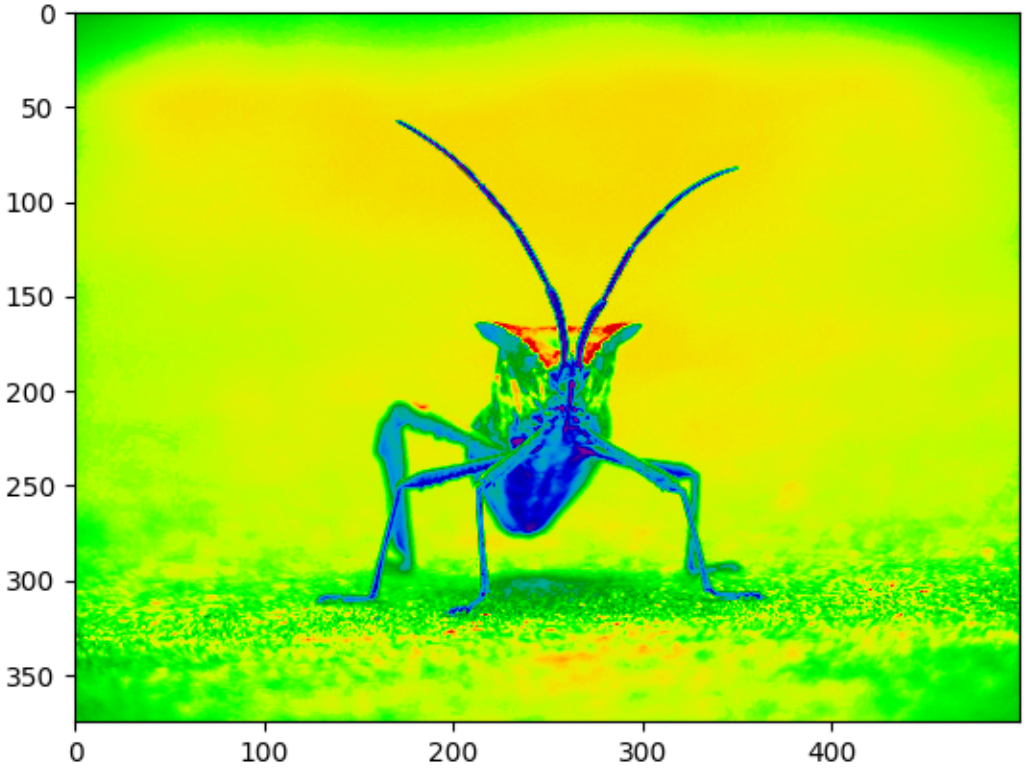


Out:

```
<matplotlib.image.AxesImage object at 0x7f53fed4b400>
```

Note that you can also change colormaps on existing plot objects using the *set_cmap()* method:

```
imgplot = plt.imshow(lum_img)
imgplot.set_cmap('nipy_spectral')
```



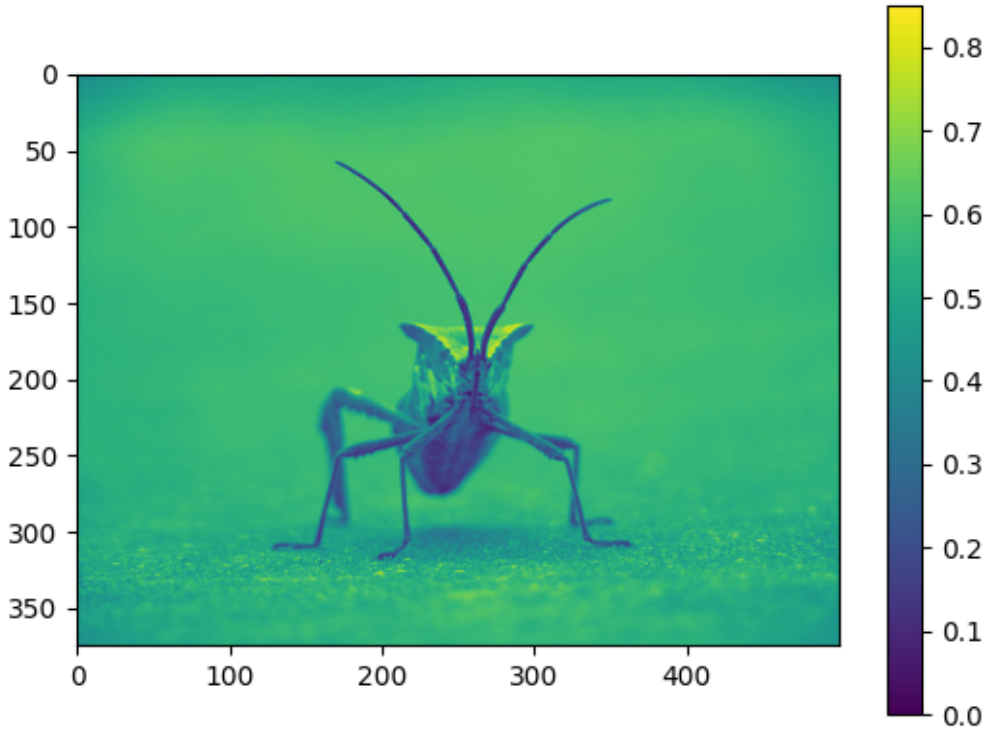
Note: However, remember that in the Jupyter Notebook with the inline backend, you can't make changes to plots that have already been rendered. If you create imgplot here in one cell, you cannot call set_cmap() on it in a later cell and expect the earlier plot to change. Make sure that you enter these commands together in one cell. plt commands will not change plots from earlier cells.

There are many other colormap schemes available. See the list and images of the colormaps.

Color scale reference

It's helpful to have an idea of what value a color represents. We can do that by adding a color bar to your figure:

```
imgplot = plt.imshow(lum_img)
plt.colorbar()
```



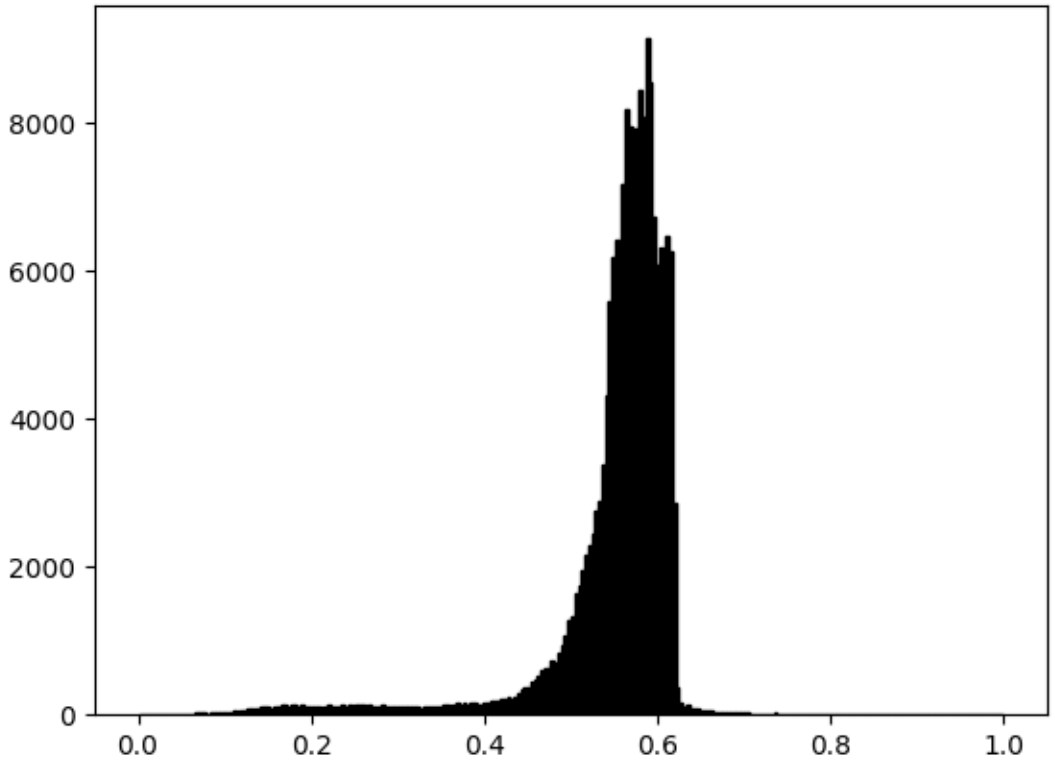
Out:

```
<matplotlib.colorbar.Colorbar object at 0x7f53fed09640>
```

Examining a specific data range

Sometimes you want to enhance the contrast in your image, or expand the contrast in a particular region while sacrificing the detail in colors that don't vary much, or don't matter. A good tool to find interesting regions is the histogram. To create a histogram of our image data, we use the *hist()* function.

```
plt.hist(lum_img.ravel(), bins=256, range=(0.0, 1.0), fc='k', ec='k')
```



Out:

```
(array([2.000e+00, 2.000e+00, 3.000e+00, 3.000e+00, 2.000e+00, 2.000e+00,
       3.000e+00, 1.000e+00, 7.000e+00, 9.000e+00, 7.000e+00, 2.000e+00,
       7.000e+00, 1.000e+01, 1.100e+01, 1.500e+01, 1.400e+01, 2.700e+01,
       2.100e+01, 2.400e+01, 1.400e+01, 3.100e+01, 2.900e+01, 2.800e+01,
       2.400e+01, 2.400e+01, 4.000e+01, 2.600e+01, 5.200e+01, 3.900e+01,
       5.700e+01, 4.600e+01, 8.400e+01, 7.600e+01, 8.900e+01, 8.000e+01,
       1.060e+02, 1.130e+02, 1.120e+02, 9.000e+01, 1.160e+02, 1.090e+02,
       1.270e+02, 1.350e+02, 9.800e+01, 1.310e+02, 1.230e+02, 1.110e+02,
       1.230e+02, 1.160e+02, 1.010e+02, 1.170e+02, 1.000e+02, 1.010e+02,
       9.000e+01, 1.060e+02, 1.260e+02, 1.040e+02, 1.070e+02, 1.110e+02,
       1.380e+02, 1.000e+02, 1.340e+02, 1.210e+02, 1.400e+02, 1.320e+02,
       1.390e+02, 1.160e+02, 1.330e+02, 1.180e+02, 1.080e+02, 1.170e+02,
       1.280e+02, 1.200e+02, 1.210e+02, 1.100e+02, 1.160e+02, 1.180e+02,
       9.700e+01, 9.700e+01, 1.140e+02, 1.070e+02, 1.170e+02, 8.700e+01,
       1.070e+02, 9.800e+01, 1.040e+02, 1.120e+02, 1.110e+02, 1.180e+02,
       1.240e+02, 1.340e+02, 1.200e+02, 1.410e+02, 1.520e+02, 1.360e+02,
       1.610e+02, 1.380e+02, 1.620e+02, 1.570e+02, 1.350e+02, 1.470e+02,
       1.690e+02, 1.710e+02, 1.820e+02, 1.980e+02, 1.970e+02, 2.060e+02,
       2.160e+02, 2.460e+02, 2.210e+02, 2.520e+02, 2.890e+02, 3.450e+02,
       3.620e+02, 3.760e+02, 4.480e+02, 4.630e+02, 5.170e+02, 6.000e+02,
       6.200e+02, 6.410e+02, 7.440e+02, 7.120e+02, 8.330e+02, 9.290e+02,
```

(continues on next page)

(continued from previous page)

```
       1.061e+03, 1.280e+03, 1.340e+03, 1.638e+03, 1.740e+03, 1.953e+03,
       2.151e+03, 2.290e+03, 2.440e+03, 2.758e+03, 2.896e+03, 3.384e+03,
       4.332e+03, 5.584e+03, 6.197e+03, 6.422e+03, 6.404e+03, 7.181e+03,
       8.196e+03, 7.968e+03, 7.474e+03, 7.926e+03, 8.460e+03, 8.091e+03,
       9.148e+03, 8.563e+03, 6.747e+03, 6.074e+03, 6.328e+03, 5.291e+03,
       6.472e+03, 6.268e+03, 2.864e+03, 3.760e+02, 1.620e+02, 1.180e+02,
       1.270e+02, 9.500e+01, 7.600e+01, 8.200e+01, 6.200e+01, 6.700e+01,
       5.600e+01, 5.900e+01, 4.000e+01, 4.200e+01, 3.000e+01, 3.400e+01,
       3.200e+01, 4.300e+01, 4.200e+01, 2.300e+01, 2.800e+01, 1.900e+01,
       2.200e+01, 1.600e+01, 1.200e+01, 1.800e+01, 9.000e+00, 1.000e+01,
       1.700e+01, 5.000e+00, 2.100e+01, 1.300e+01, 8.000e+00, 1.200e+01,
       1.000e+01, 8.000e+00, 8.000e+00, 5.000e+00, 1.300e+01, 6.000e+00,
       3.000e+00, 7.000e+00, 6.000e+00, 2.000e+00, 1.000e+00, 5.000e+00,
       3.000e+00, 3.000e+00, 1.000e+00, 1.000e+00, 1.000e+00, 5.000e+00,
       0.000e+00, 1.000e+00, 3.000e+00, 0.000e+00, 1.000e+00, 1.000e+00,
       2.000e+00, 1.000e+00, 0.000e+00, 0.000e+00, 0.000e+00, 0.000e+00,
       0.000e+00, 0.000e+00, 0.000e+00, 0.000e+00, 0.000e+00, 0.000e+00,
       0.000e+00, 0.000e+00, 0.000e+00, 0.000e+00, 0.000e+00, 0.000e+00,
       0.000e+00, 0.000e+00, 0.000e+00, 0.000e+00, 0.000e+00, 0.000e+00,
       0.000e+00, 0.000e+00, 0.000e+00, 0.000e+00, 0.000e+00, 0.000e+00,
       0.000e+00, 0.000e+00, 0.000e+00, 0.000e+00, 0.000e+00, 0.000e+00,
       0.000e+00, 0.000e+00, 0.000e+00, 0.000e+00]), array([0.        , 0.00390625, 0.
↪0078125 , 0.01171875, 0.015625  ,
       0.01953125, 0.0234375 , 0.02734375, 0.03125   , 0.03515625,
       0.0390625 , 0.04296875, 0.046875  , 0.05078125, 0.0546875 ,
       0.05859375, 0.0625    , 0.06640625, 0.0703125 , 0.07421875,
       0.078125  , 0.08203125, 0.0859375 , 0.08984375, 0.09375   ,
       0.09765625, 0.1015625 , 0.10546875, 0.109375  , 0.11328125,
       0.1171875 , 0.12109375, 0.125     , 0.12890625, 0.1328125 ,
       0.13671875, 0.140625  , 0.14453125, 0.1484375 , 0.15234375,
       0.15625   , 0.16015625, 0.1640625 , 0.16796875, 0.171875  ,
       0.17578125, 0.1796875 , 0.18359375, 0.1875    , 0.19140625,
       0.1953125 , 0.19921875, 0.203125  , 0.20703125, 0.2109375 ,
       0.21484375, 0.21875   , 0.22265625, 0.2265625 , 0.23046875,
       0.234375  , 0.23828125, 0.2421875 , 0.24609375, 0.25      ,
       0.25390625, 0.2578125 , 0.26171875, 0.265625  , 0.26953125,
       0.2734375 , 0.27734375, 0.28125   , 0.28515625, 0.2890625 ,
       0.29296875, 0.296875  , 0.30078125, 0.3046875 , 0.30859375,
       0.3125    , 0.31640625, 0.3203125 , 0.32421875, 0.328125  ,
       0.33203125, 0.3359375 , 0.33984375, 0.34375   , 0.34765625,
       0.3515625 , 0.35546875, 0.359375  , 0.36328125, 0.3671875 ,
       0.37109375, 0.375     , 0.37890625, 0.3828125 , 0.38671875,
       0.390625  , 0.39453125, 0.3984375 , 0.40234375, 0.40625   ,
       0.41015625, 0.4140625 , 0.41796875, 0.421875  , 0.42578125,
       0.4296875 , 0.43359375, 0.4375    , 0.44140625, 0.4453125 ,
       0.44921875, 0.453125  , 0.45703125, 0.4609375 , 0.46484375,
       0.46875   , 0.47265625, 0.4765625 , 0.48046875, 0.484375  ,
       0.48828125, 0.4921875 , 0.49609375, 0.5       , 0.50390625,
       0.5078125 , 0.51171875, 0.515625  , 0.51953125, 0.5234375 ,
       0.52734375, 0.53125   , 0.53515625, 0.5390625 , 0.54296875,
       0.546875  , 0.55078125, 0.5546875 , 0.55859375, 0.5625    ,
```

(continues on next page)

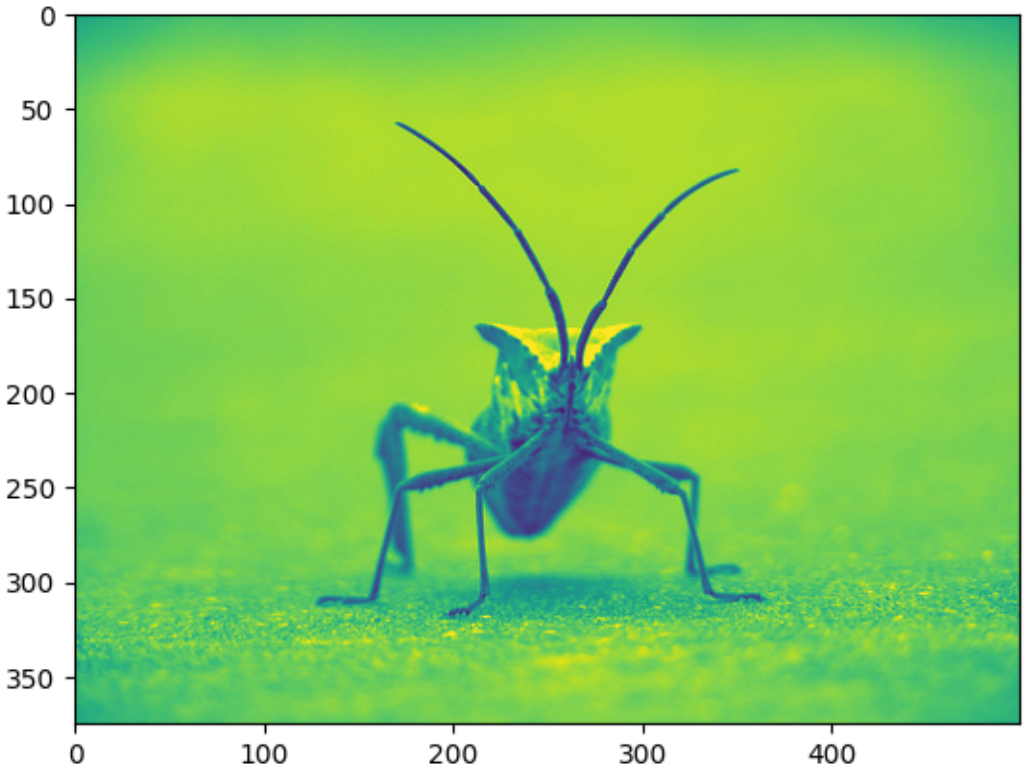
(continued from previous page)

```
       0.56640625, 0.5703125 , 0.57421875, 0.578125  , 0.58203125,
       0.5859375 , 0.58984375, 0.59375   , 0.59765625, 0.6015625 ,
       0.60546875, 0.609375  , 0.61328125, 0.6171875 , 0.62109375,
       0.625     , 0.62890625, 0.6328125 , 0.63671875, 0.640625  ,
       0.64453125, 0.6484375 , 0.65234375, 0.65625   , 0.66015625,
       0.6640625 , 0.66796875, 0.671875  , 0.67578125, 0.6796875 ,
       0.68359375, 0.6875    , 0.69140625, 0.6953125 , 0.69921875,
       0.703125  , 0.70703125, 0.7109375 , 0.71484375, 0.71875   ,
       0.72265625, 0.7265625 , 0.73046875, 0.734375  , 0.73828125,
       0.7421875 , 0.74609375, 0.75      , 0.75390625, 0.7578125 ,
       0.76171875, 0.765625  , 0.76953125, 0.7734375 , 0.77734375,
       0.78125   , 0.78515625, 0.7890625 , 0.79296875, 0.796875  ,
       0.80078125, 0.8046875 , 0.80859375, 0.8125    , 0.81640625,
       0.8203125 , 0.82421875, 0.828125  , 0.83203125, 0.8359375 ,
       0.83984375, 0.84375   , 0.84765625, 0.8515625 , 0.85546875,
       0.859375  , 0.86328125, 0.8671875 , 0.87109375, 0.875     ,
       0.87890625, 0.8828125 , 0.88671875, 0.890625  , 0.89453125,
       0.8984375 , 0.90234375, 0.90625   , 0.91015625, 0.9140625 ,
       0.91796875, 0.921875  , 0.92578125, 0.9296875 , 0.93359375,
       0.9375    , 0.94140625, 0.9453125 , 0.94921875, 0.953125  ,
       0.95703125, 0.9609375 , 0.96484375, 0.96875   , 0.97265625,
       0.9765625 , 0.98046875, 0.984375  , 0.98828125, 0.9921875 ,
       0.99609375, 1.        ], dtype=float32), <BarContainer object of 256 artists>)
```

Most often, the "interesting" part of the image is around the peak, and you can get extra contrast by clipping the regions above and/or below the peak. In our histogram, it looks like there's not much useful information in the high end (not many white things in the image). Let's adjust the upper limit, so that we effectively "zoom in on" part of the histogram. We do this by passing the clim argument to imshow. You could also do this by calling the *set_clim()* method of the image plot object, but make sure that you do so in the same cell as your plot command when working with the Jupyter Notebook - it will not change plots from earlier cells.

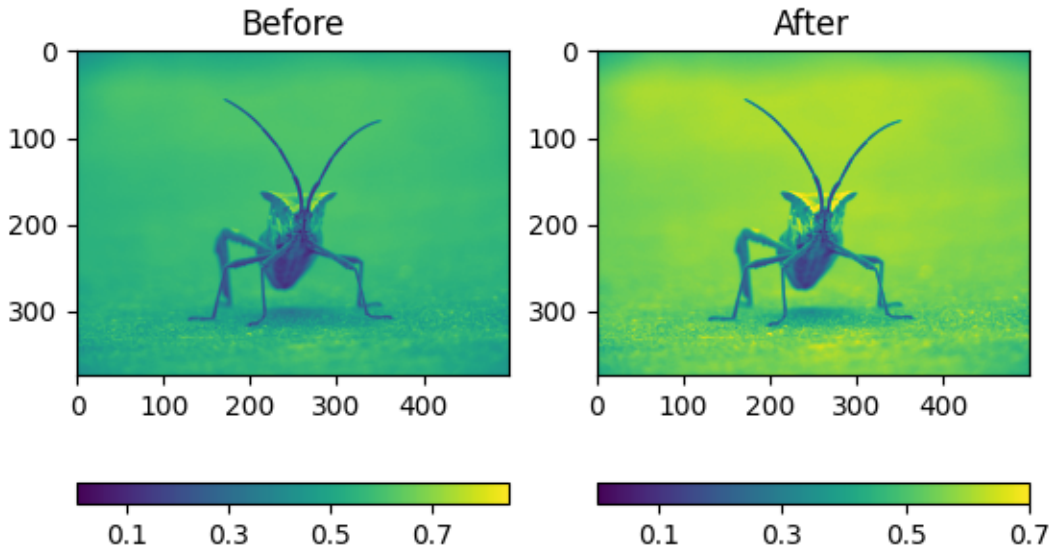
You can specify the clim in the call to `plot`.

```
imgplot = plt.imshow(lum_img, clim=(0.0, 0.7))
```



You can also specify the clim using the returned object

```
fig = plt.figure()
ax = fig.add_subplot(1, 2, 1)
imgplot = plt.imshow(lum_img)
ax.set_title('Before')
plt.colorbar(ticks=[0.1, 0.3, 0.5, 0.7], orientation='horizontal')
ax = fig.add_subplot(1, 2, 2)
imgplot = plt.imshow(lum_img)
imgplot.set_clim(0.0, 0.7)
ax.set_title('After')
plt.colorbar(ticks=[0.1, 0.3, 0.5, 0.7], orientation='horizontal')
```



Out:

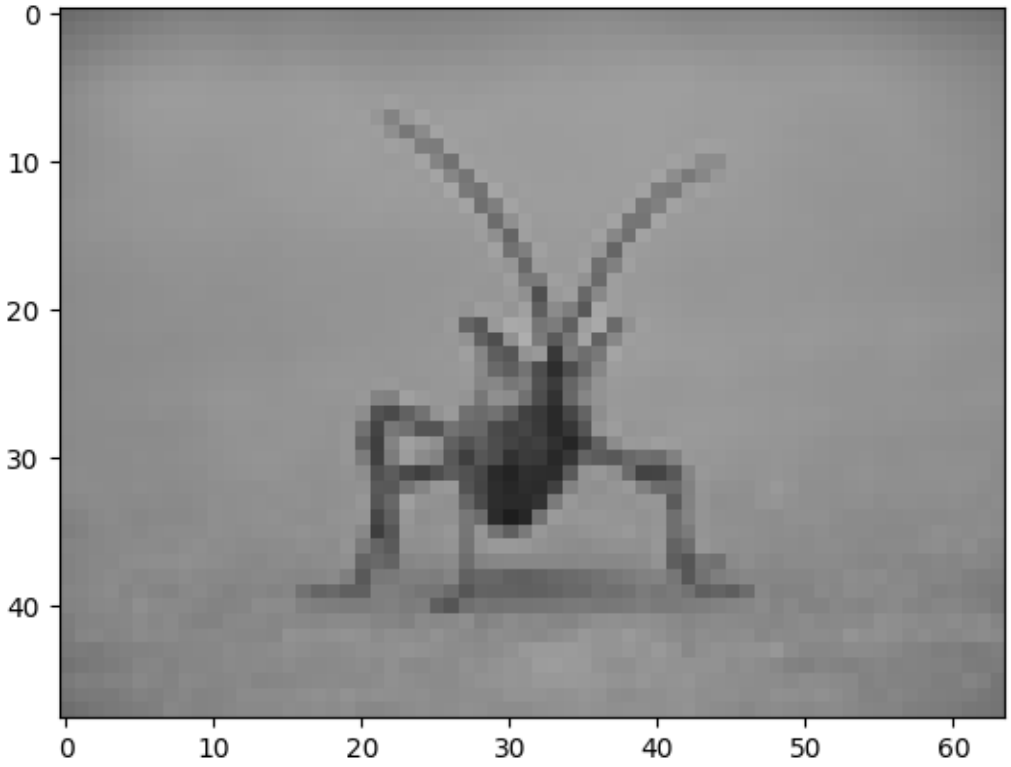
```
<matplotlib.colorbar.Colorbar object at 0x7f53fe846f10>
```

Array Interpolation schemes

Interpolation calculates what the color or value of a pixel "should" be, according to different mathematical schemes. One common place that this happens is when you resize an image. The number of pixels change, but you want the same information. Since pixels are discrete, there's missing space. Interpolation is how you fill that space. This is why your images sometimes come out looking pixelated when you blow them up. The effect is more pronounced when the difference between the original image and the expanded image is greater. Let's take our image and shrink it. We're effectively discarding pixels, only keeping a select few. Now when we plot it, that data gets blown up to the size on your screen. The old pixels aren't there anymore, and the computer has to draw in pixels to fill that space.

We'll use the Pillow library that we used to load the image also to resize the image.

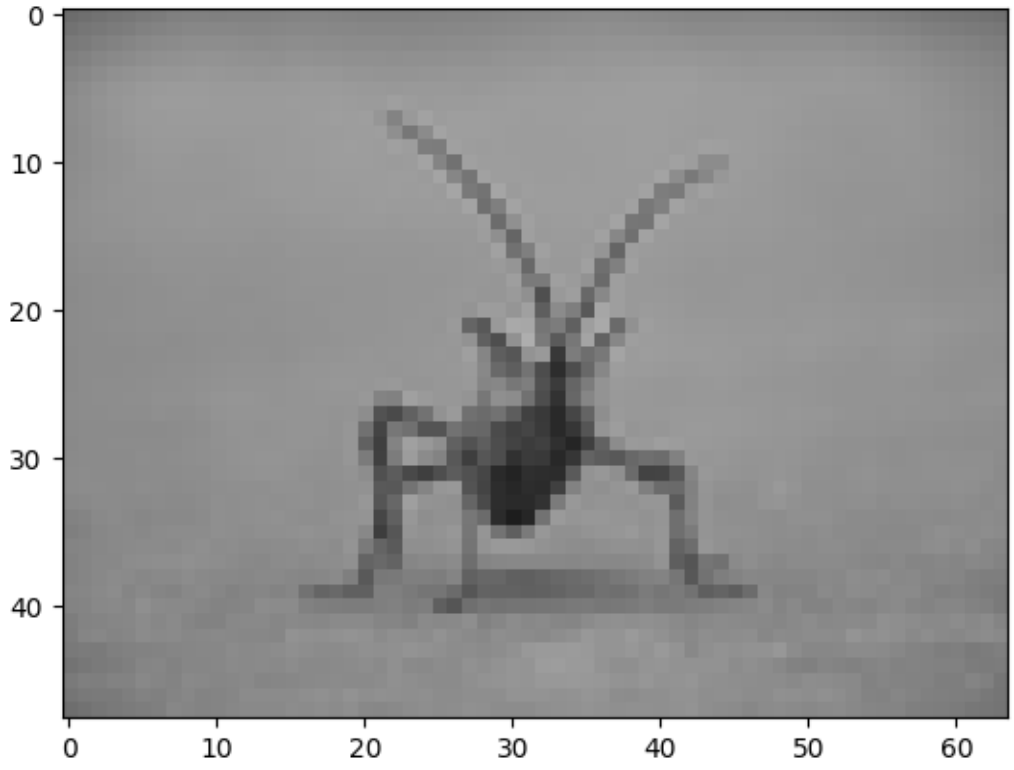
```
from PIL import Image

img = Image.open('../../doc/_static/stinkbug.png')
img.thumbnail((64, 64), Image.ANTIALIAS)  # resizes image in-place
imgplot = plt.imshow(img)
```



Here we have the default interpolation, bilinear, since we did not give *imshow()* any interpolation argument.

Let's try some others. Here's "nearest", which does no interpolation.

```
imgplot = plt.imshow(img, interpolation="nearest")
```



and bicubic:

```
imgplot = plt.imshow(img, interpolation="bicubic")
```

Bicubic interpolation is often used when blowing up photos - people tend to prefer blurry over pixelated.

Total running time of the script: (0 minutes 7.057 seconds)

2.1.5 The Lifecycle of a Plot

This tutorial aims to show the beginning, middle, and end of a single visualization using Matplotlib. We'll begin with some raw data and end by saving a figure of a customized visualization. Along the way we try to highlight some neat features and best-practices using Matplotlib.

Note: This tutorial is based on this excellent blog post by Chris Moffitt. It was transformed into this tutorial by Chris Holdgraf.

A note on the Object-Oriented API vs. Pyplot

Matplotlib has two interfaces. The first is an object-oriented (OO) interface. In this case, we utilize an instance of *axes.Axes* in order to render visualizations on an instance of *figure.Figure*.

The second is based on MATLAB and uses a state-based interface. This is encapsulated in the *pyplot* module. See the *pyplot tutorials* for a more in-depth look at the pyplot interface.

Most of the terms are straightforward but the main thing to remember is that:

- The Figure is the final image that may contain 1 or more Axes.
- The Axes represent an individual plot (don't confuse this with the word "axis", which refers to the x/y axis of a plot).

We call methods that do the plotting directly from the Axes, which gives us much more flexibility and power in customizing our plot.

Note: In general, try to use the object-oriented interface over the pyplot interface.

Our data

We'll use the data from the post from which this tutorial was derived. It contains sales information for a number of companies.

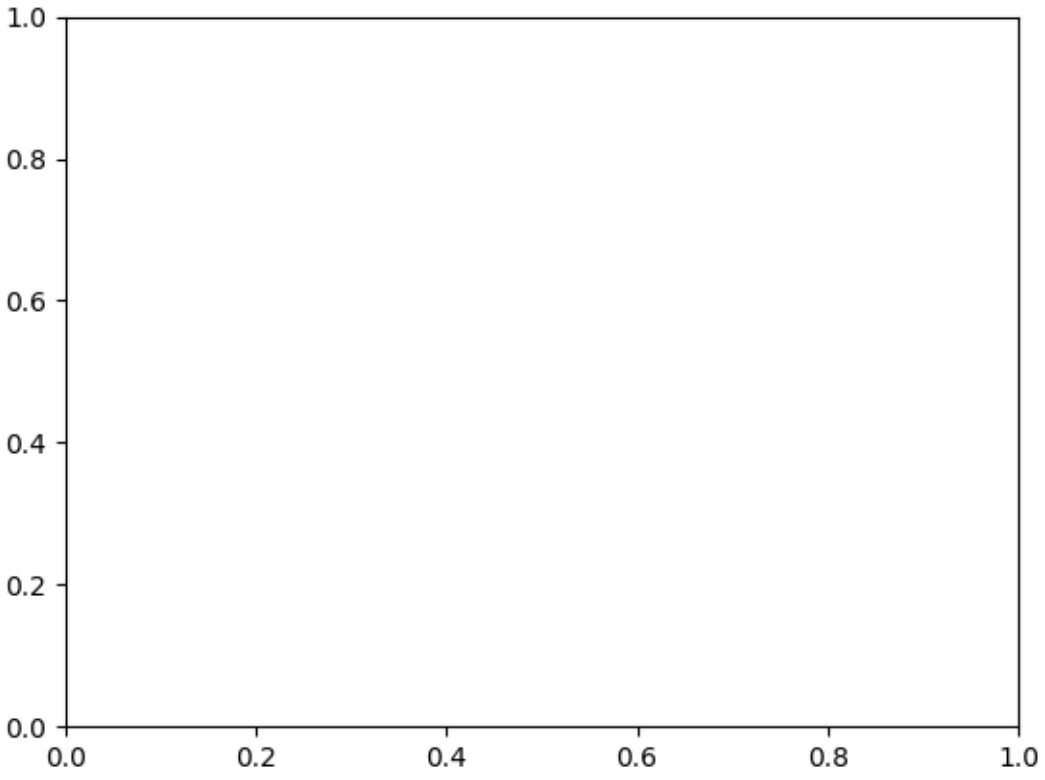
```
import numpy as np
import matplotlib.pyplot as plt


data = {'Barton LLC': 109438.50,
        'Frami, Hills and Schmidt': 103569.59,
        'Fritsch, Russel and Anderson': 112214.71,
        'Jerde-Hilpert': 112591.43,
        'Keeling LLC': 100934.30,
        'Koepp Ltd': 103660.54,
        'Kulas Inc': 137351.96,
        'Trantow-Barrows': 123381.38,
        'White-Trantow': 135841.99,
        'Will LLC': 104437.60}
group_data = list(data.values())
group_names = list(data.keys())
group_mean = np.mean(group_data)
```

Getting started

This data is naturally visualized as a barplot, with one bar per group. To do this with the object-oriented approach, we first generate an instance of *figure.Figure* and *axes.Axes*. The Figure is like a canvas, and the Axes is a part of that canvas on which we will make a particular visualization.

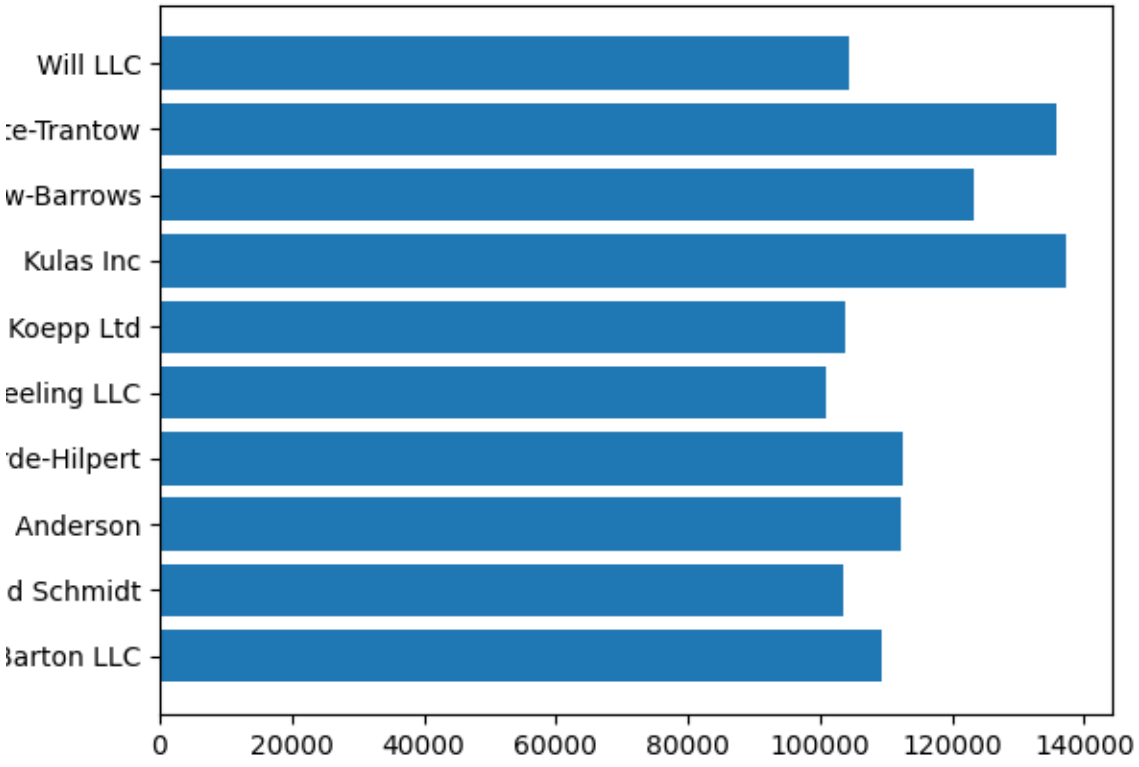
Note: Figures can have multiple axes on them. For information on how to do this, see the *Tight Layout tutorial*.

```
fig, ax = plt.subplots()
```



Now that we have an Axes instance, we can plot on top of it.

```
fig, ax = plt.subplots()
ax.barh(group_names, group_data)
```



Out:

```
<BarContainer object of 10 artists>
```

Controlling the style

There are many styles available in Matplotlib in order to let you tailor your visualization to your needs. To see a list of styles, we can use *style*.

```
print(plt.style.available)
```

Out:

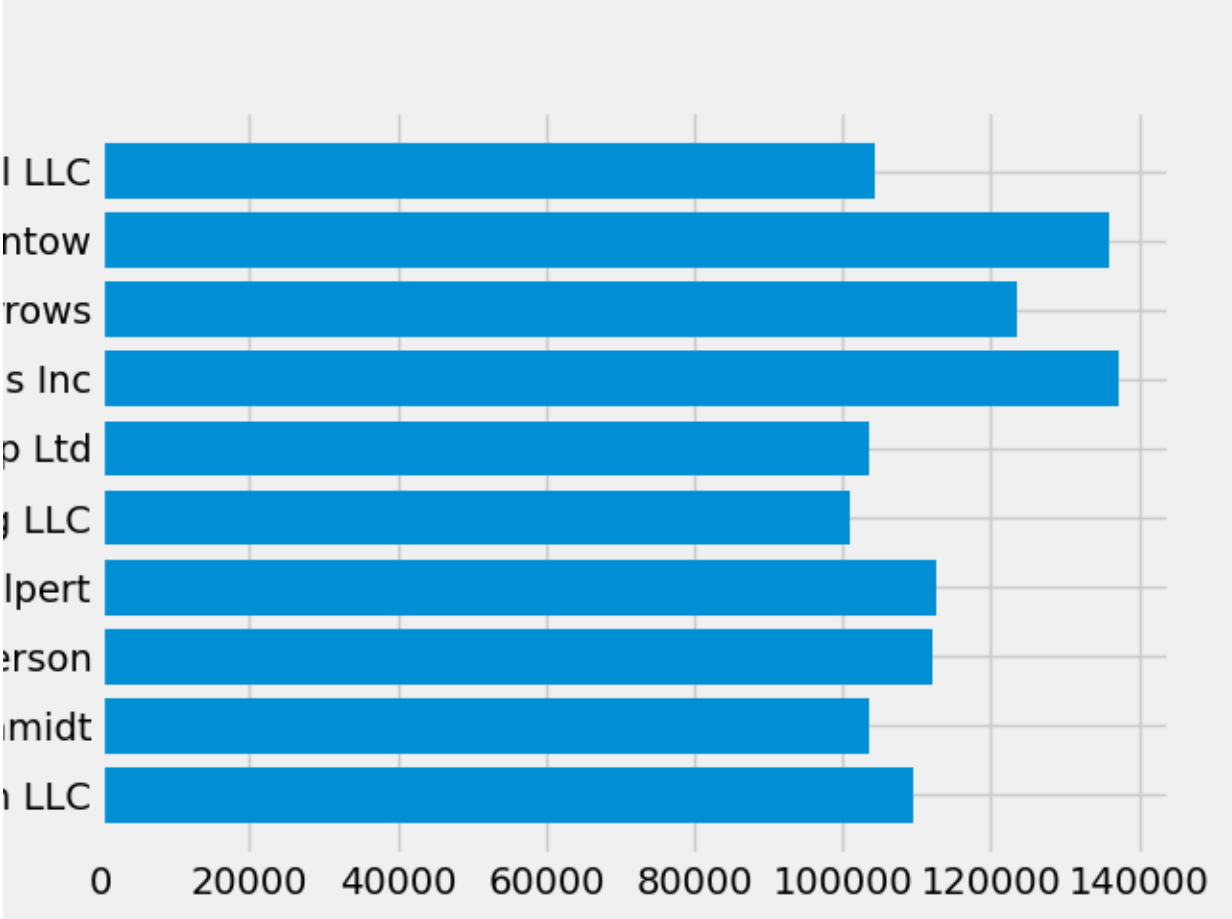
```
['Solarize_Light2', '_classic_test_patch', 'bmh', 'classic', 'dark_background', 'fast',
↪'fivethirtyeight', 'ggplot', 'grayscale', 'seaborn', 'seaborn-bright', 'seaborn-
↪colorblind', 'seaborn-dark', 'seaborn-dark-palette', 'seaborn-darkgrid', 'seaborn-deep
↪', 'seaborn-muted', 'seaborn-notebook', 'seaborn-paper', 'seaborn-pastel', 'seaborn-
↪poster', 'seaborn-talk', 'seaborn-ticks', 'seaborn-white', 'seaborn-whitegrid',
↪'tableau-colorblind10']
```

You can activate a style with the following:

```
plt.style.use('fivethirtyeight')
```

Now let's remake the above plot to see how it looks:

```
fig, ax = plt.subplots()
ax.barh(group_names, group_data)
```



Out:

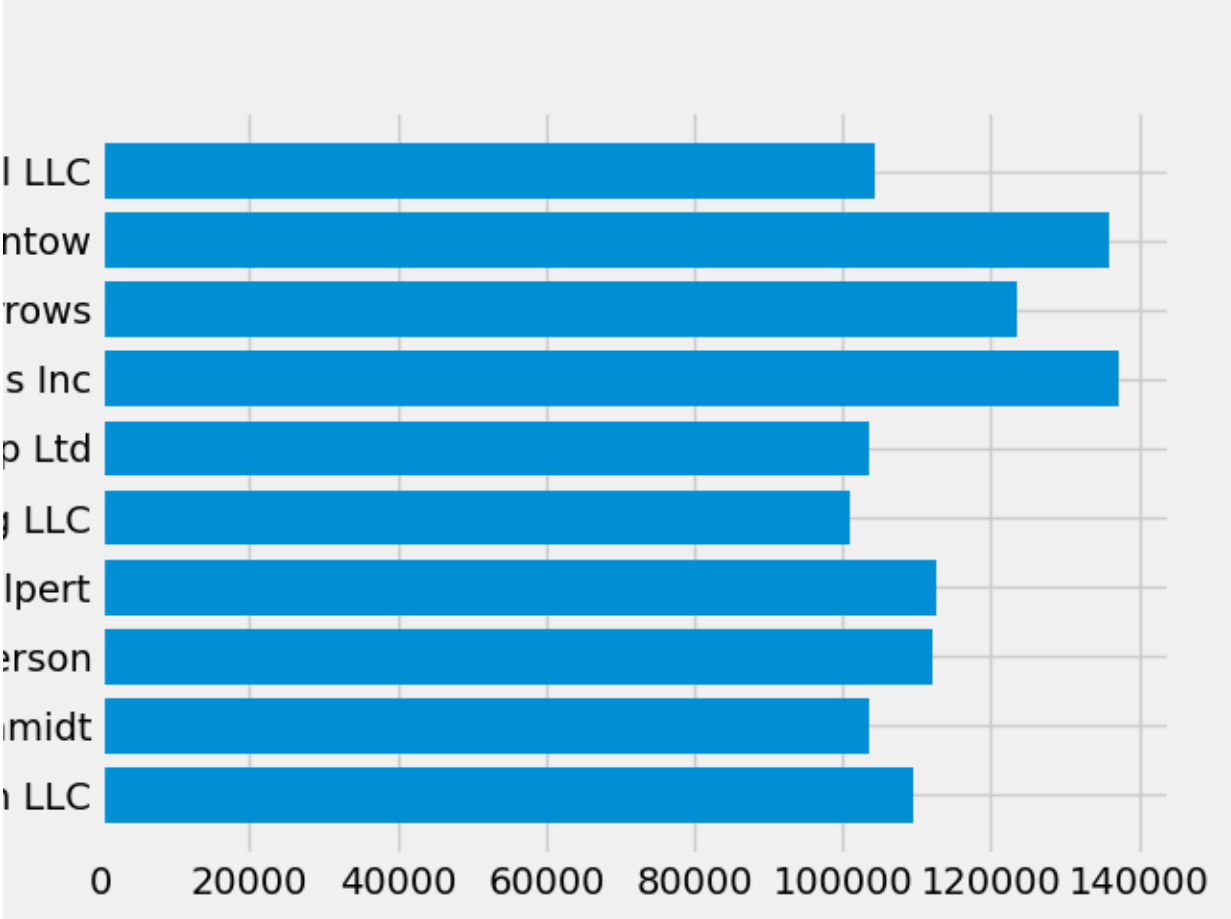
```
<BarContainer object of 10 artists>
```

The style controls many things, such as color, linewidths, backgrounds, etc.

Customizing the plot

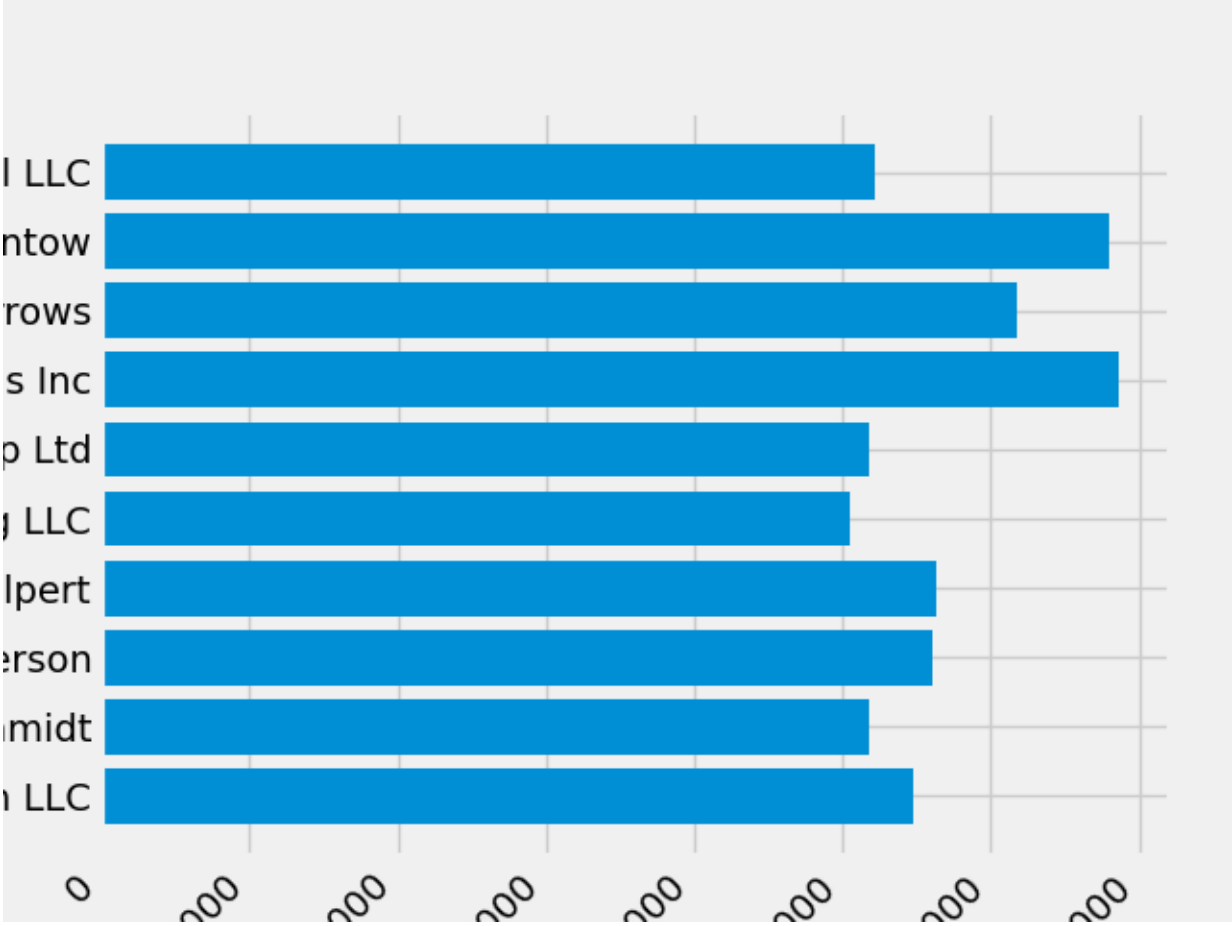
Now we've got a plot with the general look that we want, so let's fine-tune it so that it's ready for print. First let's rotate the labels on the x-axis so that they show up more clearly. We can gain access to these labels with the *axes.Axes.get_xticklabels()* method:

```
fig, ax = plt.subplots()
ax.barh(group_names, group_data)
labels = ax.get_xticklabels()
```



If we'd like to set the property of many items at once, it's useful to use the *pyplot.setp()* function. This will take a list (or many lists) of Matplotlib objects, and attempt to set some style element of each one.

```
fig, ax = plt.subplots()
ax.barh(group_names, group_data)
labels = ax.get_xticklabels()
plt.setp(labels, rotation=45, horizontalalignment='right')
```

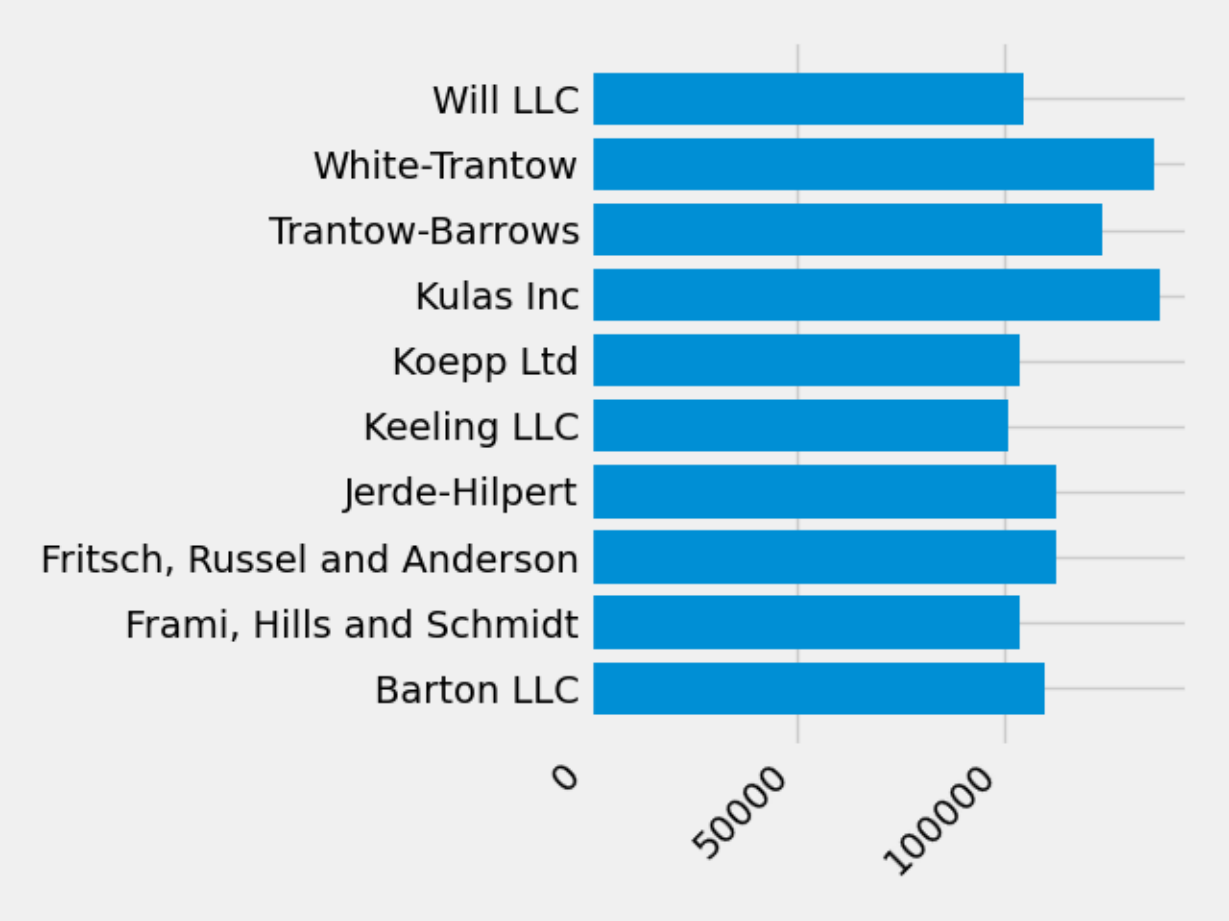


Out:

```
[None, None, None, None, None, None, None, None, None, None, None, None, None,␣
↪None, None, None, None]
```

It looks like this cut off some of the labels on the bottom. We can tell Matplotlib to automatically make room for elements in the figures that we create. To do this we set the `autolayout` value of our rcParams. For more information on controlling the style, layout, and other features of plots with rcParams, see *Customizing Matplotlib with style sheets and rcParams*.

```
plt.rcParams.update({'figure.autolayout': True})

fig, ax = plt.subplots()
ax.barh(group_names, group_data)
labels = ax.get_xticklabels()
plt.setp(labels, rotation=45, horizontalalignment='right')
```

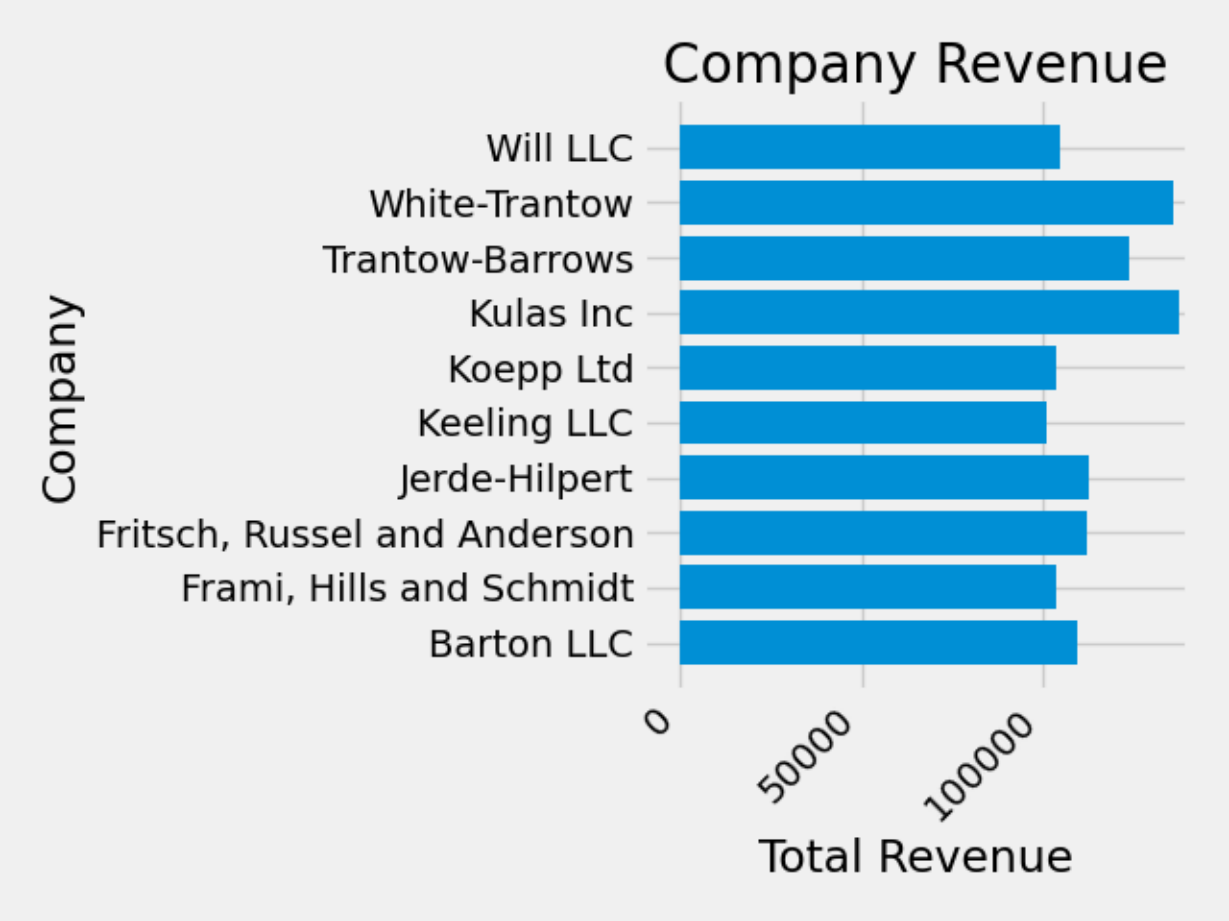


Out:

```
[None, None, None, None, None, None, None, None, None, None, None, None, None,␣
↪None, None, None, None]
```

Next, we add labels to the plot. To do this with the OO interface, we can use the *Artist.set()* method to set properties of this Axes object.

```
fig, ax = plt.subplots()
ax.barh(group_names, group_data)
labels = ax.get_xticklabels()
plt.setp(labels, rotation=45, horizontalalignment='right')
ax.set(xlim=[-10000, 140000], xlabel='Total Revenue', ylabel='Company',
       title='Company Revenue')
```



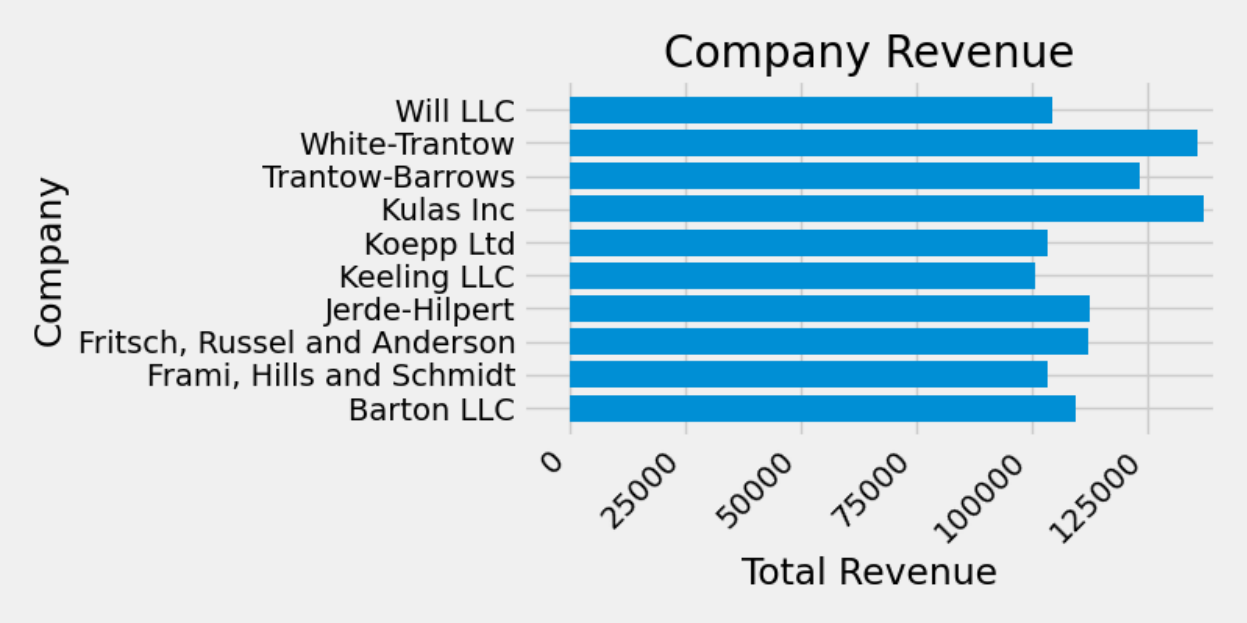
Out:

```
[(-10000.0, 140000.0), Text(0.5, 0, 'Total Revenue'), Text(0, 0.5, 'Company'), Text(0.5,␣
↪1.0, 'Company Revenue')]
```

We can also adjust the size of this plot using the *pyplot.subplots()* function. We can do this with the `figsize` kwarg.

Note: While indexing in NumPy follows the form (row, column), the figsize kwarg follows the form (width, height). This follows conventions in visualization, which unfortunately are different from those of linear algebra.

```
fig, ax = plt.subplots(figsize=(8, 4))
ax.barh(group_names, group_data)
labels = ax.get_xticklabels()
plt.setp(labels, rotation=45, horizontalalignment='right')
ax.set(xlim=[-10000, 140000], xlabel='Total Revenue', ylabel='Company',
       title='Company Revenue')
```



Out:

```
[(-10000.0, 140000.0), Text(0.5, 0, 'Total Revenue'), Text(0, 0.5, 'Company'), Text(0.5,␣
↪1.0, 'Company Revenue')]
```

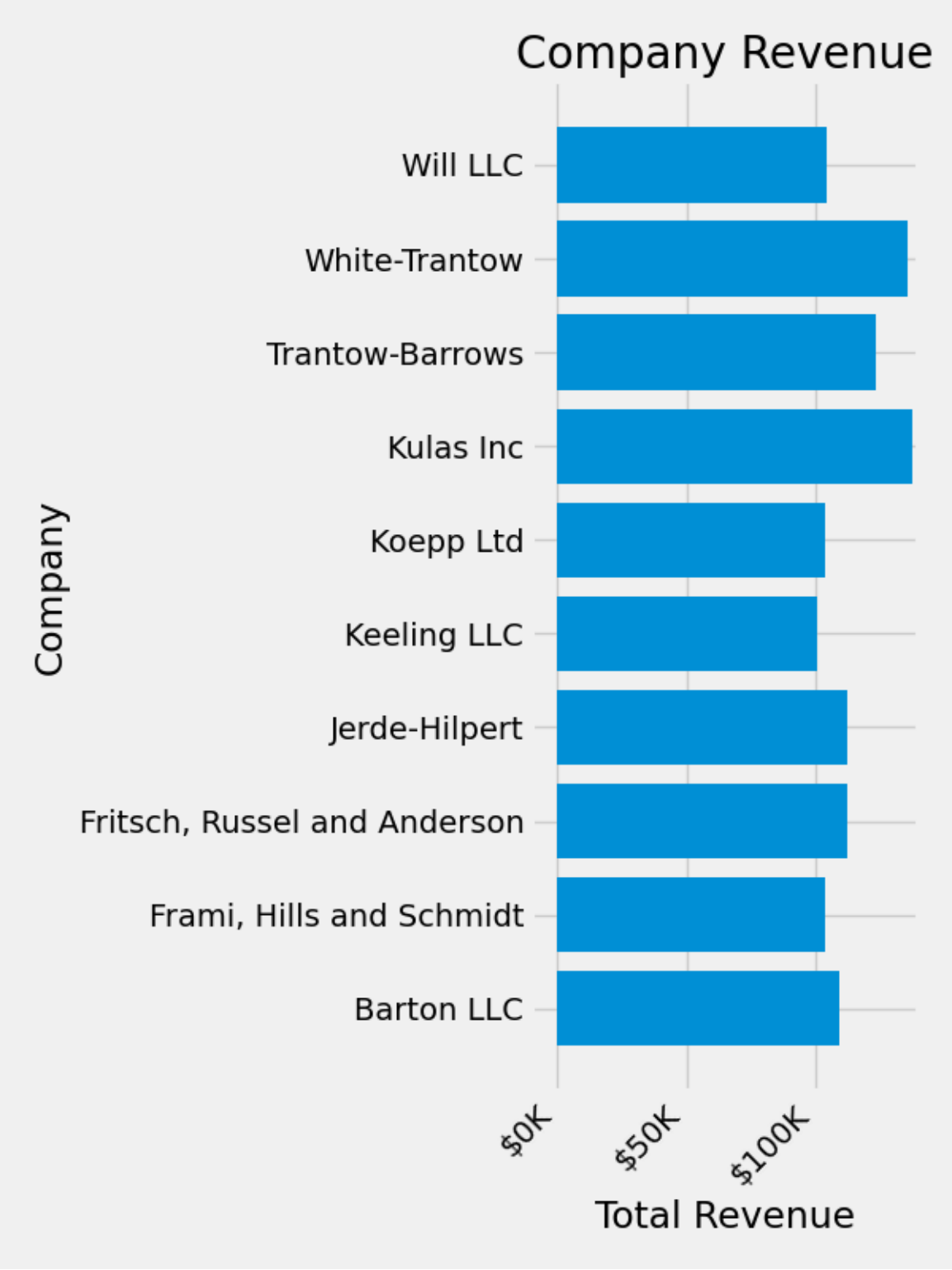
For labels, we can specify custom formatting guidelines in the form of functions. Below we define a function that takes an integer as input, and returns a string as an output. When used with *Axis.set_major_formatter* or *Axis.set_minor_formatter*, they will automatically create and use a *ticker.FuncFormatter* class.

For this function, the `x` argument is the original tick label and `pos` is the tick position. We will only use `x` here but both arguments are needed.

```
def currency(x, pos):
    """The two args are the value and tick position"""
    if x >= 1e6:
        s = '${:1.1f}M'.format(x*1e-6)
    else:
        s = '${:1.0f}K'.format(x*1e-3)
    return s
```

We can then apply this function to the labels on our plot. To do this, we use the `xaxis` attribute of our axes. This lets you perform actions on a specific axis on our plot.

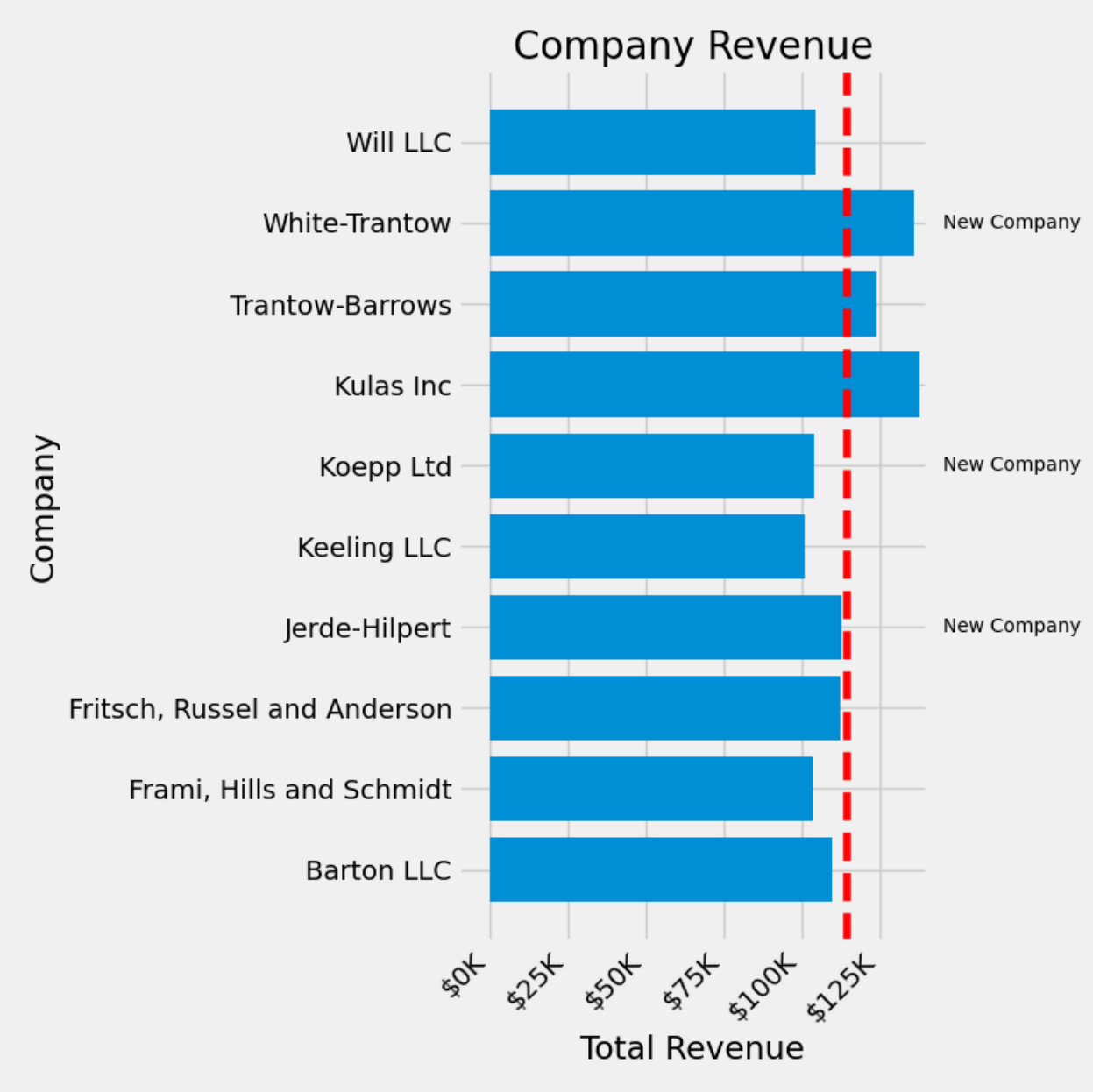
```
fig, ax = plt.subplots(figsize=(6, 8))
ax.barh(group_names, group_data)
labels = ax.get_xticklabels()
plt.setp(labels, rotation=45, horizontalalignment='right')

ax.set(xlim=[-10000, 140000], xlabel='Total Revenue', ylabel='Company',
       title='Company Revenue')
ax.xaxis.set_major_formatter(currency)
```


Company Revenue
Will LLC
White-Trantow
Trantow-Barrows
Kulas Inc
Koepp Ltd
Keeling LLC
Jerde-Hilpert
Fritsch, Russel and Anderson
Frami, Hills and Schmidt
Barton LLC
Company
$0K
$50K
$100K
Total Revenue

Combining multiple visualizations

It is possible to draw multiple plot elements on the same instance of *axes.Axes*. To do this we simply need to call another one of the plot methods on that axes object.

```
fig, ax = plt.subplots(figsize=(8, 8))
ax.barh(group_names, group_data)
labels = ax.get_xticklabels()
plt.setp(labels, rotation=45, horizontalalignment='right')

# Add a vertical line, here we set the style in the function call
ax.axvline(group_mean, ls='--', color='r')

# Annotate new companies
for group in [3, 5, 8]:
    ax.text(145000, group, "New Company", fontsize=10,
            verticalalignment="center")

# Now we move our title up since it's getting a little cramped
ax.title.set(y=1.05)

ax.set(xlim=[-10000, 140000], xlabel='Total Revenue', ylabel='Company',
       title='Company Revenue')
ax.xaxis.set_major_formatter(currency)
ax.set_xticks([0, 25e3, 50e3, 75e3, 100e3, 125e3])
fig.subplots_adjust(right=.1)

plt.show()
```



Saving our plot

Now that we're happy with the outcome of our plot, we want to save it to disk. There are many file formats we can save to in Matplotlib. To see a list of available options, use:

```
print(fig.canvas.get_supported_filetypes())
```

Out:

```
{'eps': 'Encapsulated Postscript', 'jpg': 'Joint Photographic Experts Group', 'jpeg':
↪'Joint Photographic Experts Group', 'pdf': 'Portable Document Format', 'pgf': 'PGF␣
↪code for LaTeX', 'png': 'Portable Network Graphics', 'ps': 'Postscript', 'raw': 'Raw␣
↪RGBA bitmap', 'rgba': 'Raw RGBA bitmap', 'svg': 'Scalable Vector Graphics', 'svgz':
↪'Scalable Vector Graphics', 'tif': 'Tagged Image File Format', 'tiff': 'Tagged Image␣
↪File Format'}
```

We can then use the *figure.Figure.savefig()* in order to save the figure to disk. Note that there are several useful flags we show below:

- `transparent=True` makes the background of the saved figure transparent if the format supports it.
- `dpi=80` controls the resolution (dots per square inch) of the output.
- `bbox_inches="tight"` fits the bounds of the figure to our plot.

```
# Uncomment this line to save the figure.
# fig.savefig('sales.png', transparent=False, dpi=80, bbox_inches="tight")
```

Total running time of the script: (0 minutes 2.779 seconds)

2.1.6 Customizing Matplotlib with style sheets and rcParams

Tips for customizing the properties and default styles of Matplotlib.

Using style sheets

The *style* package adds support for easy-to-switch plotting "styles" with the same parameters as a *matplotlib rc* file (which is read at startup to configure Matplotlib).

There are a number of pre-defined styles provided by Matplotlib. For example, there's a pre-defined style called "ggplot", which emulates the aesthetics of ggplot (a popular plotting package for R). To use this style, just add:

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
from cycler import cycler
plt.style.use('ggplot')
data = np.random.randn(50)
```

To list all available styles, use:

```
print(plt.style.available)
```

Out:

```
['Solarize_Light2', '_classic_test_patch', 'bmh', 'classic', 'dark_background', 'fast',
↪'fivethirtyeight', 'ggplot', 'grayscale', 'seaborn', 'seaborn-bright', 'seaborn-
↪colorblind', 'seaborn-dark', 'seaborn-dark-palette', 'seaborn-darkgrid', 'seaborn-deep
↪', 'seaborn-muted', 'seaborn-notebook', 'seaborn-paper', 'seaborn-pastel', 'seaborn-
↪poster', 'seaborn-talk', 'seaborn-ticks', 'seaborn-white', 'seaborn-whitegrid',
↪'tableau-colorblind10']
```

(continues on next page)

(continued from previous page)

Defining your own style

You can create custom styles and use them by calling *style.use* with the path or URL to the style sheet.

For example, you might want to create `./images/presentation.mplstyle` with the following:

```
axes.titlesize : 24
axes.labelsize : 20
lines.linewidth : 3
lines.markersize : 10
xtick.labelsize : 16
ytick.labelsize : 16
```

Then, when you want to adapt a plot designed for a paper to one that looks good in a presentation, you can just add:

```
>>> import matplotlib.pyplot as plt
>>> plt.style.use('./images/presentation.mplstyle')
```

Alternatively, you can make your style known to Matplotlib by placing your `<style-name>.mplstyle` file into `mpl_configdir/stylelib`. You can then load your custom style sheet with a call to `style.use(<style-name>)`. By default `mpl_configdir` should be `~/.config/matplotlib`, but you can check where yours is with *matplotlib.get_configdir()*; you may need to create this directory. You also can change the directory where Matplotlib looks for the stylelib/ folder by setting the *MPLCONFIGDIR* environment variable, see *matplotlib configuration and cache directory locations*.

Note that a custom style sheet in `mpl_configdir/stylelib` will override a style sheet defined by Matplotlib if the styles have the same name.

Once your `<style-name>.mplstyle` file is in the appropriate `mpl_configdir` you can specify your style with:

```
>>> import matplotlib.pyplot as plt
>>> plt.style.use(<style-name>)
```

Composing styles

Style sheets are designed to be composed together. So you can have a style sheet that customizes colors and a separate style sheet that alters element sizes for presentations. These styles can easily be combined by passing a list of styles:

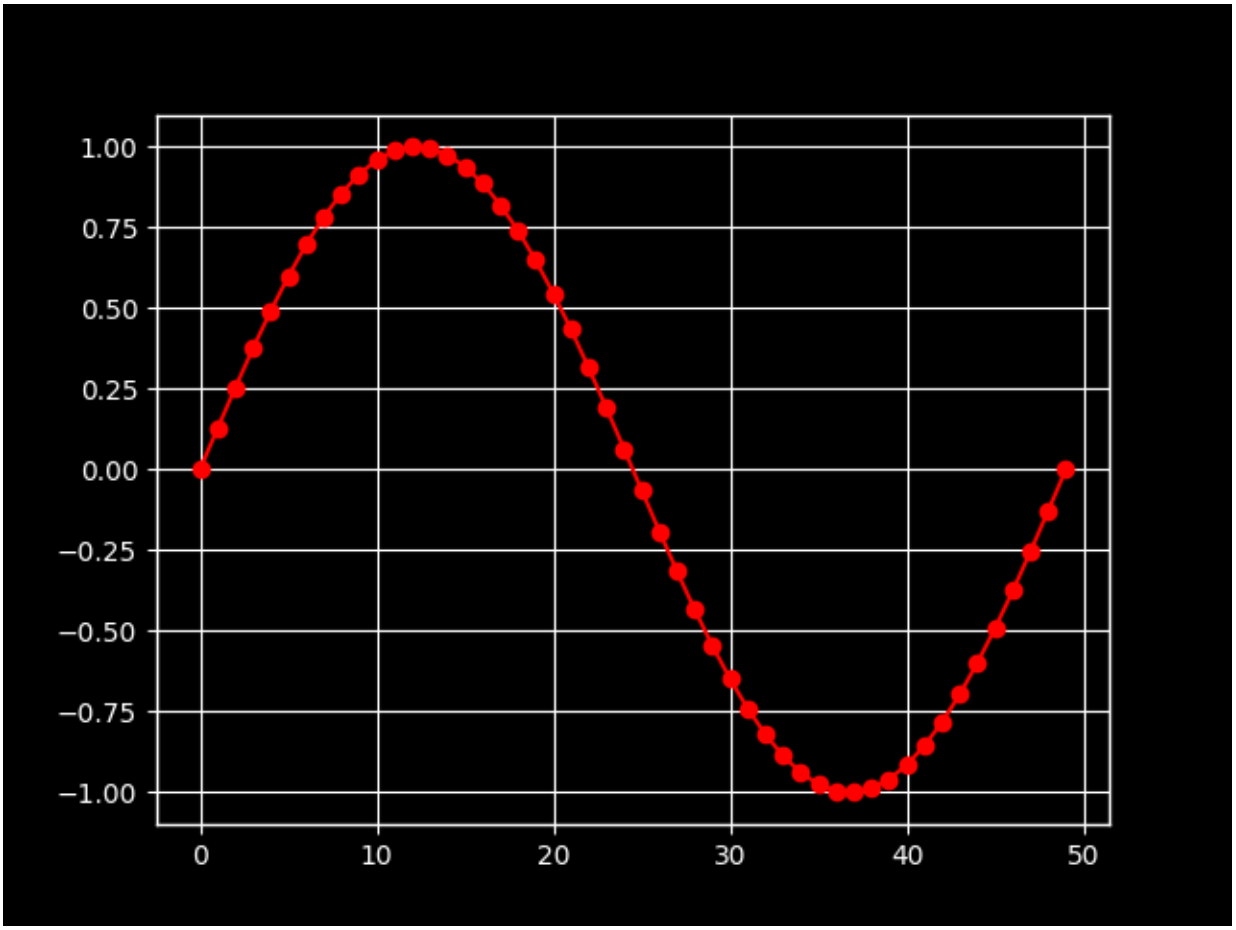
```
>>> import matplotlib.pyplot as plt
>>> plt.style.use(['dark_background', 'presentation'])
```

Note that styles further to the right will overwrite values that are already defined by styles on the left.

Temporary styling

If you only want to use a style for a specific block of code but don't want to change the global styling, the style package provides a context manager for limiting your changes to a specific scope. To isolate your styling changes, you can write something like the following:

```
with plt.style.context('dark_background'):
    plt.plot(np.sin(np.linspace(0, 2 * np.pi)), 'r-o')
plt.show()
```

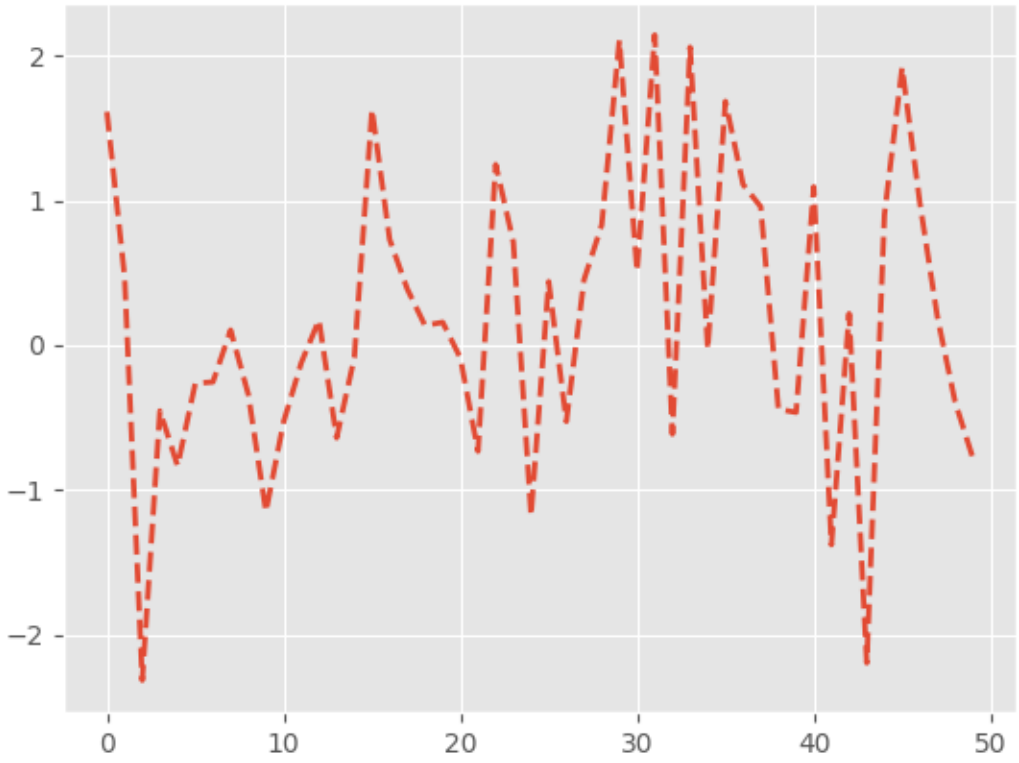


2.1.7 Matplotlib rcParams

Dynamic rc settings

You can also dynamically change the default rc settings in a python script or interactively from the python shell. All of the rc settings are stored in a dictionary-like variable called *matplotlib.rcParams*, which is global to the matplotlib package. rcParams can be modified directly, for example:

```
mpl.rcParams['lines.linewidth'] = 2
mpl.rcParams['lines.linestyle'] = '--'
plt.plot(data)
```

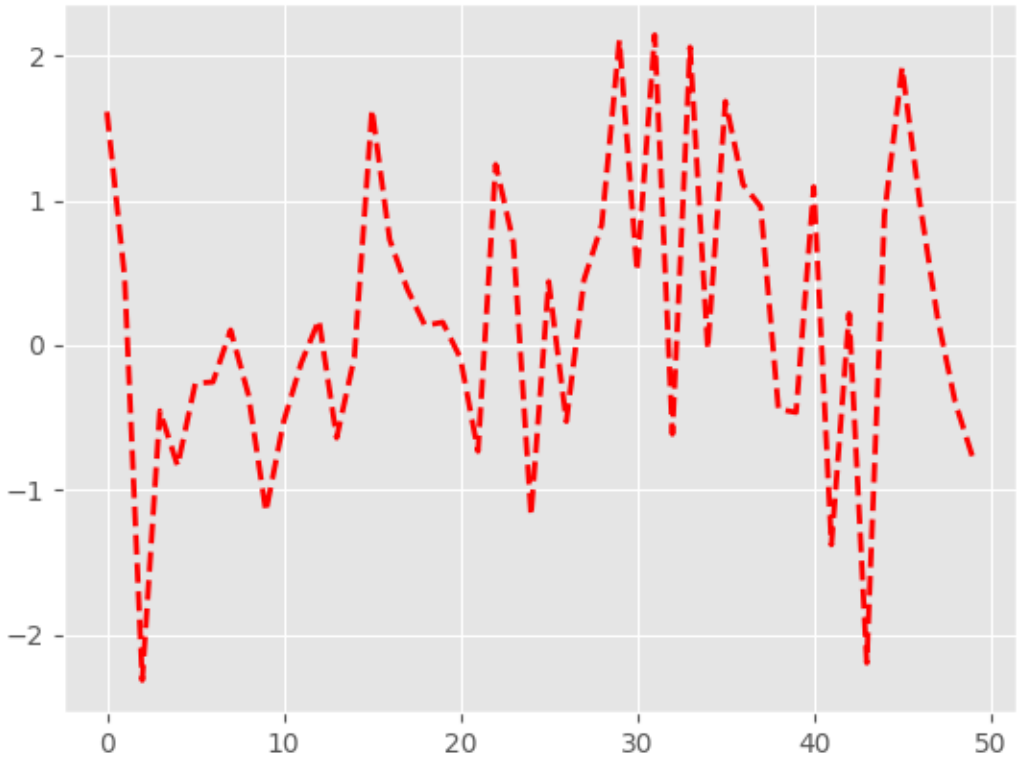


Out:

```
[<matplotlib.lines.Line2D object at 0x7f53ff786550>]
```

Note, that in order to change the usual *plot* color you have to change the *prop_cycle* property of *axes*:

```
mpl.rcParams['axes.prop_cycle'] = cycler(color=['r', 'g', 'b', 'y'])
plt.plot(data)  # first color is red
```

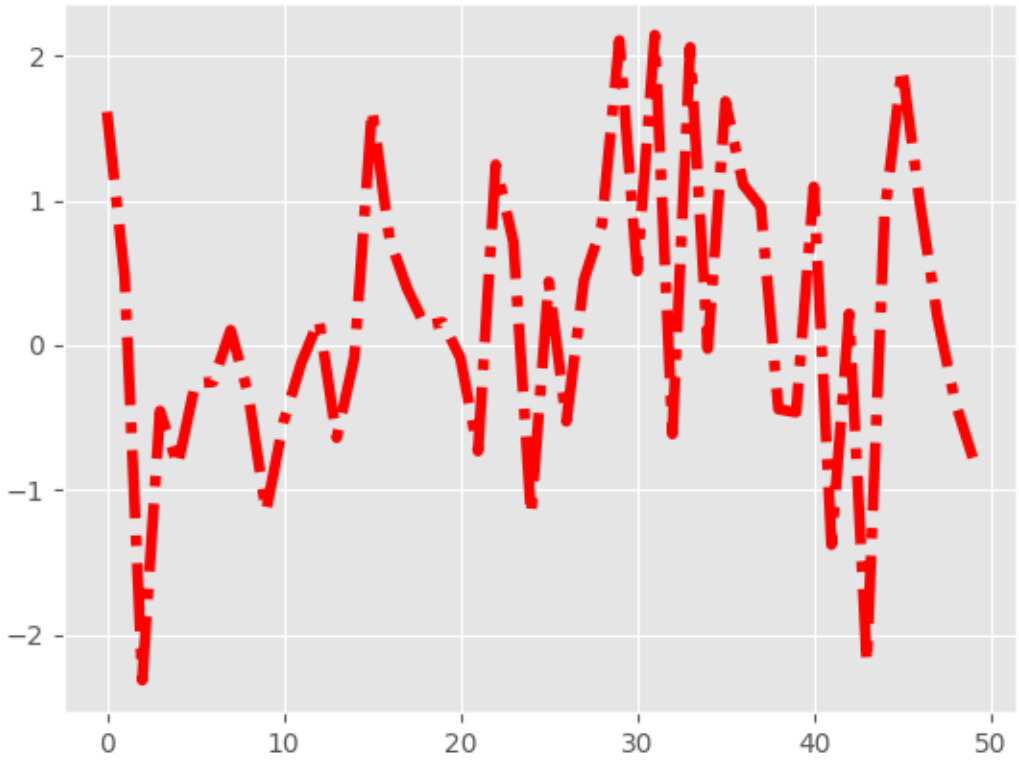


Out:

```
[<matplotlib.lines.Line2D object at 0x7f54003ff550>]
```

Matplotlib also provides a couple of convenience functions for modifying rc settings. *matplotlib.rc* can be used to modify multiple settings in a single group at once, using keyword arguments:

```
mpl.rc('lines', linewidth=4, linestyle='-.')
plt.plot(data)
```



Out:

```
[<matplotlib.lines.Line2D object at 0x7f53fb3ccf40>]
```

matplotlib.rcdefaults will restore the standard Matplotlib default settings.

There is some degree of validation when setting the values of rcParams, see *matplotlib.rcsetup* for details.

The `matplotlibrc` file

Matplotlib uses `matplotlibrc` configuration files to customize all kinds of properties, which we call 'rc settings' or 'rc parameters'. You can control the defaults of almost every property in Matplotlib: figure size and DPI, line width, color and style, axes, axis and grid properties, text and font properties and so on. When a URL or path is not specified with a call to `style.use('<path>/<style-name>.mplstyle')`, Matplotlib looks for `matplotlibrc` in four locations, in the following order:

1. `matplotlibrc` in the current working directory, usually used for specific customizations that you do not want to apply elsewhere.
2. `$MATPLOTLIBRC` if it is a file, else `$MATPLOTLIBRC/matplotlibrc`.

3. It next looks in a user-specific place, depending on your platform:

   - On Linux and FreeBSD, it looks in `.config/matplotlib/matplotlibrc` (or `$XDG_CONFIG_HOME/matplotlib/matplotlibrc`) if you've customized your environment.
   - On other platforms, it looks in `.matplotlib/matplotlibrc`.

   See *matplotlib configuration and cache directory locations*.

4. *INSTALL*`/matplotlib/mpl-data/matplotlibrc`, where *INSTALL* is something like `/usr/lib/python3.7/site-packages` on Linux, and maybe `C:\Python37\Lib\site-packages` on Windows. Every time you install matplotlib, this file will be overwritten, so if you want your customizations to be saved, please move this file to your user-specific matplotlib directory.

Once a `matplotlibrc` file has been found, it will *not* search any of the other paths.

To display where the currently active `matplotlibrc` file was loaded from, one can do the following:

```
>>> import matplotlib
>>> matplotlib.matplotlib_fname()
'/home/foo/.config/matplotlib/matplotlibrc'
```

See below for a sample *matplotlibrc file*.

A sample matplotlibrc file

```
#### MATPLOTLIBRC FORMAT

## NOTE FOR END USERS: DO NOT EDIT THIS FILE!
##
## This is a sample matplotlib configuration file - you can find a copy
## of it on your system in site-packages/matplotlib/mpl-data/matplotlibrc
## (which related to your Python installation location).
##
## You should find a copy of it on your system at
## site-packages/matplotlib/mpl-data/matplotlibrc (relative to your Python
## installation location).  DO NOT EDIT IT!
##
## If you wish to change your default style, copy this file to one of the
## following locations
##     unix/linux:
##         $HOME/.config/matplotlib/matplotlibrc OR
##         $XDG_CONFIG_HOME/matplotlib/matplotlibrc (if $XDG_CONFIG_HOME is set)
##     other platforms:
##         $HOME/.matplotlib/matplotlibrc
## and edit that copy.
##
## See https://matplotlib.org/users/customizing.html#the-matplotlibrc-file
## for more details on the paths which are checked for the configuration file.
```

(continues on next page)

(continued from previous page)

```
##
## Blank lines, or lines starting with a comment symbol, are ignored, as are
## trailing comments.  Other lines must have the format:
##     key: val  # optional comment
##
## Formatting: Use PEP8-like style (as enforced in the rest of the codebase).
## All lines start with an additional '#', so that removing all leading '#'s
## yields a valid style file.
##
## Colors: for the color values below, you can either use
##     - a matplotlib color string, such as r, k, or b
##     - an rgb tuple, such as (1.0, 0.5, 0.0)
##     - a hex string, such as ff00ff
##     - a scalar grayscale intensity such as 0.75
##     - a legal html color name, e.g., red, blue, darkslategray
##
## Matplotlib configuration are currently divided into following parts:
##     - BACKENDS
##     - LINES
##     - PATCHES
##     - HATCHES
##     - BOXPLOT
##     - FONT
##     - TEXT
##     - LaTeX
##     - AXES
##     - DATES
##     - TICKS
##     - GRIDS
##     - LEGEND
##     - FIGURE
##     - IMAGES
##     - CONTOUR PLOTS
##     - ERRORBAR PLOTS
##     - HISTOGRAM PLOTS
##     - SCATTER PLOTS
##     - AGG RENDERING
##     - PATHS
##     - SAVING FIGURES
##     - INTERACTIVE KEYMAPS
##     - ANIMATION

##### CONFIGURATION BEGINS HERE


## ***************************************************************************
## * BACKENDS                                                                *
## ***************************************************************************
## The default backend.  If you omit this parameter, the first working
## backend from the following list is used:
##     MacOSX Qt5Agg Gtk3Agg TkAgg WxAgg Agg
```

(continues on next page)

(continued from previous page)

```
## Other choices include:
##     Qt5Cairo GTK3Cairo TkCairo WxCairo Cairo
##     Qt4Agg Qt4Cairo Wx  # deprecated.
##     PS PDF SVG Template
## You can also deploy your own backend outside of matplotlib by referring to
## the module name (which must be in the PYTHONPATH) as 'module://my_backend'.
#backend: Agg

## The port to use for the web server in the WebAgg backend.
#webagg.port: 8988

## The address on which the WebAgg web server should be reachable
#webagg.address: 127.0.0.1

## If webagg.port is unavailable, a number of other random ports will
## be tried until one that is available is found.
#webagg.port_retries: 50

## When True, open the webbrowser to the plot that is shown
#webagg.open_in_browser: True

## If you are running pyplot inside a GUI and your backend choice
## conflicts, we will automatically try to find a compatible one for
## you if backend_fallback is True
#backend_fallback: True

#interactive: False
#toolbar:     toolbar2  # {None, toolbar2, toolmanager}
#timezone:    UTC       # a pytz timezone string, e.g., US/Central or Europe/Paris


## ***************************************************************************
## * LINES                                                                   *
## ***************************************************************************
## See https://matplotlib.org/api/artist_api.html#module-matplotlib.lines
## for more information on line properties.
#lines.linewidth: 1.5               # line width in points
#lines.linestyle: -                 # solid line
#lines.color:     C0                # has no affect on plot(); see axes.prop_cycle
#lines.marker:          None        # the default marker
#lines.markerfacecolor: auto        # the default marker face color
#lines.markeredgecolor: auto        # the default marker edge color
#lines.markeredgewidth: 1.0         # the line width around the marker symbol
#lines.markersize:      6           # marker size, in points
#lines.dash_joinstyle:  round       # {miter, round, bevel}
#lines.dash_capstyle:   butt        # {butt, round, projecting}
#lines.solid_joinstyle: round       # {miter, round, bevel}
#lines.solid_capstyle:  projecting  # {butt, round, projecting}
#lines.antialiased: True            # render lines in antialiased (no jaggies)

## The three standard dash patterns.  These are scaled by the linewidth.
```

(continues on next page)

(continued from previous page)

```
#lines.dashed_pattern: 3.7, 1.6
#lines.dashdot_pattern: 6.4, 1.6, 1, 1.6
#lines.dotted_pattern: 1, 1.65
#lines.scale_dashes: True

#markers.fillstyle: full  # {full, left, right, bottom, top, none}

#pcolor.shading : flat

## ***************************************************************************
## * PATCHES                                                                 *
## ***************************************************************************
## Patches are graphical objects that fill 2D space, like polygons or circles.
## See https://matplotlib.org/api/artist_api.html#module-matplotlib.patches
## for more information on patch properties.
#patch.linewidth:       1      # edge width in points.
#patch.facecolor:       C0
#patch.edgecolor:       black  # if forced, or patch is not filled
#patch.force_edgecolor: False  # True to always use edgecolor
#patch.antialiased:     True   # render patches in antialiased (no jaggies)


## ***************************************************************************
## * HATCHES                                                                 *
## ***************************************************************************
#hatch.color:     black
#hatch.linewidth: 1.0


## ***************************************************************************
## * BOXPLOT                                                                 *
## ***************************************************************************
#boxplot.notch:       False
#boxplot.vertical:    True
#boxplot.whiskers:    1.5
#boxplot.bootstrap:   None
#boxplot.patchartist: False
#boxplot.showmeans:   False
#boxplot.showcaps:    True
#boxplot.showbox:     True
#boxplot.showfliers:  True
#boxplot.meanline:    False

#boxplot.flierprops.color:           black
#boxplot.flierprops.marker:          o
#boxplot.flierprops.markerfacecolor: none
#boxplot.flierprops.markeredgecolor: black
#boxplot.flierprops.markeredgewidth: 1.0
#boxplot.flierprops.markersize:      6
#boxplot.flierprops.linestyle:       none
#boxplot.flierprops.linewidth:       1.0
```

(continues on next page)

(continued from previous page)

```
#boxplot.boxprops.color:     black
#boxplot.boxprops.linewidth: 1.0
#boxplot.boxprops.linestyle: -

#boxplot.whiskerprops.color:     black
#boxplot.whiskerprops.linewidth: 1.0
#boxplot.whiskerprops.linestyle: -

#boxplot.capprops.color:     black
#boxplot.capprops.linewidth: 1.0
#boxplot.capprops.linestyle: -

#boxplot.medianprops.color:     C1
#boxplot.medianprops.linewidth: 1.0
#boxplot.medianprops.linestyle: -

#boxplot.meanprops.color:           C2
#boxplot.meanprops.marker:          ^
#boxplot.meanprops.markerfacecolor: C2
#boxplot.meanprops.markeredgecolor: C2
#boxplot.meanprops.markersize:       6
#boxplot.meanprops.linestyle:       --
#boxplot.meanprops.linewidth:       1.0


## ***************************************************************************
## * FONT                                                                    *
## ***************************************************************************
## The font properties used by `text.Text`.
## See https://matplotlib.org/api/font_manager_api.html for more information
## on font properties.  The 6 font properties used for font matching are
## given below with their default values.
##
## The font.family property has five values:
##     - 'serif' (e.g., Times),
##     - 'sans-serif' (e.g., Helvetica),
##     - 'cursive' (e.g., Zapf-Chancery),
##     - 'fantasy' (e.g., Western), and
##     - 'monospace' (e.g., Courier).
## Each of these font families has a default list of font names in decreasing
## order of priority associated with them.  When text.usetex is False,
## font.family may also be one or more concrete font names.
##
## The font.style property has three values: normal (or roman), italic
## or oblique.  The oblique style will be used for italic, if it is not
## present.
##
## The font.variant property has two values: normal or small-caps.  For
## TrueType fonts, which are scalable fonts, small-caps is equivalent
## to using a font size of 'smaller', or about 83%% of the current font
```

(continues on next page)

(continued from previous page)

```
## size.
##
## The font.weight property has effectively 13 values: normal, bold,
## bolder, lighter, 100, 200, 300, ..., 900.  Normal is the same as
## 400, and bold is 700.  bolder and lighter are relative values with
## respect to the current weight.
##
## The font.stretch property has 11 values: ultra-condensed,
## extra-condensed, condensed, semi-condensed, normal, semi-expanded,
## expanded, extra-expanded, ultra-expanded, wider, and narrower.  This
## property is not currently implemented.
##
## The font.size property is the default font size for text, given in pts.
## 10 pt is the standard value.
##
## Note that font.size controls default text sizes.  To configure
## special text sizes tick labels, axes, labels, title, etc, see the rc
## settings for axes and ticks.  Special text sizes can be defined
## relative to font.size, using the following values: xx-small, x-small,
## small, medium, large, x-large, xx-large, larger, or smaller

#font.family:  sans-serif
#font.style:   normal
#font.variant: normal
#font.weight:  normal
#font.stretch: normal
#font.size:    10.0

#font.serif:      DejaVu Serif, Bitstream Vera Serif, Computer Modern Roman, New Century␣
↪Schoolbook, Century Schoolbook L, Utopia, ITC Bookman, Bookman, Nimbus Roman No9 L,␣
↪Times New Roman, Times, Palatino, Charter, serif
#font.sans-serif: DejaVu Sans, Bitstream Vera Sans, Computer Modern Sans Serif, Lucida␣
↪Grande, Verdana, Geneva, Lucid, Arial, Helvetica, Avant Garde, sans-serif
#font.cursive:    Apple Chancery, Textile, Zapf Chancery, Sand, Script MT, Felipa,␣
↪cursive
#font.fantasy:    Comic Neue, Comic Sans MS, Chicago, Charcoal, ImpactWestern, Humor␣
↪Sans, xkcd, fantasy
#font.monospace:  DejaVu Sans Mono, Bitstream Vera Sans Mono, Computer Modern Typewriter,
↪ Andale Mono, Nimbus Mono L, Courier New, Courier, Fixed, Terminal, monospace


## ***************************************************************************
## * TEXT                                                                    *
## ***************************************************************************
## The text properties used by `text.Text`.
## See https://matplotlib.org/api/artist_api.html#module-matplotlib.text
## for more information on text properties
#text.color: black


## ***************************************************************************
```

(continues on next page)

(continued from previous page)

```
## * LaTeX                                                                *
## ************************************************************************
## For more information on LaTex properties, see
## https://matplotlib.org/tutorials/text/usetex.html
#text.usetex: False  # use latex for all text handling. The following fonts
                     # are supported through the usual rc parameter settings:
                     # new century schoolbook, bookman, times, palatino,
                     # zapf chancery, charter, serif, sans-serif, helvetica,
                     # avant garde, courier, monospace, computer modern roman,
                     # computer modern sans serif, computer modern typewriter
                     # If another font is desired which can loaded using the
                     # LaTeX \usepackage command, please inquire at the
                     # matplotlib mailing list
#text.latex.preamble:   # IMPROPER USE OF THIS FEATURE WILL LEAD TO LATEX FAILURES
                       # AND IS THEREFORE UNSUPPORTED. PLEASE DO NOT ASK FOR HELP
                       # IF THIS FEATURE DOES NOT DO WHAT YOU EXPECT IT TO.
                       # text.latex.preamble is a single line of LaTeX code that
                       # will be passed on to the LaTeX system. It may contain
                       # any code that is valid for the LaTeX "preamble", i.e.
                       # between the "\documentclass" and "\begin{document}"
                       # statements.
                       # Note that it has to be put on a single line, which may
                       # become quite long.
                       # The following packages are always loaded with usetex, so
                       # beware of package collisions: color, geometry, graphicx,
                       # type1cm, textcomp.
                       # Adobe Postscript (PSSNFS) font packages may also be
                       # loaded, depending on your font settings.

## FreeType hinting flag ("foo" corresponds to FT_LOAD_FOO); may be one of the
## following (Proprietary Matplotlib-specific synonyms are given in parentheses,
## but their use is discouraged):
## - default: Use the font's native hinter if possible, else FreeType's auto-hinter.
##            ("either" is a synonym).
## - no_autohint: Use the font's native hinter if possible, else don't hint.
##                ("native" is a synonym.)
## - force_autohint: Use FreeType's auto-hinter.  ("auto" is a synonym.)
## - no_hinting: Disable hinting.  ("none" is a synonym.)
#text.hinting: force_autohint

#text.hinting_factor: 8  # Specifies the amount of softness for hinting in the
                        # horizontal direction.  A value of 1 will hint to full
                        # pixels.  A value of 2 will hint to half pixels etc.
#text.kerning_factor : 0  # Specifies the scaling factor for kerning values. This
                         # is provided solely to allow old test images to remain
                         # unchanged. Set to 6 to obtain previous behavior. Values
                         # other than 0 or 6 have no defined meaning.
#text.antialiased: True  # If True (default), the text will be antialiased.
                        # This only affects the Agg backend.

## The following settings allow you to select the fonts in math mode.
```

(continues on next page)

(continued from previous page)

```
#mathtext.fontset: dejavusans  # Should be 'dejavusans' (default),
                               # 'dejavuserif', 'cm' (Computer Modern), 'stix',
                               # 'stixsans' or 'custom' (unsupported, may go
                               # away in the future)
## "mathtext.fontset: custom" is defined by the mathtext.bf, .cal, .it, ...
## settings which map a TeX font name to a fontconfig font pattern.  (These
## settings are not used for other font sets.)
#mathtext.bf:  sans:bold
#mathtext.cal: cursive
#mathtext.it:  sans:italic
#mathtext.rm:  sans
#mathtext.sf:  sans
#mathtext.tt:  monospace
#mathtext.fallback: cm  # Select fallback font from ['cm' (Computer Modern), 'stix'
                        # 'stixsans'] when a symbol can not be found in one of the
                        # custom math fonts. Select 'None' to not perform fallback
                        # and replace the missing character by a dummy symbol.
#mathtext.default: it  # The default font to use for math.
                       # Can be any of the LaTeX font names, including
                       # the special name "regular" for the same font
                       # used in regular text.


## ***************************************************************************
## * AXES                                                                    *
## ***************************************************************************
## Following are default face and edge colors, default tick sizes,
## default fontsizes for ticklabels, and so on.  See
## https://matplotlib.org/api/axes_api.html#module-matplotlib.axes
#axes.facecolor:     white   # axes background color
#axes.edgecolor:     black   # axes edge color
#axes.linewidth:     0.8     # edge linewidth
#axes.grid:          False   # display grid or not
#axes.grid.axis:     both    # which axis the grid should apply to
#axes.grid.which:    major   # gridlines at {major, minor, both} ticks
#axes.titlelocation: center  # alignment of the title: {left, right, center}
#axes.titlesize:     large   # fontsize of the axes title
#axes.titleweight:   normal  # font weight of title
#axes.titlecolor:    auto    # color of the axes title, auto falls back to
                             # text.color as default value
#axes.titley:        None    # position title (axes relative units).  None implies auto
#axes.titlepad:      6.0     # pad between axes and title in points
#axes.labelsize:     medium  # fontsize of the x any y labels
#axes.labelpad:      4.0     # space between label and axis
#axes.labelweight:   normal  # weight of the x and y labels
#axes.labelcolor:    black
#axes.axisbelow:     line    # draw axis gridlines and ticks:
                             #     - below patches (True)
                             #     - above patches but below lines ('line')
                             #     - above all (False)
```

(continues on next page)

(continued from previous page)

```
#axes.formatter.limits: -5, 6  # use scientific notation if log10
                               # of the axis range is smaller than the
                               # first or larger than the second
#axes.formatter.use_locale: False  # When True, format tick labels
                                   # according to the user's locale.
                                   # For example, use ',' as a decimal
                                   # separator in the fr_FR locale.
#axes.formatter.use_mathtext: False  # When True, use mathtext for scientific
                                     # notation.
#axes.formatter.min_exponent: 0  # minimum exponent to format in scientific notation
#axes.formatter.useoffset: True  # If True, the tick label formatter
                                 # will default to labeling ticks relative
                                 # to an offset when the data range is
                                 # small compared to the minimum absolute
                                 # value of the data.
#axes.formatter.offset_threshold: 4  # When useoffset is True, the offset
                                     # will be used when it can remove
                                     # at least this number of significant
                                     # digits from tick labels.

#axes.spines.left:   True  # display axis spines
#axes.spines.bottom: True
#axes.spines.top:    True
#axes.spines.right:  True

#axes.unicode_minus: True  # use Unicode for the minus symbol rather than hyphen.  See
                           # https://en.wikipedia.org/wiki/Plus_and_minus_signs
↪#Character_codes
#axes.prop_cycle: cycler('color', ['1f77b4', 'ff7f0e', '2ca02c', 'd62728', '9467bd',
↪'8c564b', 'e377c2', '7f7f7f', 'bcbd22', '17becf'])
                  # color cycle for plot lines as list of string colorspecs:
                  # single letter, long name, or web-style hex
                  # As opposed to all other paramters in this file, the color
                  # values must be enclosed in quotes for this parameter,
                  # e.g. '1f77b4', instead of 1f77b4.
                  # See also https://matplotlib.org/tutorials/intermediate/color_cycle.
↪html
                  # for more details on prop_cycle usage.
#axes.autolimit_mode: data  # How to scale axes limits to the data.  By using:
                           #     - "data" to use data limits, plus some margin
                           #     - "round_numbers" move to the nearest "round" number
#axes.xmargin:   .05  # x margin.  See `axes.Axes.margins`
#axes.ymargin:   .05  # y margin.  See `axes.Axes.margins`
#polaraxes.grid: True  # display grid on polar axes
#axes3d.grid:    True  # display grid on 3d axes


## ***************************************************************************
## * AXIS                                                                    *
## ***************************************************************************
#xaxis.labellocation: center  # alignment of the xaxis label: {left, right, center}
```

(continues on next page)

(continued from previous page)

```
#yaxis.labellocation: center  # alignment of the yaxis label: {bottom, top, center}


## ***************************************************************************
## * DATES                                                                   *
## ***************************************************************************
## These control the default format strings used in AutoDateFormatter.
## Any valid format datetime format string can be used (see the python
## `datetime` for details).  For example, by using:
##     - '%%x' will use the locale date representation
##     - '%%X' will use the locale time representation
##     - '%%c' will use the full locale datetime representation
## These values map to the scales:
##     {'year': 365, 'month': 30, 'day': 1, 'hour': 1/24, 'minute': 1 / (24 * 60)}

#date.autoformatter.year:        %Y
#date.autoformatter.month:       %Y-%m
#date.autoformatter.day:         %Y-%m-%d
#date.autoformatter.hour:        %m-%d %H
#date.autoformatter.minute:      %d %H:%M
#date.autoformatter.second:      %H:%M:%S
#date.autoformatter.microsecond: %M:%S.%f

## The reference date for Matplotlib's internal date representation
## See https://matplotlib.org/examples/ticks_and_spines/date_precision_and_epochs.py
#date.epoch: 1970-01-01T00:00:00

## ***************************************************************************
## * TICKS                                                                   *
## ***************************************************************************
## See https://matplotlib.org/api/axis_api.html#matplotlib.axis.Tick
#xtick.top:           False   # draw ticks on the top side
#xtick.bottom:        True    # draw ticks on the bottom side
#xtick.labeltop:      False   # draw label on the top
#xtick.labelbottom:   True    # draw label on the bottom
#xtick.major.size:    3.5     # major tick size in points
#xtick.minor.size:    2       # minor tick size in points
#xtick.major.width:   0.8     # major tick width in points
#xtick.minor.width:   0.6     # minor tick width in points
#xtick.major.pad:     3.5     # distance to major tick label in points
#xtick.minor.pad:     3.4     # distance to the minor tick label in points
#xtick.color:         black   # color of the tick labels
#xtick.labelsize:     medium  # fontsize of the tick labels
#xtick.direction:     out     # direction: {in, out, inout}
#xtick.minor.visible: False   # visibility of minor ticks on x-axis
#xtick.major.top:     True    # draw x axis top major ticks
#xtick.major.bottom:  True    # draw x axis bottom major ticks
#xtick.minor.top:     True    # draw x axis top minor ticks
#xtick.minor.bottom:  True    # draw x axis bottom minor ticks
#xtick.alignment:     center  # alignment of xticks
```

(continues on next page)

(continued from previous page)

```
#ytick.left:          True    # draw ticks on the left side
#ytick.right:         False   # draw ticks on the right side
#ytick.labelleft:     True    # draw tick labels on the left side
#ytick.labelright:    False   # draw tick labels on the right side
#ytick.major.size:    3.5     # major tick size in points
#ytick.minor.size:    2       # minor tick size in points
#ytick.major.width:   0.8     # major tick width in points
#ytick.minor.width:   0.6     # minor tick width in points
#ytick.major.pad:     3.5     # distance to major tick label in points
#ytick.minor.pad:     3.4     # distance to the minor tick label in points
#ytick.color:         black   # color of the tick labels
#ytick.labelsize:     medium  # fontsize of the tick labels
#ytick.direction:     out     # direction: {in, out, inout}
#ytick.minor.visible: False   # visibility of minor ticks on y-axis
#ytick.major.left:    True    # draw y axis left major ticks
#ytick.major.right:   True    # draw y axis right major ticks
#ytick.minor.left:    True    # draw y axis left minor ticks
#ytick.minor.right:   True    # draw y axis right minor ticks
#ytick.alignment:     center_baseline  # alignment of yticks


## ***************************************************************************
## * GRIDS                                                                   *
## ***************************************************************************
#grid.color:     b0b0b0  # grid color
#grid.linestyle: -       # solid
#grid.linewidth: 0.8     # in points
#grid.alpha:     1.0     # transparency, between 0.0 and 1.0


## ***************************************************************************
## * LEGEND                                                                  *
## ***************************************************************************
#legend.loc:           best
#legend.frameon:       True     # if True, draw the legend on a background patch
#legend.framealpha:    0.8      # legend patch transparency
#legend.facecolor:     inherit  # inherit from axes.facecolor; or color spec
#legend.edgecolor:     0.8      # background patch boundary color
#legend.fancybox:      True     # if True, use a rounded box for the
                                # legend background, else a rectangle
#legend.shadow:        False    # if True, give background a shadow effect
#legend.numpoints:     1        # the number of marker points in the legend line
#legend.scatterpoints: 1        # number of scatter points
#legend.markerscale:   1.0      # the relative size of legend markers vs. original
#legend.fontsize:      medium
#legend.title_fontsize: None    # None sets to the same as the default axes.

## Dimensions as fraction of fontsize:
#legend.borderpad:     0.4  # border whitespace
#legend.labelspacing:  0.5  # the vertical space between the legend entries
#legend.handlelength:  2.0  # the length of the legend lines
```

(continues on next page)

(continued from previous page)

```
#legend.handleheight:  0.7  # the height of the legend handle
#legend.handletextpad: 0.8  # the space between the legend line and legend text
#legend.borderaxespad: 0.5  # the border between the axes and legend edge
#legend.columnspacing: 2.0  # column separation


## ***************************************************************************
## * FIGURE                                                                  *
## ***************************************************************************
## See https://matplotlib.org/api/figure_api.html#matplotlib.figure.Figure
#figure.titlesize:   large     # size of the figure title (``Figure.suptitle()``)
#figure.titleweight: normal    # weight of the figure title
#figure.figsize:     6.4, 4.8  # figure size in inches
#figure.dpi:         100       # figure dots per inch
#figure.facecolor:   white     # figure facecolor
#figure.edgecolor:   white     # figure edgecolor
#figure.frameon:     True      # enable figure frame
#figure.max_open_warning: 20   # The maximum number of figures to open through
                               # the pyplot interface before emitting a warning.
                               # If less than one this feature is disabled.
#figure.raise_window : True    # Raise the GUI window to front when show() is called.

## The figure subplot parameters.  All dimensions are a fraction of the figure width and␣
↪height.
#figure.subplot.left:   0.125  # the left side of the subplots of the figure
#figure.subplot.right:  0.9    # the right side of the subplots of the figure
#figure.subplot.bottom: 0.11   # the bottom of the subplots of the figure
#figure.subplot.top:    0.88   # the top of the subplots of the figure
#figure.subplot.wspace: 0.2    # the amount of width reserved for space between subplots,
                               # expressed as a fraction of the average axis width
#figure.subplot.hspace: 0.2    # the amount of height reserved for space between␣
↪subplots,
                               # expressed as a fraction of the average axis height

## Figure layout
#figure.autolayout: False  # When True, automatically adjust subplot
                           # parameters to make the plot fit the figure
                           # using `tight_layout`
#figure.constrained_layout.use: False  # When True, automatically make plot
                                       # elements fit on the figure. (Not
                                       # compatible with `autolayout`, above).
#figure.constrained_layout.h_pad:  0.04167  # Padding around axes objects. Float␣
↪representing
#figure.constrained_layout.w_pad:  0.04167  # inches. Default is 3./72. inches (3 pts)
#figure.constrained_layout.hspace: 0.02     # Space between subplot groups. Float␣
↪representing
#figure.constrained_layout.wspace: 0.02     # a fraction of the subplot widths being␣
↪separated.


## ***************************************************************************
```

(continues on next page)

(continued from previous page)

```
## * IMAGES                                                                    *
## ****************************************************************************
#image.aspect: equal                # {equal, auto} or a number
#image.interpolation:  antialiased  # see help(imshow) for options
#image.cmap:   viridis              # A colormap name, gray etc...
#image.lut:    256                 # the size of the colormap lookup table
#image.origin: upper               # {lower, upper}
#image.resample:  True
#image.composite_image: True  # When True, all the images on a set of axes are
                              # combined into a single composite image before
                              # saving a figure as a vector graphics file,
                              # such as a PDF.


## ****************************************************************************
## * CONTOUR PLOTS                                                             *
## ****************************************************************************
#contour.negative_linestyle: dashed  # string or on-off ink sequence
#contour.corner_mask:        True    # {True, False, legacy}
#contour.linewidth:          None    # {float, None} Size of the contour
                                     # linewidths. If set to None,
                                     # it falls back to `line.linewidth`.


## ****************************************************************************
## * ERRORBAR PLOTS                                                            *
## ****************************************************************************
#errorbar.capsize: 0  # length of end cap on error bars in pixels


## ****************************************************************************
## * HISTOGRAM PLOTS                                                           *
## ****************************************************************************
#hist.bins: 10  # The default number of histogram bins or 'auto'.


## ****************************************************************************
## * SCATTER PLOTS                                                             *
## ****************************************************************************
#scatter.marker: o         # The default marker type for scatter plots.
#scatter.edgecolors: face  # The default edge colors for scatter plots.


## ****************************************************************************
## * AGG RENDERING                                                             *
## ****************************************************************************
## Warning: experimental, 2008/10/10
#agg.path.chunksize: 0  # 0 to disable; values in the range
                       # 10000 to 100000 can improve speed slightly
                       # and prevent an Agg rendering failure
                       # when plotting very large data sets,
```

(continues on next page)

(continued from previous page)

```
                      # especially if they are very gappy.
                      # It may cause minor artifacts, though.
                      # A value of 20000 is probably a good
                      # starting point.


## ***************************************************************************
## * PATHS                                                                   *
## ***************************************************************************
#path.simplify: True  # When True, simplify paths by removing "invisible"
                      # points to reduce file size and increase rendering
                      # speed
#path.simplify_threshold: 0.111111111111  # The threshold of similarity below
                                          # which vertices will be removed in
                                          # the simplification process.
#path.snap: True  # When True, rectilinear axis-aligned paths will be snapped
                  # to the nearest pixel when certain criteria are met.
                  # When False, paths will never be snapped.
#path.sketch: None  # May be None, or a 3-tuple of the form:
                    # (scale, length, randomness).
                    #     - *scale* is the amplitude of the wiggle
                    #         perpendicular to the line (in pixels).
                    #     - *length* is the length of the wiggle along the
                    #         line (in pixels).
                    #     - *randomness* is the factor by which the length is
                    #         randomly scaled.
#path.effects:


## ***************************************************************************
## * SAVING FIGURES                                                          *
## ***************************************************************************
## The default savefig params can be different from the display params
## e.g., you may want a higher resolution, or to make the figure
## background white
#savefig.dpi:       figure      # figure dots per inch or 'figure'
#savefig.facecolor: auto        # figure facecolor when saving
#savefig.edgecolor: auto        # figure edgecolor when saving
#savefig.format:    png         # {png, ps, pdf, svg}
#savefig.bbox:      standard    # {tight, standard}
                               # 'tight' is incompatible with pipe-based animation
                               # backends (e.g. 'ffmpeg') but will work with those
                               # based on temporary files (e.g. 'ffmpeg_file')
#savefig.pad_inches:   0.1     # Padding to be used when bbox is set to 'tight'
#savefig.directory:    ~       # default directory in savefig dialog box,
                               # leave empty to always use current working directory
#savefig.transparent: False    # setting that controls whether figures are saved with a
                               # transparent background by default
#savefig.orientation: portrait  # Orientation of saved figure

### tk backend params
```

(continues on next page)

(continued from previous page)

```
#tk.window_focus:   False  # Maintain shell focus for TkAgg

### ps backend params
#ps.papersize:      letter  # {auto, letter, legal, ledger, A0-A10, B0-B10}
#ps.useafm:         False   # use of afm fonts, results in small files
#ps.usedistiller:   False   # {ghostscript, xpdf, None}
                            # Experimental: may produce smaller files.
                            # xpdf intended for production of publication quality files,
                            # but requires ghostscript, xpdf and ps2eps
#ps.distiller.res:  6000    # dpi
#ps.fonttype:       3       # Output Type 3 (Type3) or Type 42 (TrueType)

### PDF backend params
#pdf.compression:    6  # integer from 0 to 9
                        # 0 disables compression (good for debugging)
#pdf.fonttype:       3  # Output Type 3 (Type3) or Type 42 (TrueType)
#pdf.use14corefonts : False
#pdf.inheritcolor:   False

### SVG backend params
#svg.image_inline: True  # Write raster image data directly into the SVG file
#svg.fonttype: path      # How to handle SVG fonts:
                         #     path: Embed characters as paths -- supported
                         #           by most SVG renderers
                         #     None: Assume fonts are installed on the
                         #           machine where the SVG will be viewed.
#svg.hashsalt: None      # If not None, use this string as hash salt instead of uuid4

### pgf parameter
## See https://matplotlib.org/tutorials/text/pgf.html for more information.
#pgf.rcfonts: True
#pgf.preamble:  # See text.latex.preamble for documentation
#pgf.texsystem: xelatex

### docstring params
#docstring.hardcopy: False  # set this when you want to generate hardcopy docstring


## ***************************************************************************
## * INTERACTIVE KEYMAPS                                                     *
## ***************************************************************************
## Event keys to interact with figures/plots via keyboard.
## See https://matplotlib.org/users/navigation_toolbar.html for more details on
## interactive navigation.  Customize these settings according to your needs.
## Leave the field(s) empty if you don't need a key-map. (i.e., fullscreen : '')
#keymap.fullscreen: f, ctrl+f   # toggling
#keymap.home: h, r, home        # home or reset mnemonic
#keymap.back: left, c, backspace, MouseButton.BACK  # forward / backward keys
#keymap.forward: right, v, MouseButton.FORWARD      # for quick navigation
#keymap.pan: p                  # pan mnemonic
#keymap.zoom: o                 # zoom mnemonic
```

(continues on next page)

(continued from previous page)

```
#keymap.save: s, ctrl+s         # saving current figure
#keymap.help: f1                # display help about active tools
#keymap.quit: ctrl+w, cmd+w, q  # close the current figure
#keymap.quit_all:               # close all figures
#keymap.grid: g                 # switching on/off major grids in current axes
#keymap.grid_minor: G           # switching on/off minor grids in current axes
#keymap.yscale: l               # toggle scaling of y-axes ('log'/'linear')
#keymap.xscale: k, L            # toggle scaling of x-axes ('log'/'linear')
#keymap.copy: ctrl+c, cmd+c     # Copy figure to clipboard


## ***************************************************************************
## * ANIMATION                                                               *
## ***************************************************************************
#animation.html: none  # How to display the animation as HTML in
                       # the IPython notebook:
                       #     - 'html5' uses HTML5 video tag
                       #     - 'jshtml' creates a Javascript animation
#animation.writer:  ffmpeg        # MovieWriter 'backend' to use
#animation.codec:   h264          # Codec to use for writing movie
#animation.bitrate: -1            # Controls size/quality tradeoff for movie.
                                  # -1 implies let utility auto-determine
#animation.frame_format: png      # Controls frame format used by temp files
#animation.ffmpeg_path:  ffmpeg   # Path to ffmpeg binary. Without full path
                                  # $PATH is searched
#animation.ffmpeg_args:           # Additional arguments to pass to ffmpeg
#animation.convert_path: convert  # Path to ImageMagick's convert binary.
                                  # On Windows use the full path since convert
                                  # is also the name of a system tool.
#animation.convert_args:          # Additional arguments to pass to convert
#animation.embed_limit:  20.0     # Limit, in MB, of size of base64 encoded
                                  # animation in HTML (i.e. IPython notebook)

#mpl_toolkits.legacy_colorbar: True
```

Total running time of the script: (0 minutes 1.592 seconds)

2.2 Intermediate

These tutorials cover some of the more complicated classes and functions in Matplotlib. They can be useful for particular custom and complex visualizations.

2.2.1 Artist tutorial

Using Artist objects to render on the canvas.

There are three layers to the matplotlib API.

- the `matplotlib.backend_bases.FigureCanvas` is the area onto which the figure is drawn
- the `matplotlib.backend_bases.Renderer` is the object which knows how to draw on the `FigureCanvas`
- and the *matplotlib.artist.Artist* is the object that knows how to use a renderer to paint onto the canvas.

The `FigureCanvas` and `Renderer` handle all the details of talking to user interface toolkits like wxPython or drawing languages like PostScript®, and the `Artist` handles all the high level constructs like representing and laying out the figure, text, and lines. The typical user will spend 95% of their time working with the `Artists`.

There are two types of `Artists`: primitives and containers. The primitives represent the standard graphical objects we want to paint onto our canvas: *Line2D*, *Rectangle*, *Text*, *AxesImage*, etc., and the containers are places to put them (*Axis*, *Axes* and *Figure*). The standard use is to create a *Figure* instance, use the `Figure` to create one or more *Axes* or `Subplot` instances, and use the `Axes` instance helper methods to create the primitives. In the example below, we create a `Figure` instance using *matplotlib.pyplot.figure()*, which is a convenience method for instantiating `Figure` instances and connecting them with your user interface or drawing toolkit `FigureCanvas`. As we will discuss below, this is not necessary -- you can work directly with PostScript, PDF Gtk+, or wxPython `FigureCanvas` instances, instantiate your `Figures` directly and connect them yourselves -- but since we are focusing here on the `Artist` API we'll let *pyplot* handle some of those details for us:

```
import matplotlib.pyplot as plt
fig = plt.figure()
ax = fig.add_subplot(2, 1, 1) # two rows, one column, first plot
```

The *Axes* is probably the most important class in the matplotlib API, and the one you will be working with most of the time. This is because the `Axes` is the plotting area into which most of the objects go, and the `Axes` has many special helper methods (*plot()*, *text()*, *hist()*, *imshow()*) to create the most common graphics primitives (*Line2D*, *Text*, *Rectangle*, *AxesImage*, respectively). These helper methods will take your data (e.g., `numpy` arrays and strings) and create primitive `Artist` instances as needed (e.g., `Line2D`), add them to the relevant containers, and draw them when requested. Most of you are probably familiar with the `Subplot`, which is just a special case of an `Axes` that lives on a regular rows by columns grid of `Subplot` instances. If you want to create an `Axes` at an arbitrary location, simply use the *add_axes()* method which takes a list of `[left, bottom, width, height]` values in 0-1 relative figure coordinates:

```
fig2 = plt.figure()
ax2 = fig2.add_axes([0.15, 0.1, 0.7, 0.3])
```

Continuing with our example:

```
import numpy as np
t = np.arange(0.0, 1.0, 0.01)
s = np.sin(2*np.pi*t)
line, = ax.plot(t, s, color='blue', lw=2)
```

In this example, `ax` is the `Axes` instance created by the `fig.add_subplot` call above (remember `Subplot` is just a subclass of `Axes`) and when you call `ax.plot`, it creates a `Line2D` instance and adds it to the `Axes.lines` list. In the interactive ipython session below, you can see that the `Axes.lines` list is length one and contains the same line that was returned by the `line, = ax.plot...` call:

```
In [101]: ax.lines[0]
Out[101]: <matplotlib.lines.Line2D instance at 0x19a95710>

In [102]: line
Out[102]: <matplotlib.lines.Line2D instance at 0x19a95710>
```

If you make subsequent calls to `ax.plot` (and the hold state is "on" which is the default) then additional lines will be added to the list. You can remove lines later simply by calling the list methods; either of these will work:

```
del ax.lines[0]
ax.lines.remove(line)  # one or the other, not both!
```

The Axes also has helper methods to configure and decorate the x-axis and y-axis tick, tick labels and axis labels:

```
xtext = ax.set_xlabel('my xdata') # returns a Text instance
ytext = ax.set_ylabel('my ydata')
```

When you call *ax.set_xlabel*, it passes the information on the *Text* instance of the *XAxis*. Each `Axes` instance contains an *XAxis* and a *YAxis* instance, which handle the layout and drawing of the ticks, tick labels and axis labels.

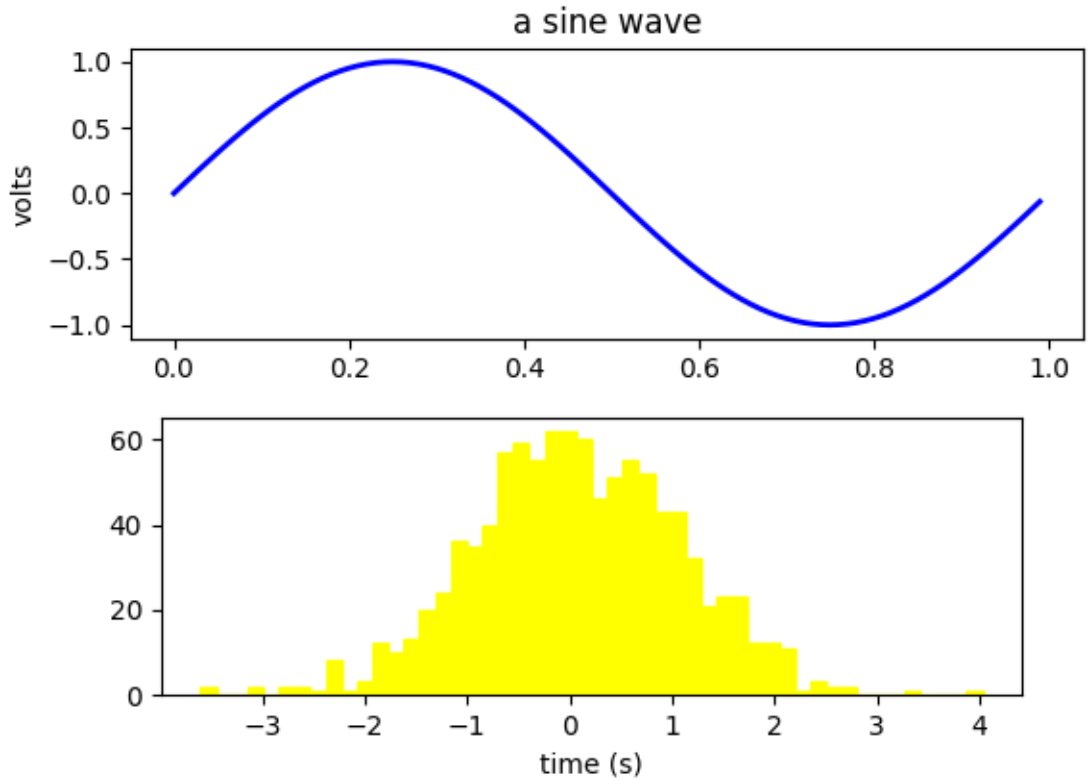
Try creating the figure below.

```
import numpy as np
import matplotlib.pyplot as plt

fig = plt.figure()
fig.subplots_adjust(top=0.8)
ax1 = fig.add_subplot(211)
ax1.set_ylabel('volts')
ax1.set_title('a sine wave')

t = np.arange(0.0, 1.0, 0.01)
s = np.sin(2*np.pi*t)
line, = ax1.plot(t, s, color='blue', lw=2)

# Fixing random state for reproducibility
```

(continues on next page)

(continued from previous page)

```
np.random.seed(19680801)

ax2 = fig.add_axes([0.15, 0.1, 0.7, 0.3])
n, bins, patches = ax2.hist(np.random.randn(1000), 50,
                            facecolor='yellow', edgecolor='yellow')
ax2.set_xlabel('time (s)')

plt.show()
```



Customizing your objects

Every element in the figure is represented by a matplotlib *Artist*, and each has an extensive list of properties to configure its appearance. The figure itself contains a *Rectangle* exactly the size of the figure, which you can use to set the background color and transparency of the figures. Likewise, each *Axes* bounding box (the standard white box with black edges in the typical matplotlib plot, has a `Rectangle` instance that determines the color, transparency, and other properties of the Axes. These instances are stored as member variables `Figure.patch` and `Axes.patch` ("Patch" is a name inherited from MATLAB, and is a 2D "patch" of color on the figure, e.g., rectangles, circles and polygons). Every matplotlib `Artist` has the following properties

Property	Description
alpha	The transparency - a scalar from 0-1
animated	A boolean that is used to facilitate animated drawing
axes	The axes that the Artist lives in, possibly None
clip_box	The bounding box that clips the Artist
clip_on	Whether clipping is enabled
clip_path	The path the artist is clipped to
contains	A picking function to test whether the artist contains the pick point
figure	The figure instance the artist lives in, possibly None
label	A text label (e.g., for auto-labeling)
picker	A python object that controls object picking
transform	The transformation
visible	A boolean whether the artist should be drawn
zorder	A number which determines the drawing order
rasterized	Boolean; Turns vectors into raster graphics (for compression & eps transparency)

Each of the properties is accessed with an old-fashioned setter or getter (yes we know this irritates Pythonistas and we plan to support direct access via properties or traits but it hasn't been done yet). For example, to multiply the current alpha by a half:

```
a = o.get_alpha()
o.set_alpha(0.5*a)
```

If you want to set a number of properties at once, you can also use the `set` method with keyword arguments. For example:

```
o.set(alpha=0.5, zorder=2)
```

If you are working interactively at the python shell, a handy way to inspect the `Artist` properties is to use the *matplotlib.artist.getp()* function (simply *getp()* in pyplot), which lists the properties and their values. This works for classes derived from `Artist` as well, e.g., `Figure` and `Rectangle`. Here are the `Figure` rectangle properties mentioned above:

```
In [149]: matplotlib.artist.getp(fig.patch)
    alpha = 1.0
    animated = False
    antialiased or aa = True
    axes = None
    clip_box = None
    clip_on = False
    clip_path = None
    contains = None
    edgecolor or ec = w
    facecolor or fc = 0.75
```

(continues on next page)

(continued from previous page)

```
    figure = Figure(8.125x6.125)
    fill = 1
    hatch = None
    height = 1
    label =
    linewidth or lw = 1.0
    picker = None
    transform = <Affine object at 0x134cca84>
    verts = ((0, 0), (0, 1), (1, 1), (1, 0))
    visible = True
    width = 1
    window_extent = <Bbox object at 0x134acbcc>
    x = 0
    y = 0
    zorder = 1
```

The docstrings for all of the classes also contain the `Artist` properties, so you can consult the interactive "help" or the *matplotlib.artist* for a listing of properties for a given object.

Object containers

Now that we know how to inspect and set the properties of a given object we want to configure, we need to know how to get at that object. As mentioned in the introduction, there are two kinds of objects: primitives and containers. The primitives are usually the things you want to configure (the font of a *Text* instance, the width of a *Line2D*) although the containers also have some properties as well -- for example the *Axes Artist* is a container that contains many of the primitives in your plot, but it also has properties like the `xscale` to control whether the xaxis is 'linear' or 'log'. In this section we'll review where the various container objects store the `Artists` that you want to get at.

Figure container

The top level container `Artist` is the *matplotlib.figure.Figure*, and it contains everything in the figure. The background of the figure is a *Rectangle* which is stored in `Figure.patch`. As you add subplots (*add_subplot()*) and axes (*add_axes()*) to the figure these will be appended to the *Figure.axes*. These are also returned by the methods that create them:

```
In [156]: fig = plt.figure()

In [157]: ax1 = fig.add_subplot(211)

In [158]: ax2 = fig.add_axes([0.1, 0.1, 0.7, 0.3])
```

(continues on next page)

(continued from previous page)

```
In [159]: ax1
Out[159]: <matplotlib.axes.Subplot instance at 0xd54b26c>

In [160]: print(fig.axes)
[<matplotlib.axes.Subplot instance at 0xd54b26c>,
<matplotlib.axes.Axes instance at 0xd3f0b2c>]
```

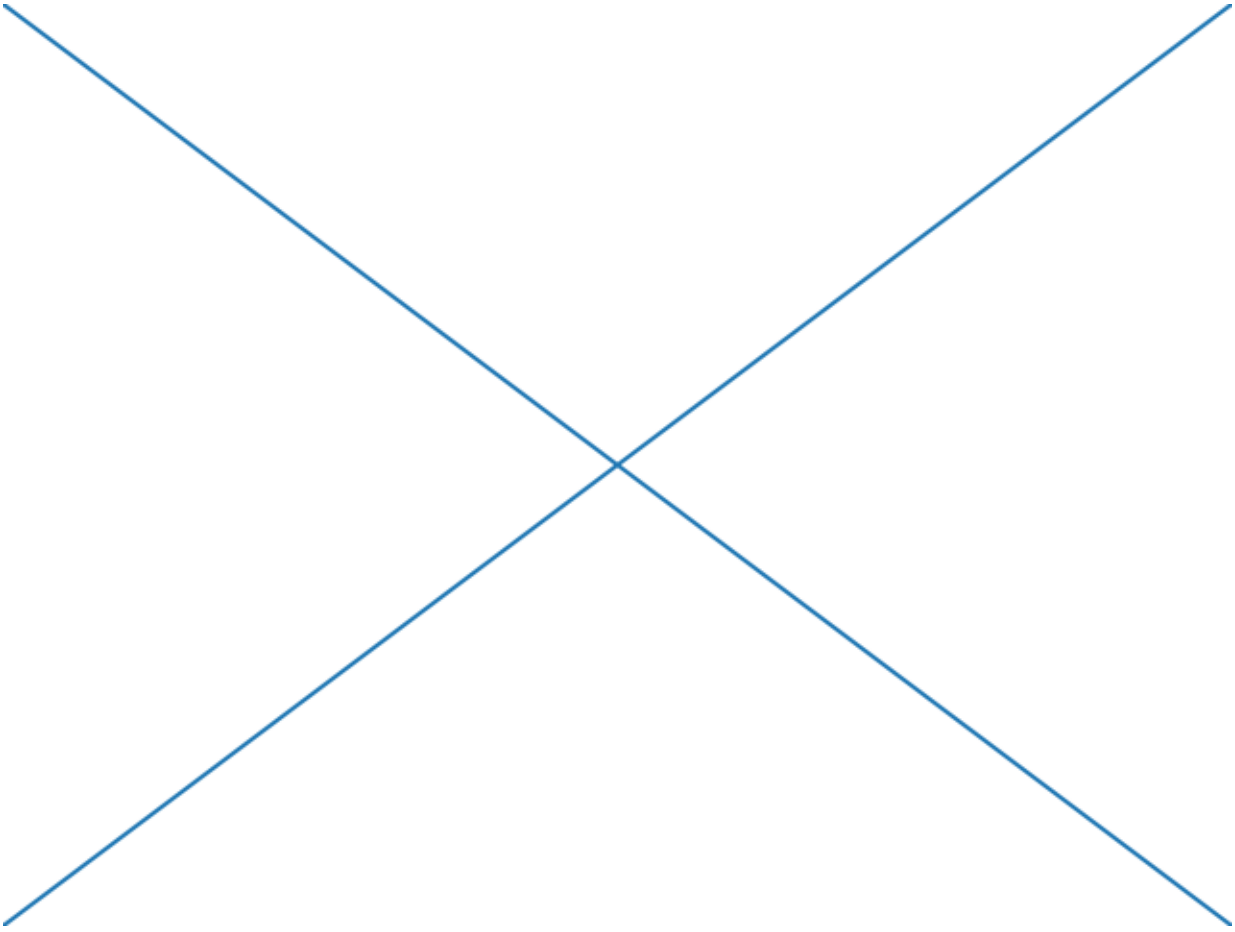
Because the figure maintains the concept of the "current axes" (see *Figure.gca* and *Figure.sca*) to support the pylab/pyplot state machine, you should not insert or remove axes directly from the axes list, but rather use the *add_subplot()* and *add_axes()* methods to insert, and the *delaxes()* method to delete. You are free however, to iterate over the list of axes or index into it to get access to `Axes` instances you want to customize. Here is an example which turns all the axes grids on:

```
for ax in fig.axes:
    ax.grid(True)
```

The figure also has its own text, lines, patches and images, which you can use to add primitives directly. The default coordinate system for the `Figure` will simply be in pixels (which is not usually what you want) but you can control this by setting the transform property of the `Artist` you are adding to the figure.

More useful is "figure coordinates" where (0, 0) is the bottom-left of the figure and (1, 1) is the top-right of the figure which you can obtain by setting the `Artist` transform to `fig.transFigure`:

```
import matplotlib.lines as lines

fig = plt.figure()

l1 = lines.Line2D([0, 1], [0, 1], transform=fig.transFigure, figure=fig)
l2 = lines.Line2D([0, 1], [1, 0], transform=fig.transFigure, figure=fig)
fig.lines.extend([l1, l2])

plt.show()
```



Here is a summary of the Artists the figure contains

Figure attribute	Description
axes	A list of Axes instances (includes Subplot)
patch	The Rectangle background
images	A list of FigureImage patches - useful for raw pixel display
legends	A list of Figure Legend instances (different from Axes.legends)
lines	A list of Figure Line2D instances (rarely used, see Axes.lines)
patches	A list of Figure patches (rarely used, see Axes.patches)
texts	A list Figure Text instances

Axes container

The *matplotlib.axes.Axes* is the center of the matplotlib universe -- it contains the vast majority of all the `Artists` used in a figure with many helper methods to create and add these `Artists` to itself, as well as helper methods to access and customize the `Artists` it contains. Like the *Figure*, it contains a *Patch* `patch` which is a *Rectangle* for Cartesian coordinates and a *Circle* for polar coordinates; this patch determines the shape, background and border of the plotting region:

```
ax = fig.add_subplot(111)
rect = ax.patch  # a Rectangle instance
rect.set_facecolor('green')
```

When you call a plotting method, e.g., the canonical *plot()* and pass in arrays or lists of values, the method will create a *matplotlib.lines.Line2D()* instance, update the line with all the `Line2D` properties passed as keyword arguments, add the line to the `Axes.lines` container, and returns it to you:

```
In [213]: x, y = np.random.rand(2, 100)

In [214]: line, = ax.plot(x, y, '-', color='blue', linewidth=2)
```

`plot` returns a list of lines because you can pass in multiple x, y pairs to plot, and we are unpacking the first element of the length one list into the line variable. The line has been added to the `Axes.lines` list:

```
In [229]: print(ax.lines)
[<matplotlib.lines.Line2D instance at 0xd378b0c>]
```

Similarly, methods that create patches, like *bar()* creates a list of rectangles, will add the patches to the `Axes.patches` list:

```
In [233]: n, bins, rectangles = ax.hist(np.random.randn(1000), 50)

In [234]: rectangles
Out[234]: <a list of 50 Patch objects>

In [235]: print(len(ax.patches))
```

You should not add objects directly to the `Axes.lines` or `Axes.patches` lists unless you know exactly what you are doing, because the `Axes` needs to do a few things when it creates and adds an object. It sets the figure and axes property of the `Artist`, as well as the default `Axes` transformation (unless a transformation is set). It also inspects the data contained in the `Artist` to update the data structures controlling auto-scaling, so that the view limits can be adjusted to contain the plotted data. You can, nonetheless, create objects yourself and add them directly to the `Axes` using helper methods like *add_line()* and *add_patch()*. Here is an annotated interactive session illustrating what is going on:

```
In [262]: fig, ax = plt.subplots()

# create a rectangle instance
In [263]: rect = matplotlib.patches.Rectangle((1, 1), width=5, height=12)

# by default the axes instance is None
In [264]: print(rect.axes)
None

# and the transformation instance is set to the "identity transform"
```

(continues on next page)

(continued from previous page)

```
In [265]: print(rect.get_transform())
<Affine object at 0x13695544>

# now we add the Rectangle to the Axes
In [266]: ax.add_patch(rect)

# and notice that the ax.add_patch method has set the axes
# instance
In [267]: print(rect.axes)
Axes(0.125,0.1;0.775x0.8)

# and the transformation has been set too
In [268]: print(rect.get_transform())
<Affine object at 0x15009ca4>

# the default axes transformation is ax.transData
In [269]: print(ax.transData)
<Affine object at 0x15009ca4>

# notice that the xlimits of the Axes have not been changed
In [270]: print(ax.get_xlim())
(0.0, 1.0)

# but the data limits have been updated to encompass the rectangle
In [271]: print(ax.dataLim.bounds)
(1.0, 1.0, 5.0, 12.0)

# we can manually invoke the auto-scaling machinery
In [272]: ax.autoscale_view()

# and now the xlim are updated to encompass the rectangle
In [273]: print(ax.get_xlim())
(1.0, 6.0)

# we have to manually force a figure draw
In [274]: ax.figure.canvas.draw()
```

There are many, many `Axes` helper methods for creating primitive `Artists` and adding them to their respective containers. The table below summarizes a small sampling of them, the kinds of `Artist` they create, and where they store them

Helper method	Artist	Container
ax.annotate - text annotations	Annotate	ax.texts
ax.bar - bar charts	Rectangle	ax.patches
ax.errorbar - error bar plots	Line2D and Rectangle	ax.lines and ax.patches
ax.fill - shared area	Polygon	ax.patches
ax.hist - histograms	Rectangle	ax.patches
ax.imshow - image data	AxesImage	ax.images
ax.legend - axes legends	Legend	ax.legends
ax.plot - xy plots	Line2D	ax.lines
ax.scatter - scatter charts	PolygonCollection	ax.collections
ax.text - text	Text	ax.texts

In addition to all of these `Artists`, the `Axes` contains two important `Artist` containers: the *`XAxis`* and *`YAxis`*, which handle the drawing of the ticks and labels. These are stored as instance variables `xaxis` and `yaxis`. The `XAxis` and `YAxis` containers will be detailed below, but note that the `Axes` contains many helper methods which forward calls on to the *`Axis`* instances so you often do not need to work with them directly unless you want to. For example, you can set the font color of the `XAxis` ticklabels using the `Axes` helper method:

```
for label in ax.get_xticklabels():
    label.set_color('orange')
```

Below is a summary of the Artists that the Axes contains

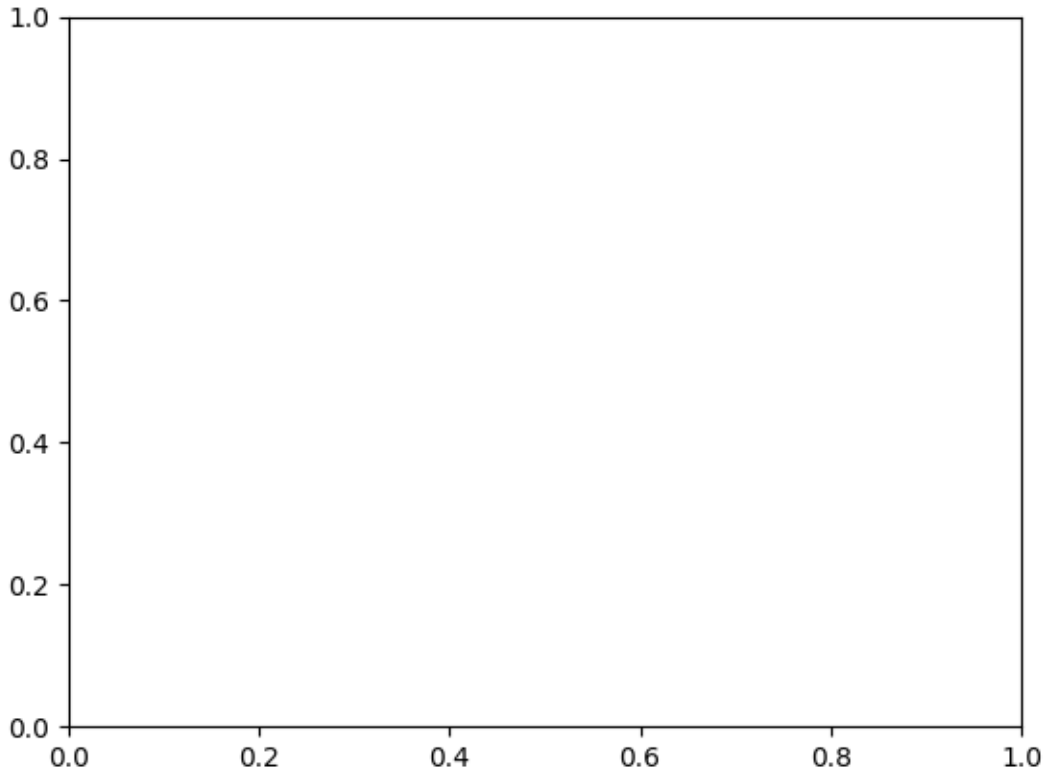
Axes attribute	Description
artists	A list of Artist instances
patch	Rectangle instance for Axes background
collections	A list of Collection instances
images	A list of AxesImage
legends	A list of Legend instances
lines	A list of Line2D instances
patches	A list of Patch instances
texts	A list of Text instances
xaxis	matplotlib.axis.XAxis instance
yaxis	matplotlib.axis.YAxis instance

Axis containers

The *matplotlib.axis.Axis* instances handle the drawing of the tick lines, the grid lines, the tick labels and the axis label. You can configure the left and right ticks separately for the y-axis, and the upper and lower ticks separately for the x-axis. The `Axis` also stores the data and view intervals used in auto-scaling, panning and zooming, as well as the *Locator* and *Formatter* instances which control where the ticks are placed and how they are represented as strings.

Each `Axis` object contains a `label` attribute (this is what *pyplot* modifies in calls to *xlabel* and *ylabel*) as well as a list of major and minor ticks. The ticks are *axis.XTick* and *axis.YTick* instances, which contain the actual line and text primitives that render the ticks and ticklabels. Because the ticks are dynamically created as needed (e.g., when panning and zooming), you should access the lists of major and minor ticks through their accessor methods *axis.Axis.get_major_ticks* and *axis.Axis.get_minor_ticks*. Although the ticks contain all the primitives and will be covered below, `Axis` instances have accessor methods that return the tick lines, tick labels, tick locations etc.:

```
fig, ax = plt.subplots()
axis = ax.xaxis
axis.get_ticklocs()
```



Out:

```
array([0. , 0.2, 0.4, 0.6, 0.8, 1. ])
```

```
axis.get_ticklabels()
```

Out:

```
[Text(0.0, 0, '0.0'), Text(0.2, 0, '0.2'), Text(0.4, 0, '0.4'), Text(0.6000000000000001,␣
↪0, '0.6'), Text(0.8, 0, '0.8'), Text(1.0, 0, '1.0')]
```

note there are twice as many ticklines as labels because by default there are tick lines at the top and bottom but only tick labels below the xaxis; however, this can be customized.

```
axis.get_ticklines()
```

Out:

```
<a list of 12 Line2D ticklines objects>
```

And with the above methods, you only get lists of major ticks back by default, but you can also ask for the minor ticks:

```
axis.get_ticklabels(minor=True)
axis.get_ticklines(minor=True)
```

Out:

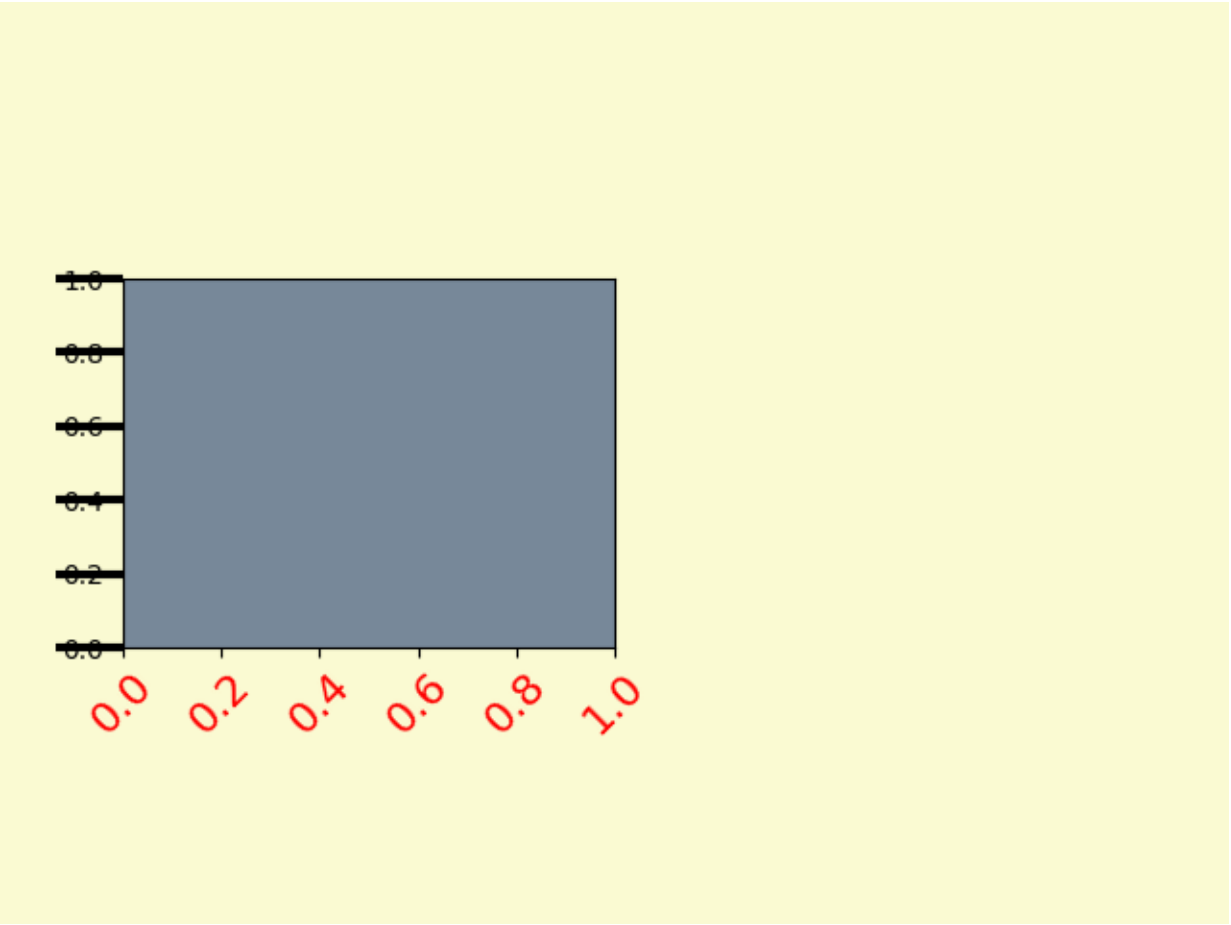
```
<a list of 0 Line2D ticklines objects>
```

Here is a summary of some of the useful accessor methods of the `Axis` (these have corresponding setters where useful, such as *set_major_formatter()*.)

Accessor method	Description
get_scale	The scale of the axis, e.g., 'log' or 'linear'
get_view_interval	The interval instance of the axis view limits
get_data_interval	The interval instance of the axis data limits
get_gridlines	A list of grid lines for the Axis
get_label	The axis label - a Text instance
get_ticklabels	A list of Text instances - keyword minor=True|False
get_ticklines	A list of Line2D instances - keyword minor=True|False
get_ticklocs	A list of Tick locations - keyword minor=True|False
get_major_locator	The *ticker.Locator* instance for major ticks
get_major_formatter	The *ticker.Formatter* instance for major ticks
get_minor_locator	The *ticker.Locator* instance for minor ticks
get_minor_formatter	The *ticker.Formatter* instance for minor ticks
get_major_ticks	A list of Tick instances for major ticks
get_minor_ticks	A list of Tick instances for minor ticks
grid	Turn the grid on or off for the major or minor ticks

Here is an example, not recommended for its beauty, which customizes the axes and tick properties

```
# plt.figure creates a matplotlib.figure.Figure instance
fig = plt.figure()
rect = fig.patch  # a rectangle instance
rect.set_facecolor('lightgoldenrodyellow')

ax1 = fig.add_axes([0.1, 0.3, 0.4, 0.4])
rect = ax1.patch
rect.set_facecolor('lightslategray')


for label in ax1.xaxis.get_ticklabels():
    # label is a Text instance
    label.set_color('red')
    label.set_rotation(45)
    label.set_fontsize(16)

for line in ax1.yaxis.get_ticklines():
    # line is a Line2D instance
    line.set_color('green')
    line.set_markersize(25)
    line.set_markeredgewidth(3)

plt.show()
```



Tick containers

The *matplotlib.axis.Tick* is the final container object in our descent from the *Figure* to the *Axes* to the *Axis* to the *Tick*. The `Tick` contains the tick and grid line instances, as well as the label instances for the upper and lower ticks. Each of these is accessible directly as an attribute of the `Tick`.

Tick attribute	Description
tick1line	Line2D instance
tick2line	Line2D instance
gridline	Line2D instance
label1	Text instance
label2	Text instance

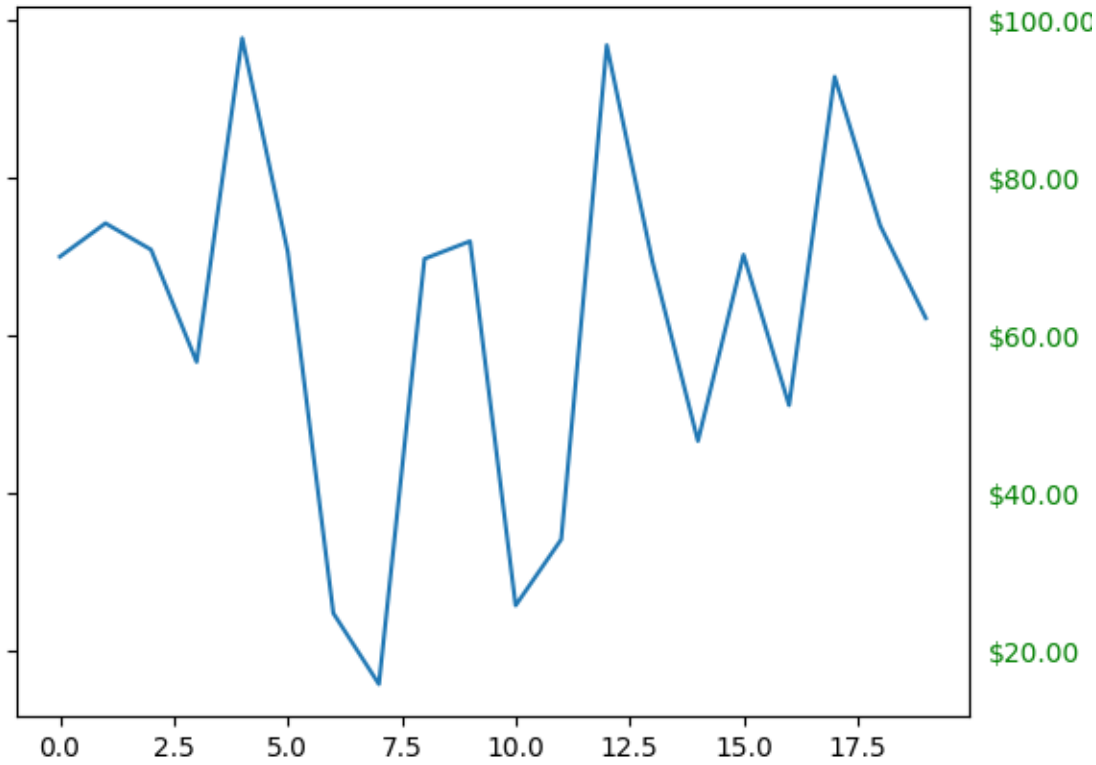
Here is an example which sets the formatter for the right side ticks with dollar signs and colors them green on the right side of the yaxis.

```
import numpy as np
import matplotlib.pyplot as plt
```

(continues on next page)

(continued from previous page)

```
# Fixing random state for reproducibility
np.random.seed(19680801)

fig, ax = plt.subplots()
ax.plot(100*np.random.rand(20))

# Use automatic StrMethodFormatter
ax.yaxis.set_major_formatter('${x:1.2f}')

ax.yaxis.set_tick_params(which='major', labelcolor='green',
                         labelleft=False, labelright=True)

plt.show()
```



2.2.2 Legend guide

Generating legends flexibly in Matplotlib.

This legend guide is an extension of the documentation available at *legend()* - please ensure you are familiar with contents of that documentation before proceeding with this guide.

This guide makes use of some common terms, which are documented here for clarity:

legend entry

A legend is made up of one or more legend entries. An entry is made up of exactly one key and one label.

legend key

The colored/patterned marker to the left of each legend label.

legend label

The text which describes the handle represented by the key.

legend handle

The original object which is used to generate an appropriate entry in the legend.

Controlling the legend entries

Calling *legend()* with no arguments automatically fetches the legend handles and their associated labels. This functionality is equivalent to:

```
handles, labels = ax.get_legend_handles_labels()
ax.legend(handles, labels)
```

The *get_legend_handles_labels()* function returns a list of handles/artists which exist on the Axes which can be used to generate entries for the resulting legend - it is worth noting however that not all artists can be added to a legend, at which point a "proxy" will have to be created (see *Creating artists specifically for adding to the legend (aka. Proxy artists)* for further details).

Those artists with an empty string as label or with a label starting with "_" will be ignored.

For full control of what is being added to the legend, it is common to pass the appropriate handles directly to *legend()*:

```
line_up, = plt.plot([1, 2, 3], label='Line 2')
line_down, = plt.plot([3, 2, 1], label='Line 1')
plt.legend(handles=[line_up, line_down])
```

In some cases, it is not possible to set the label of the handle, so it is possible to pass through the list of labels to *legend()*:

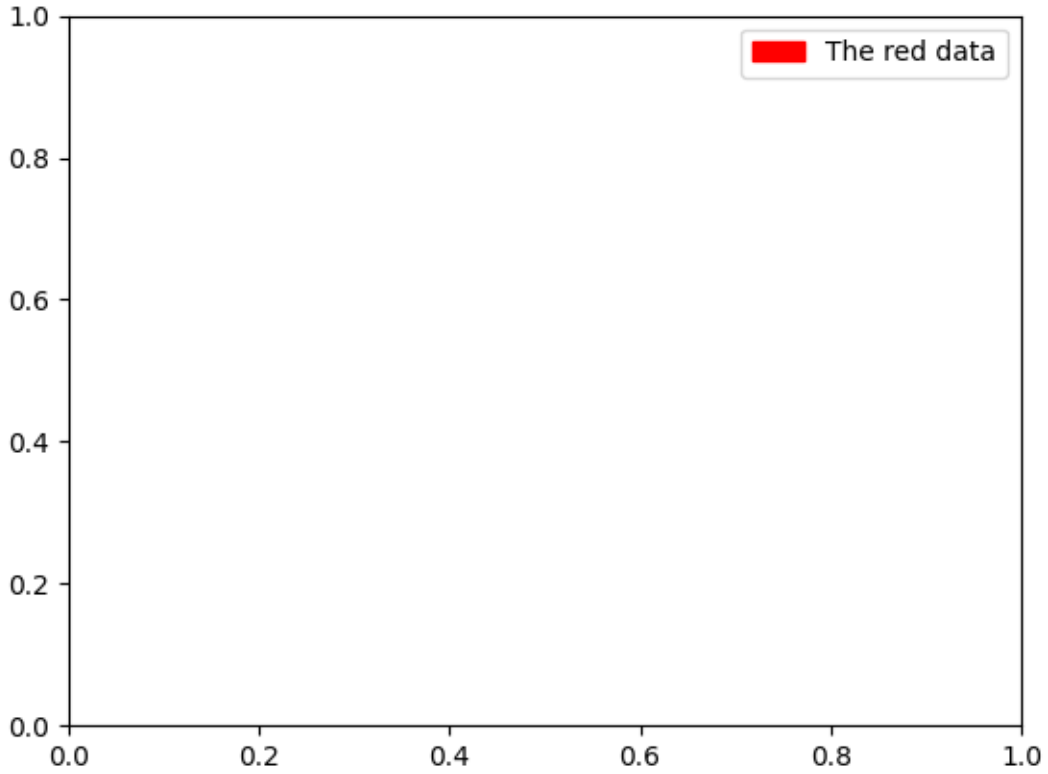
```
line_up, = plt.plot([1, 2, 3], label='Line 2')
line_down, = plt.plot([3, 2, 1], label='Line 1')
plt.legend([line_up, line_down], ['Line Up', 'Line Down'])
```

Creating artists specifically for adding to the legend (aka. Proxy artists)

Not all handles can be turned into legend entries automatically, so it is often necessary to create an artist which *can*. Legend handles don't have to exist on the Figure or Axes in order to be used.

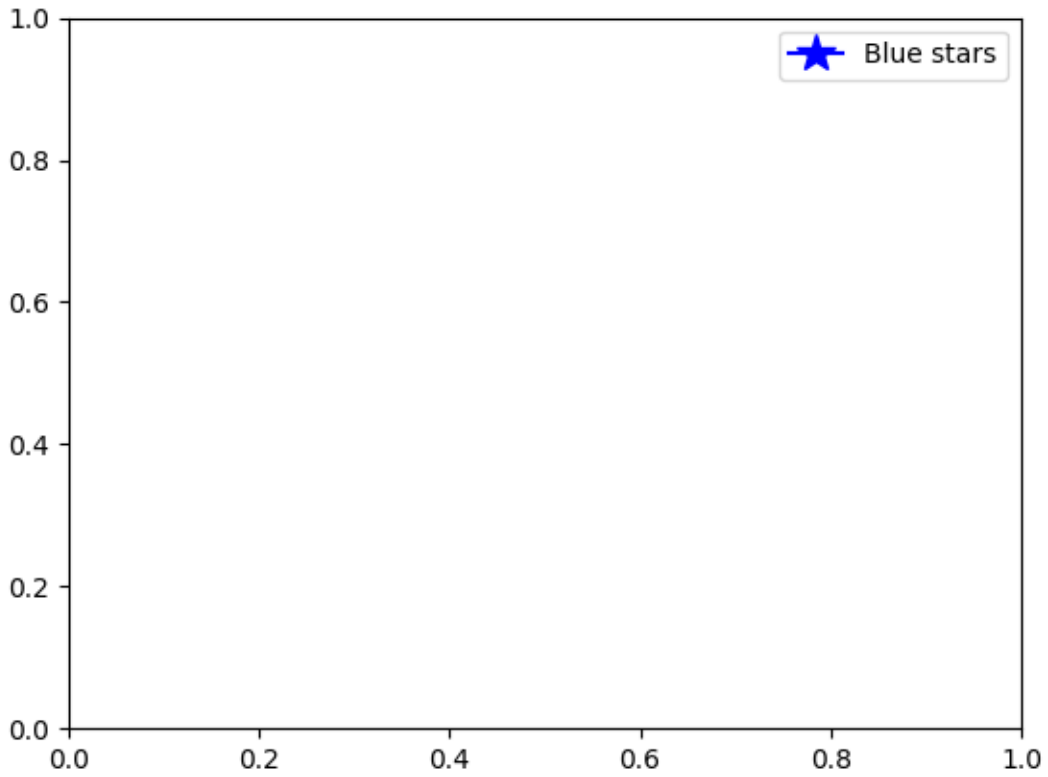
Suppose we wanted to create a legend which has an entry for some data which is represented by a red color:

```
import matplotlib.patches as mpatches
import matplotlib.pyplot as plt

red_patch = mpatches.Patch(color='red', label='The red data')
plt.legend(handles=[red_patch])

plt.show()
```



There are many supported legend handles. Instead of creating a patch of color we could have created a line with a marker:

```
import matplotlib.lines as mlines

blue_line = mlines.Line2D([], [], color='blue', marker='*',
                          markersize=15, label='Blue stars')
plt.legend(handles=[blue_line])

plt.show()
```



Legend location

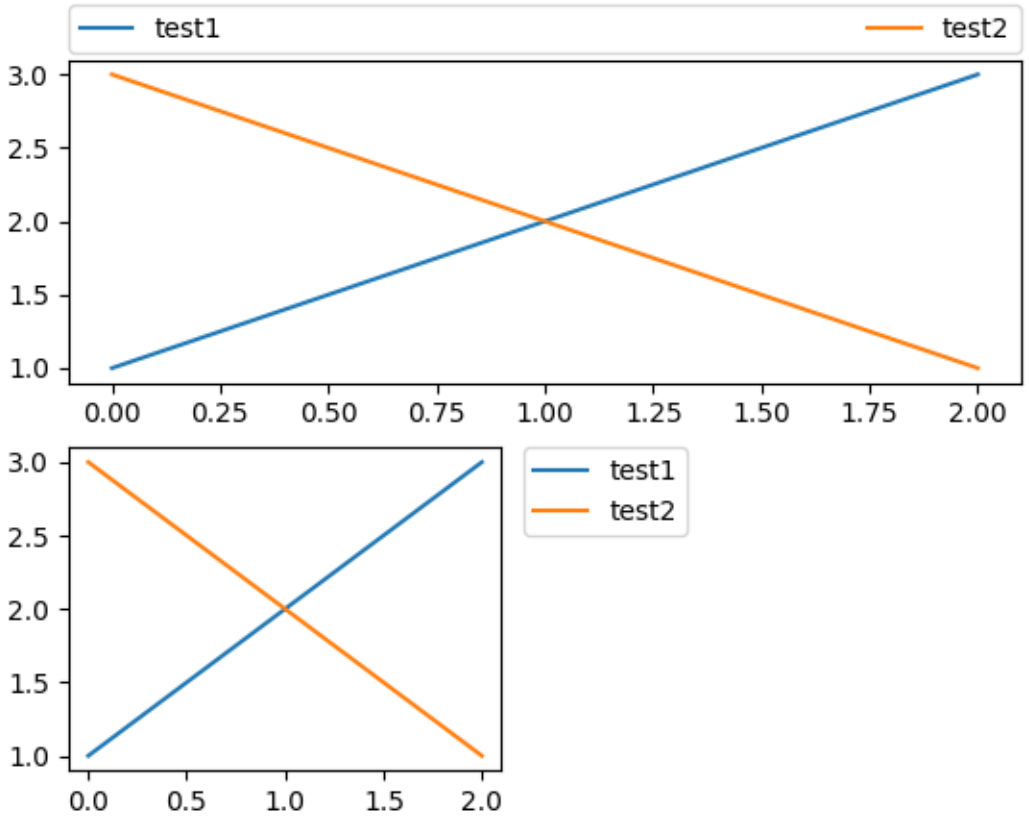
The location of the legend can be specified by the keyword argument *loc*. Please see the documentation at *legend()* for more details.

The `bbox_to_anchor` keyword gives a great degree of control for manual legend placement. For example, if you want your axes legend located at the figure's top right-hand corner instead of the axes' corner, simply specify the corner's location and the coordinate system of that location:

```
plt.legend(bbox_to_anchor=(1, 1),
           bbox_transform=plt.gcf().transFigure)
```

More examples of custom legend placement:

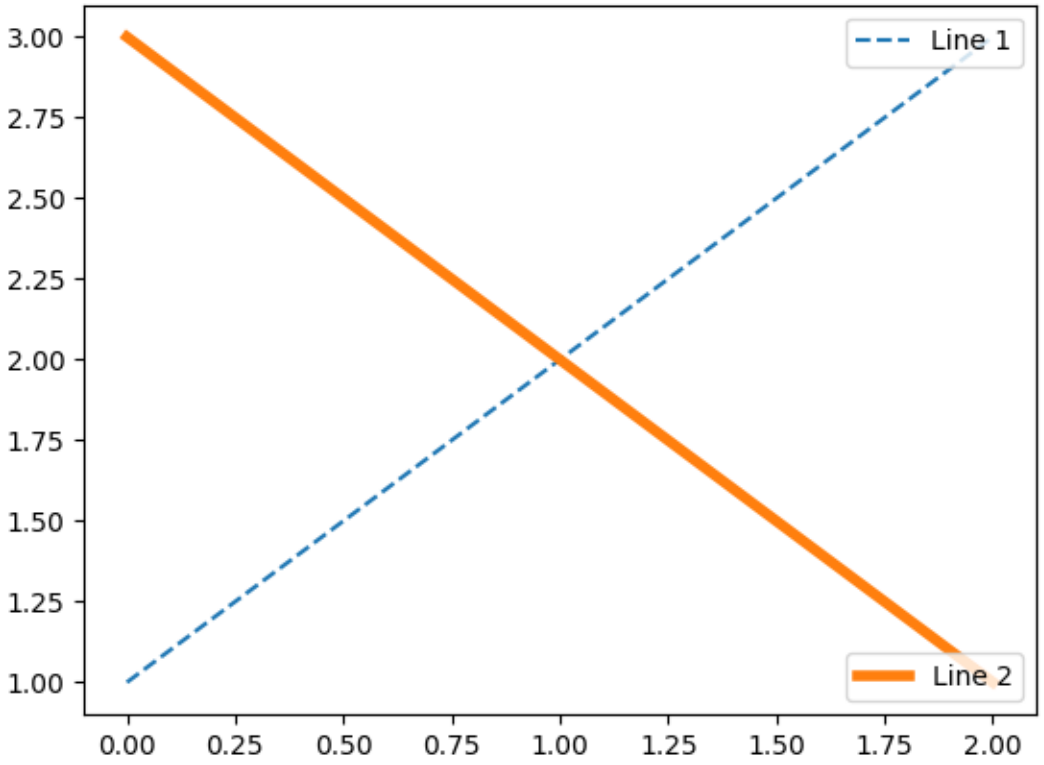
```
plt.subplot(211)
plt.plot([1, 2, 3], label="test1")
plt.plot([3, 2, 1], label="test2")

# Place a legend above this subplot, expanding itself to
# fully use the given bounding box.
plt.legend(bbox_to_anchor=(0., 1.02, 1., .102), loc='lower left',
           ncol=2, mode="expand", borderaxespad=0.)

plt.subplot(223)
plt.plot([1, 2, 3], label="test1")
plt.plot([3, 2, 1], label="test2")
# Place a legend to the right of this smaller subplot.
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left', borderaxespad=0.)

plt.show()
```



Multiple legends on the same Axes

Sometimes it is more clear to split legend entries across multiple legends. Whilst the instinctive approach to doing this might be to call the *legend()* function multiple times, you will find that only one legend ever exists on the Axes. This has been done so that it is possible to call *legend()* repeatedly to update the legend to the latest handles on the Axes. To keep old legend instances, we must add them manually to the Axes:

```
line1, = plt.plot([1, 2, 3], label="Line 1", linestyle='--')
line2, = plt.plot([3, 2, 1], label="Line 2", linewidth=4)

# Create a legend for the first line.
first_legend = plt.legend(handles=[line1], loc='upper right')

# Add the legend manually to the current Axes.
ax = plt.gca().add_artist(first_legend)

# Create another legend for the second line.
plt.legend(handles=[line2], loc='lower right')

plt.show()
```



Legend Handlers

In order to create legend entries, handles are given as an argument to an appropriate *HandlerBase* subclass. The choice of handler subclass is determined by the following rules:

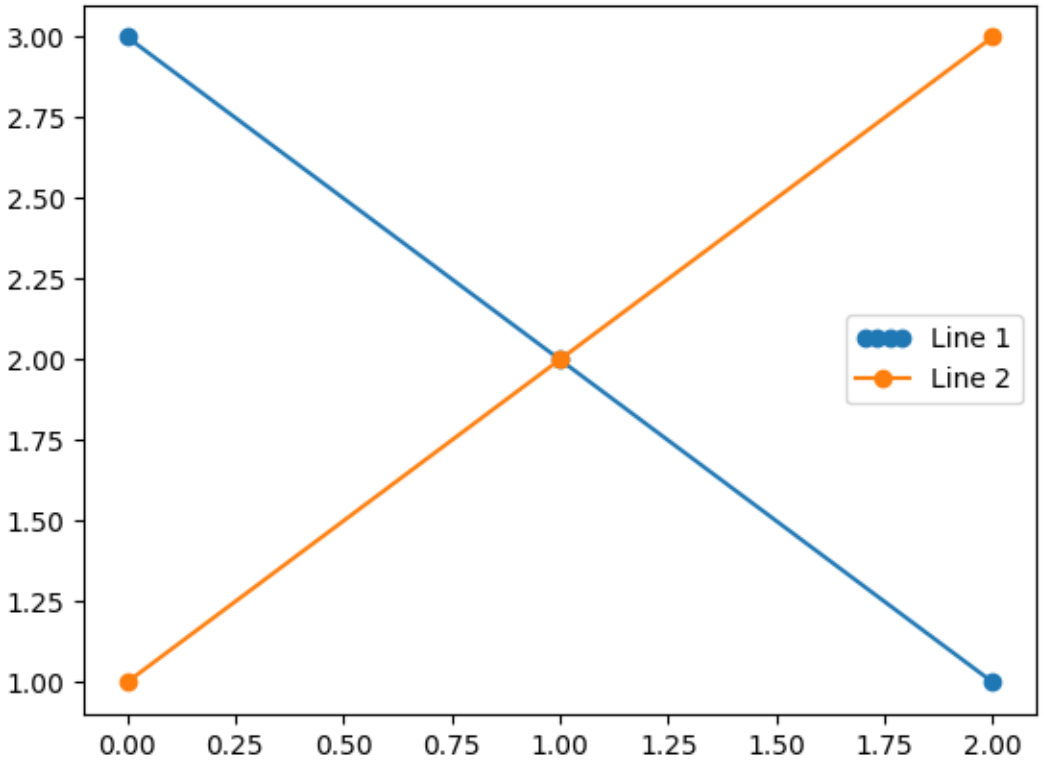
1. Update *get_legend_handler_map()* with the value in the `handler_map` keyword.
2. Check if the `handle` is in the newly created `handler_map`.
3. Check if the type of `handle` is in the newly created `handler_map`.
4. Check if any of the types in the `handle`'s mro is in the newly created `handler_map`.

For completeness, this logic is mostly implemented in *get_legend_handler()*.

All of this flexibility means that we have the necessary hooks to implement custom handlers for our own type of legend key.

The simplest example of using custom handlers is to instantiate one of the existing *legend_handler.HandlerBase* subclasses. For the sake of simplicity, let's choose *legend_handler.HandlerLine2D* which accepts a *numpoints* argument (numpoints is also a keyword on the *legend()* function for convenience). We can then pass the mapping of instance to Handler as a keyword to legend.

```
from matplotlib.legend_handler import HandlerLine2D

line1, = plt.plot([3, 2, 1], marker='o', label='Line 1')
line2, = plt.plot([1, 2, 3], marker='o', label='Line 2')

plt.legend(handler_map={line1: HandlerLine2D(numpoints=4)})
```



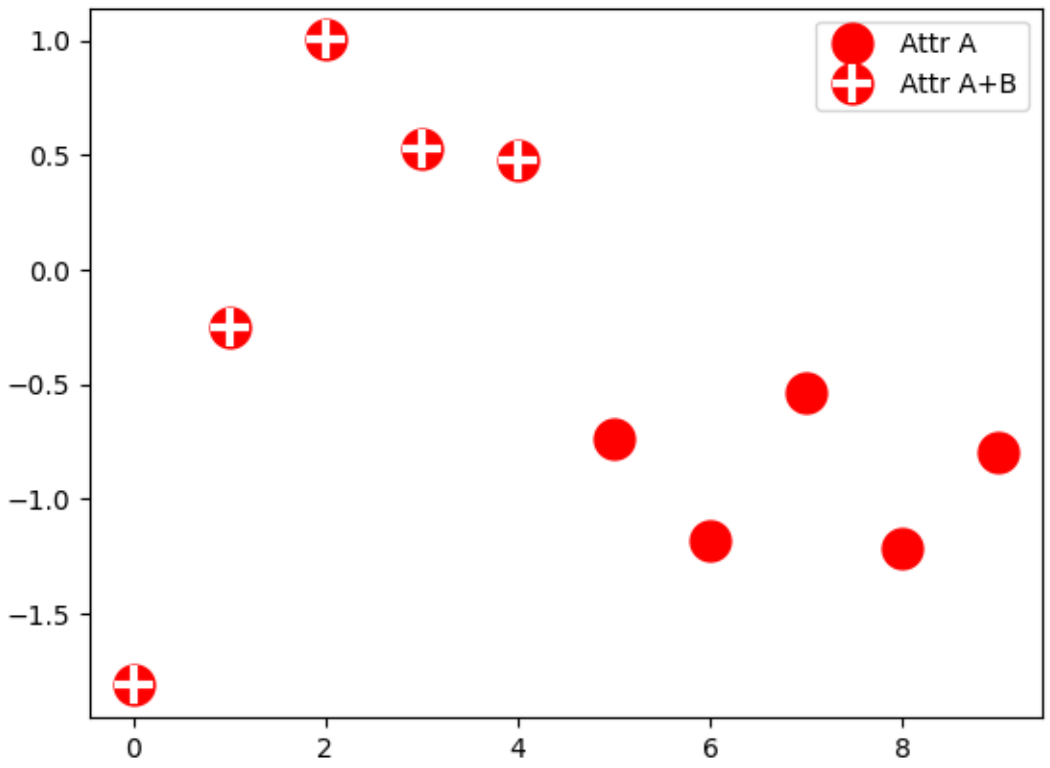
Out:

```
<matplotlib.legend.Legend object at 0x7f540146f9a0>
```

As you can see, "Line 1" now has 4 marker points, where "Line 2" has 2 (the default). Try the above code, only change the map's key from `line1` to `type(line1)`. Notice how now both *Line2D* instances get 4 markers.

Along with handlers for complex plot types such as errorbars, stem plots and histograms, the default `handler_map` has a special `tuple` handler (*legend_handler.HandlerTuple*) which simply plots the handles on top of one another for each item in the given tuple. The following example demonstrates combining two legend keys on top of one another:

```
from numpy.random import randn

z = randn(10)

red_dot, = plt.plot(z, "ro", markersize=15)
# Put a white cross over some of the data.
white_cross, = plt.plot(z[:5], "w+", markeredgewidth=3, markersize=15)

plt.legend([red_dot, (red_dot, white_cross)], ["Attr A", "Attr A+B"])
```

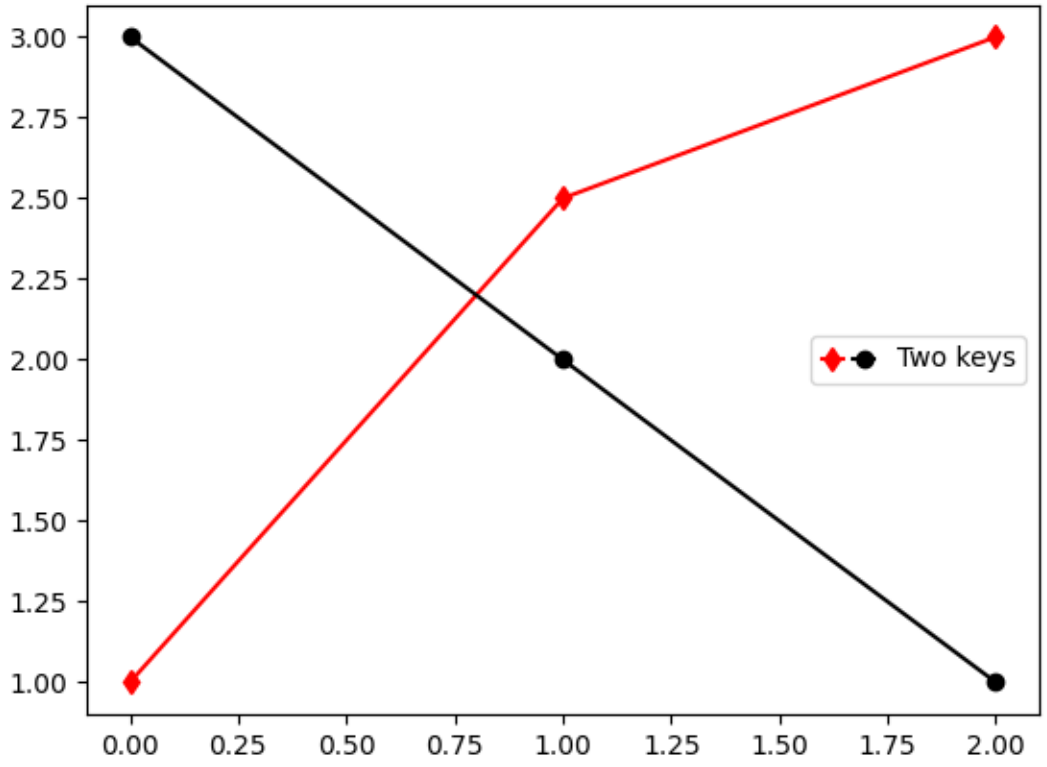


Out:

```
<matplotlib.legend.Legend object at 0x7f53fac8fd60>
```

The *legend_handler.HandlerTuple* class can also be used to assign several legend keys to the same entry:

```
from matplotlib.legend_handler import HandlerLine2D, HandlerTuple

p1, = plt.plot([1, 2.5, 3], 'r-d')
p2, = plt.plot([3, 2, 1], 'k-o')

l = plt.legend([(p1, p2)], ['Two keys'], numpoints=1,
               handler_map={tuple: HandlerTuple(ndivide=None)})
```



Implementing a custom legend handler

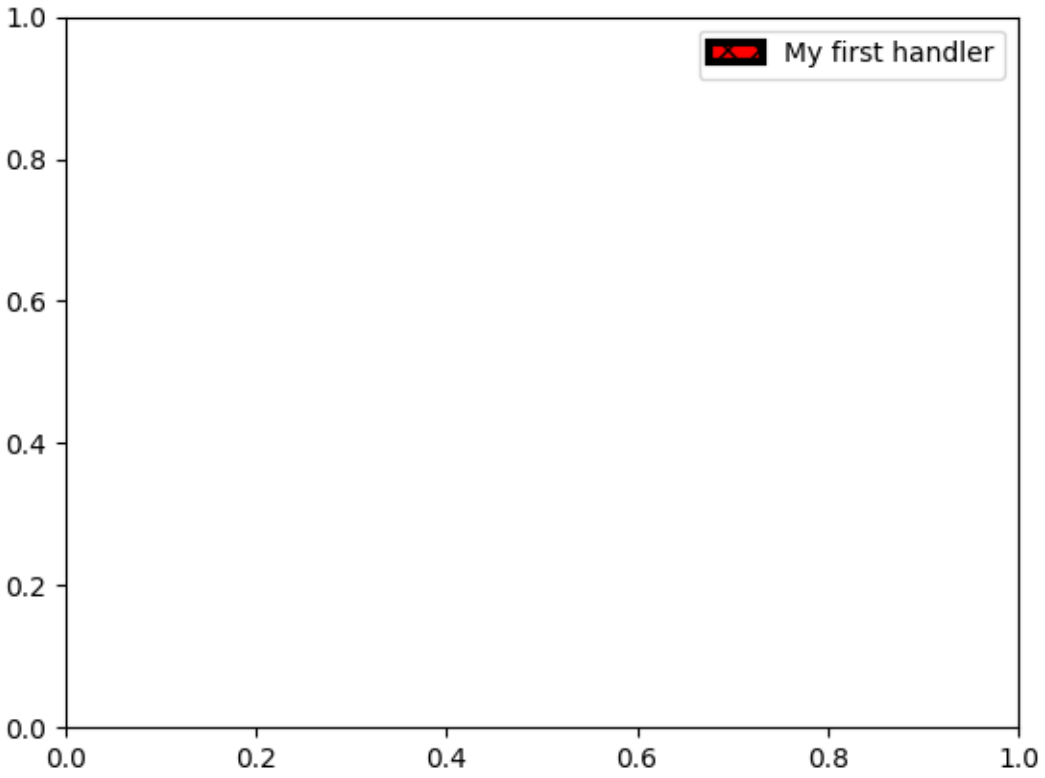
A custom handler can be implemented to turn any handle into a legend key (handles don't necessarily need to be matplotlib artists). The handler must implement a `legend_artist` method which returns a single artist for the legend to use. The required signature for `legend_artist` is documented at *`legend_artist`*.

```
import matplotlib.patches as mpatches


class AnyObject:
    pass


class AnyObjectHandler:
    def legend_artist(self, legend, orig_handle, fontsize, handlebox):
        x0, y0 = handlebox.xdescent, handlebox.ydescent
        width, height = handlebox.width, handlebox.height
        patch = mpatches.Rectangle([x0, y0], width, height, facecolor='red',
                                   edgecolor='black', hatch='xx', lw=3,
                                   transform=handlebox.get_transform())
```

(continues on next page)

(continued from previous page)

```
        handlebox.add_artist(patch)
        return patch


plt.legend([AnyObject()], ['My first handler'],
           handler_map={AnyObject: AnyObjectHandler()})
```



Out:

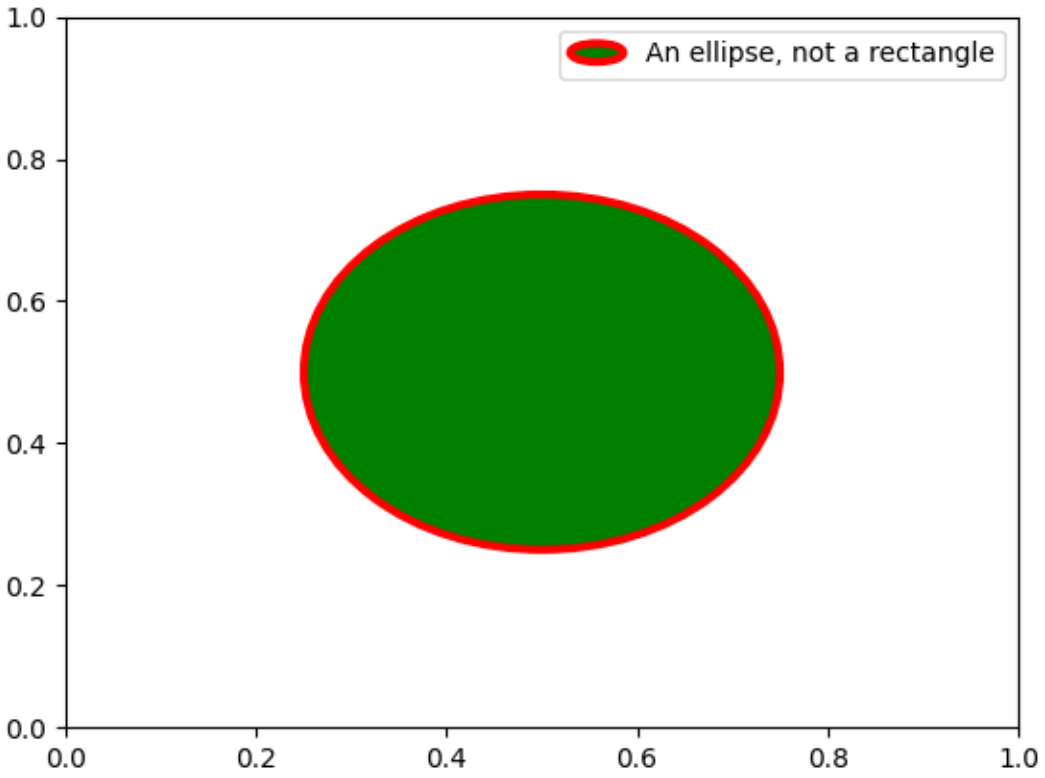
```
<matplotlib.legend.Legend object at 0x7f53fed02340>
```

Alternatively, had we wanted to globally accept `AnyObject` instances without needing to manually set the *handler_map* keyword all the time, we could have registered the new handler with:

```
from matplotlib.legend import Legend
Legend.update_default_handler_map({AnyObject: AnyObjectHandler()})
```

Whilst the power here is clear, remember that there are already many handlers implemented and what you want to achieve may already be easily possible with existing classes. For example, to produce elliptical legend keys, rather than rectangular ones:

```
from matplotlib.legend_handler import HandlerPatch


class HandlerEllipse(HandlerPatch):
    def create_artists(self, legend, orig_handle,
                       xdescent, ydescent, width, height, fontsize, trans):
        center = 0.5 * width - 0.5 * xdescent, 0.5 * height - 0.5 * ydescent
        p = mpatches.Ellipse(xy=center, width=width + xdescent,
                             height=height + ydescent)
        self.update_prop(p, orig_handle, legend)
        p.set_transform(trans)
        return [p]


c = mpatches.Circle((0.5, 0.5), 0.25, facecolor="green",
                    edgecolor="red", linewidth=3)
plt.gca().add_patch(c)

plt.legend([c], ["An ellipse, not a rectangle"],
           handler_map={mpatches.Circle: HandlerEllipse()})
```



Out:

```
<matplotlib.legend.Legend object at 0x7f53fee0cc10>
```

Total running time of the script: (0 minutes 2.676 seconds)

2.2.3 Styling with cycler

Demo of custom property-cycle settings to control colors and other style properties for multi-line plots.

Note: More complete documentation of the `cycler` API can be found here.

This example demonstrates two different APIs:

1. Setting the rc parameter specifying the default property cycle. This affects all subsequent axes (but not axes already created).
2. Setting the property cycle for a single pair of axes.

```
from cycler import cycler
import numpy as np
import matplotlib.pyplot as plt
```

First we'll generate some sample data, in this case, four offset sine curves.

```
x = np.linspace(0, 2 * np.pi, 50)
offsets = np.linspace(0, 2 * np.pi, 4, endpoint=False)
yy = np.transpose([np.sin(x + phi) for phi in offsets])
```

Now `yy` has shape

```
print(yy.shape)
```

Out:

```
(50, 4)
```

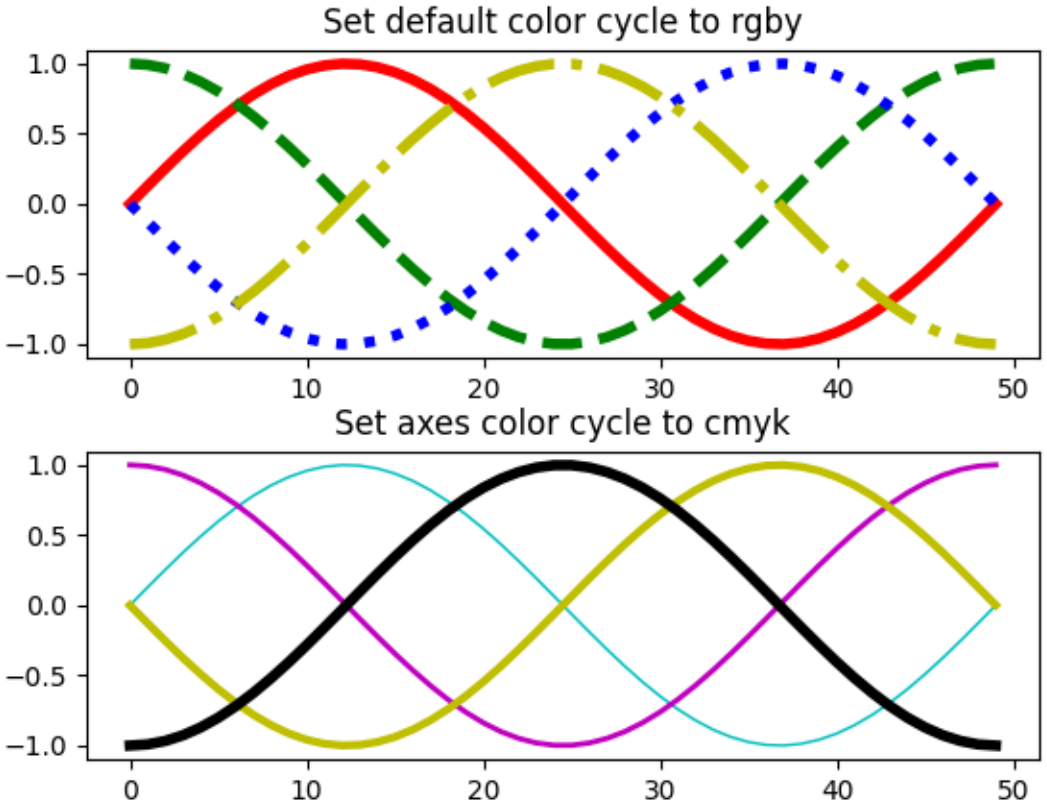
So `yy[:, i]` will give you the `i`-th offset sine curve. Let's set the default `prop_cycle` using *matplotlib.pyplot.rc()*. We'll combine a color cycler and a linestyle cycler by adding (+) two `cycler`'s together. See the bottom of this tutorial for more information about combining different cyclers.

```
default_cycler = (cycler(color=['r', 'g', 'b', 'y']) +
                  cycler(linestyle=['-', '--', ':', '-.']))

plt.rc('lines', linewidth=4)
plt.rc('axes', prop_cycle=default_cycler)
```

Now we'll generate a figure with two axes, one on top of the other. On the first axis, we'll plot with the default cycler. On the second axis, we'll set the `prop_cycle`

using *matplotlib.axes.Axes.set_prop_cycle()*, which will only set the `prop_cycle` for this *matplotlib.axes.Axes* instance. We'll use a second `cycler` that combines a color cycler and a linewidth cycler.

```
custom_cycler = (cycler(color=['c', 'm', 'y', 'k']) +
                 cycler(lw=[1, 2, 3, 4]))

fig, (ax0, ax1) = plt.subplots(nrows=2)
ax0.plot(yy)
ax0.set_title('Set default color cycle to rgby')
ax1.set_prop_cycle(custom_cycler)
ax1.plot(yy)
ax1.set_title('Set axes color cycle to cmyk')

# Add a bit more space between the two plots.
fig.subplots_adjust(hspace=0.3)
plt.show()
```



Setting `prop_cycle` in the `matplotlibrc` file or style files

Remember, a custom cycler can be set in your `matplotlibrc` file or a style file (`style.mplstyle`) under `axes.prop_cycle`:

```
axes.prop_cycle : cycler(color='bgrcmyk')
```

Cycling through multiple properties

You can add cyclers:

```
from cycler import cycler
cc = (cycler(color=list('rgb')) +
      cycler(linestyle=['-', '--', '-.']))
for d in cc:
    print(d)
```

Results in:

```
{'color': 'r', 'linestyle': '-'}
{'color': 'g', 'linestyle': '--'}
{'color': 'b', 'linestyle': '-.'}
```

You can multiply cyclers:

```
from cycler import cycler
cc = (cycler(color=list('rgb')) *
      cycler(linestyle=['-', '--', '-.']))
for d in cc:
    print(d)
```

Results in:

```
{'color': 'r', 'linestyle': '-'}
{'color': 'r', 'linestyle': '--'}
{'color': 'r', 'linestyle': '-.'}
{'color': 'g', 'linestyle': '-'}
{'color': 'g', 'linestyle': '--'}
{'color': 'g', 'linestyle': '-.'}
{'color': 'b', 'linestyle': '-'}
{'color': 'b', 'linestyle': '--'}
{'color': 'b', 'linestyle': '-.'}
```

2.2.4 Customizing Figure Layouts Using GridSpec and Other Functions

How to create grid-shaped combinations of axes.

subplots()

Perhaps the primary function used to create figures and axes. It's also similar to *matplotlib.pyplot.subplot()*, but creates and places all axes on the figure at once. See also *matplotlib.figure.Figure.subplots*.

GridSpec

Specifies the geometry of the grid that a subplot will be placed. The number of rows and number of columns of the grid need to be set. Optionally, the subplot layout parameters (e.g., left, right, etc.) can be tuned.

SubplotSpec

Specifies the location of the subplot in the given *GridSpec*.

subplot2grid()

A helper function that is similar to *subplot()*, but uses 0-based indexing and let subplot to occupy multiple cells. This function is not covered in this tutorial.

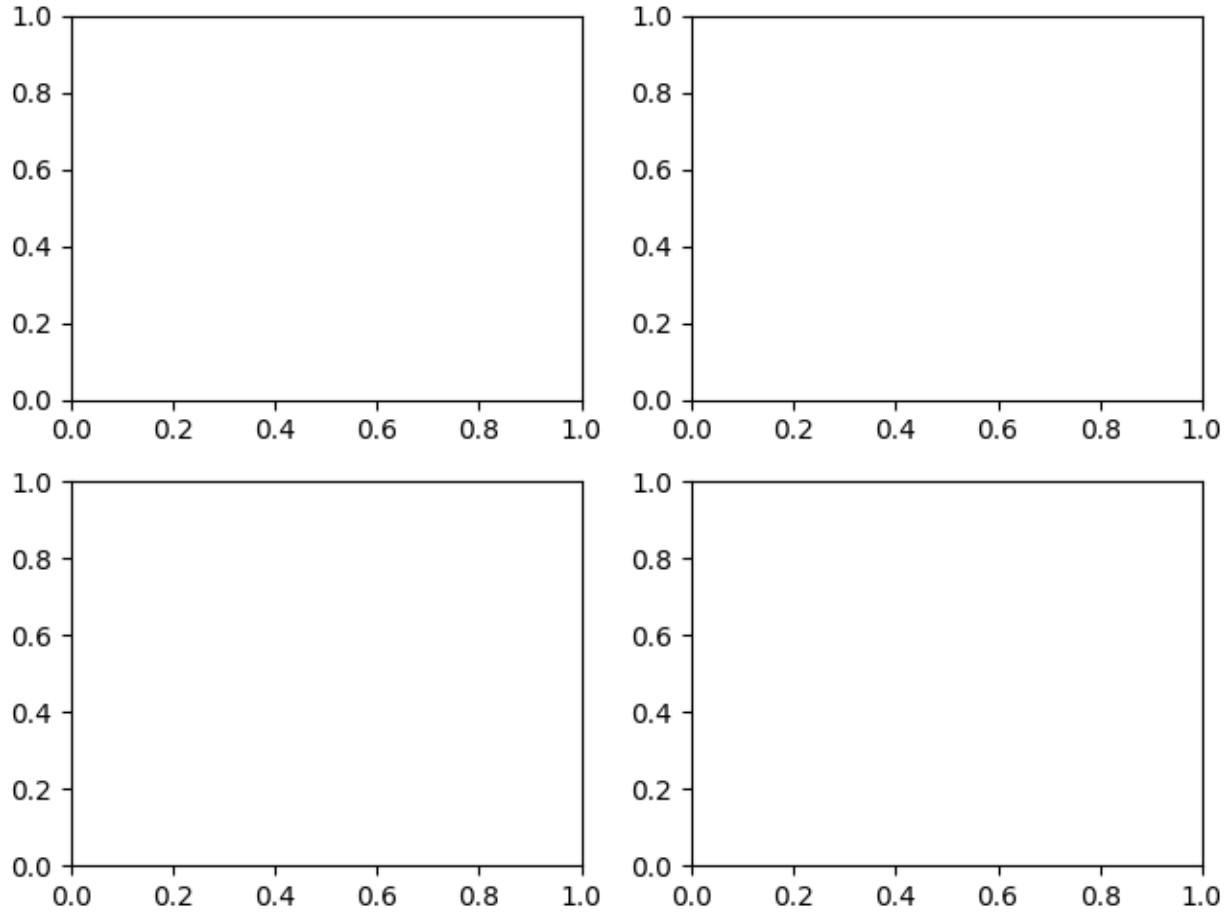
```
import matplotlib
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec
```

Basic Quickstart Guide

These first two examples show how to create a basic 2-by-2 grid using both *subplots()* and *gridspec*.

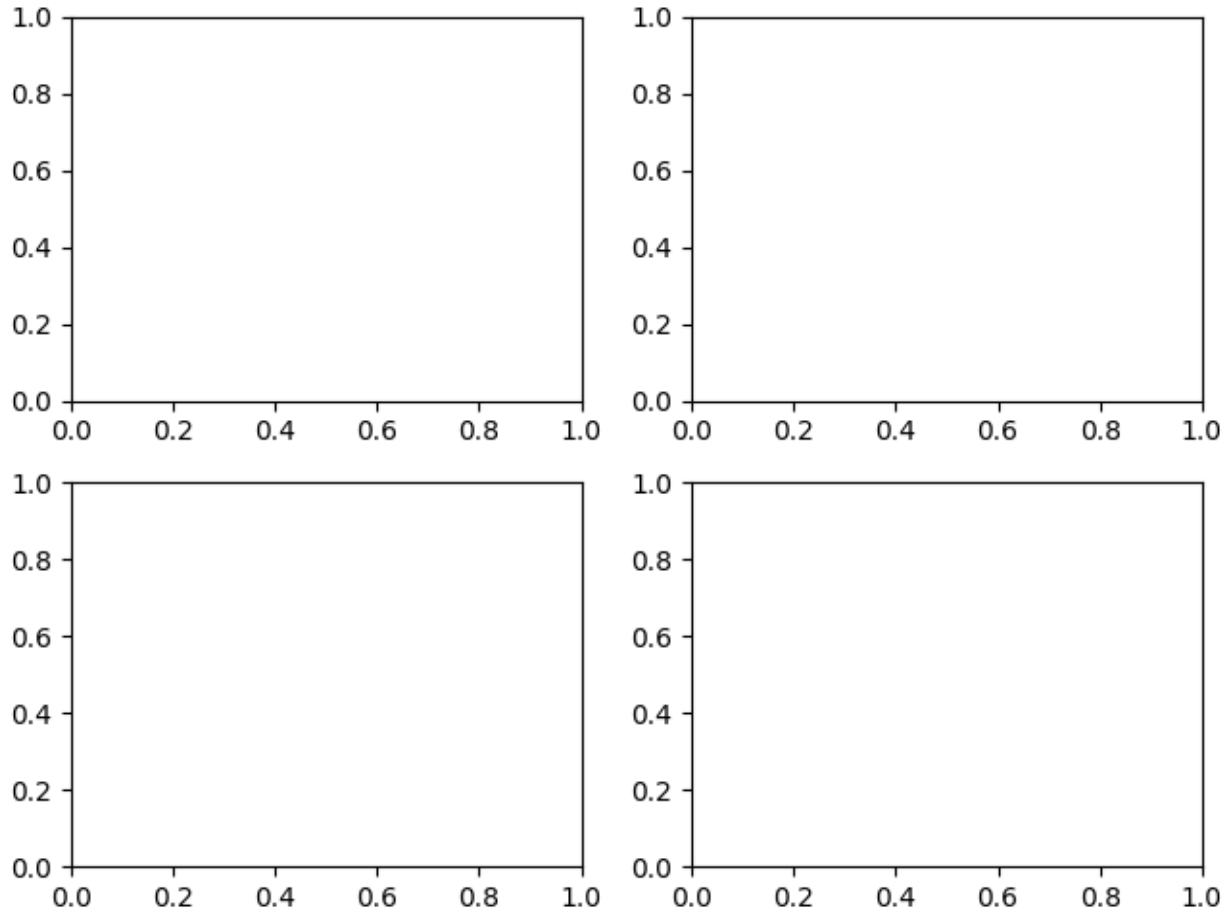
Using *subplots()* is quite simple. It returns a *Figure* instance and an array of *Axes* objects.

```
fig1, f1_axes = plt.subplots(ncols=2, nrows=2, constrained_layout=True)
```



For a simple use case such as this, *gridspec* is perhaps overly verbose. You have to create the figure and *GridSpec* instance separately, then pass elements of gridspec instance to the *add_subplot()* method to create the axes objects. The elements of the gridspec are accessed in generally the same manner as numpy arrays.

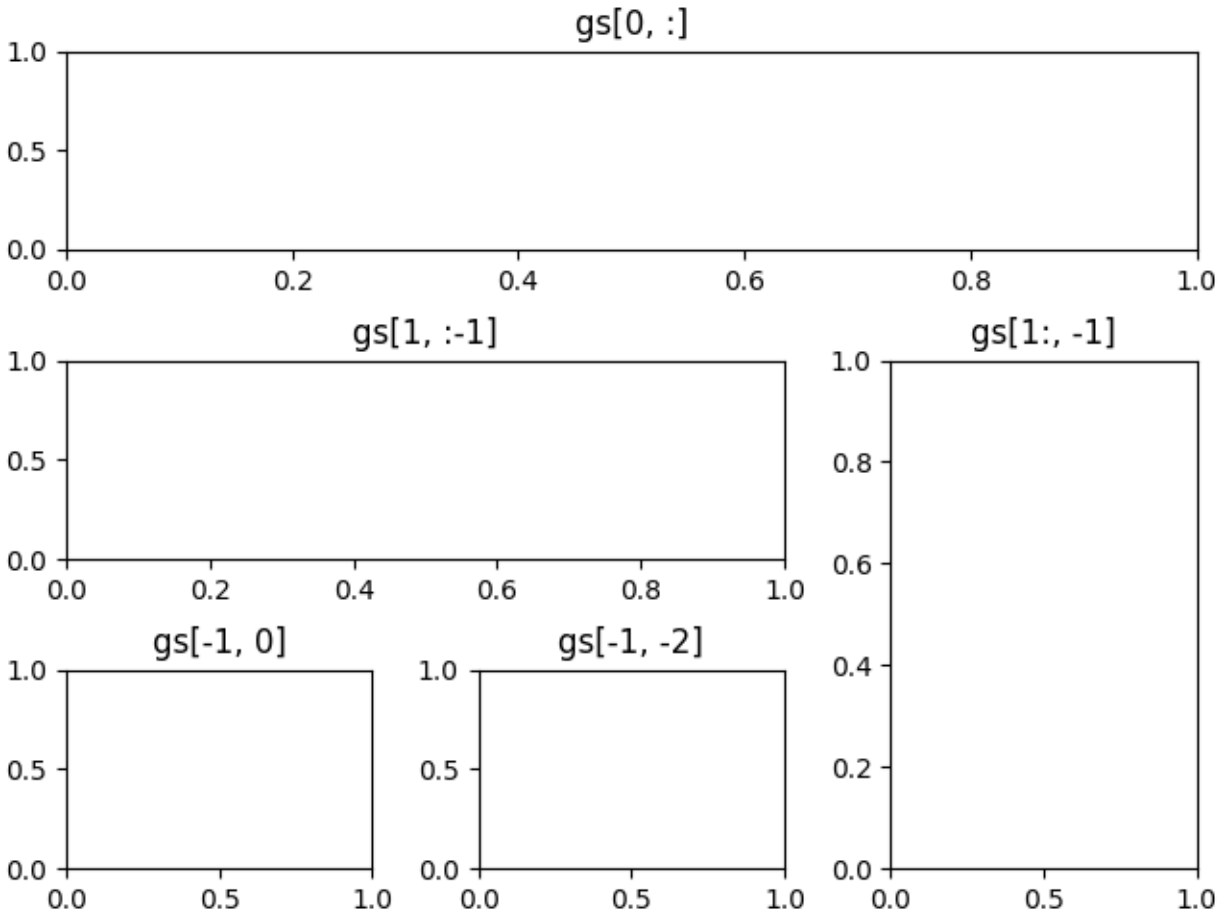
```
fig2 = plt.figure(constrained_layout=True)
spec2 = gridspec.GridSpec(ncols=2, nrows=2, figure=fig2)
f2_ax1 = fig2.add_subplot(spec2[0, 0])
f2_ax2 = fig2.add_subplot(spec2[0, 1])
f2_ax3 = fig2.add_subplot(spec2[1, 0])
f2_ax4 = fig2.add_subplot(spec2[1, 1])
```



The power of gridspec comes in being able to create subplots that span rows and columns. Note the NumPy slice syntax for selecting the part of the gridspec each subplot will occupy.

Note that we have also used the convenience method *Figure.add_gridspec* instead of *gridspec.GridSpec*, potentially saving the user an import, and keeping the namespace cleaner.

```
fig3 = plt.figure(constrained_layout=True)
gs = fig3.add_gridspec(3, 3)
f3_ax1 = fig3.add_subplot(gs[0, :])
f3_ax1.set_title('gs[0, :]')
f3_ax2 = fig3.add_subplot(gs[1, :-1])
f3_ax2.set_title('gs[1, :-1]')
f3_ax3 = fig3.add_subplot(gs[1:, -1])
f3_ax3.set_title('gs[1:, -1]')
f3_ax4 = fig3.add_subplot(gs[-1, 0])
f3_ax4.set_title('gs[-1, 0]')
f3_ax5 = fig3.add_subplot(gs[-1, -2])
f3_ax5.set_title('gs[-1, -2]')
```



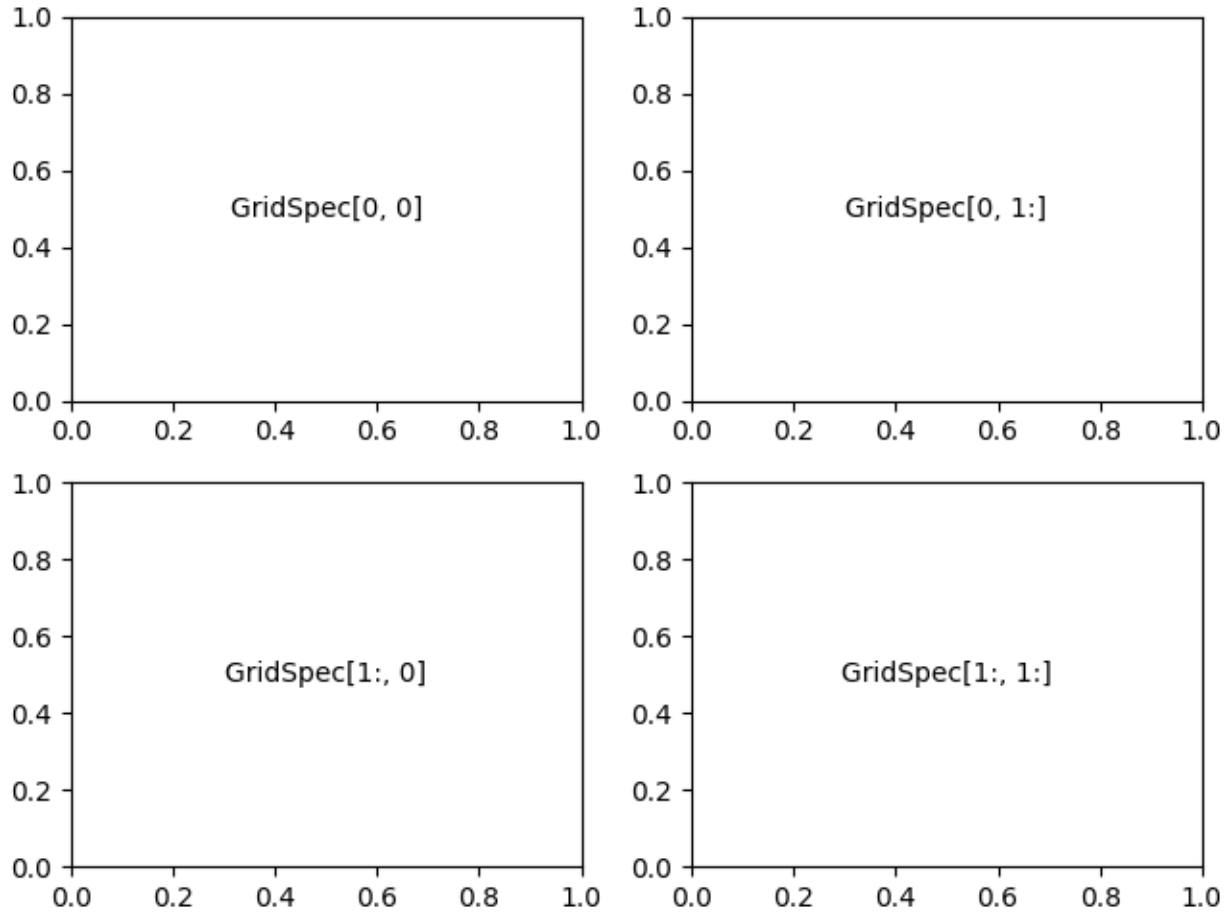
Out:

```
Text(0.5, 1.0, 'gs[-1, -2]')
```

gridspec is also indispensable for creating subplots of different widths via a couple of methods.

The method shown here is similar to the one above and initializes a uniform grid specification, and then uses numpy indexing and slices to allocate multiple "cells" for a given subplot.

```
fig4 = plt.figure(constrained_layout=True)
spec4 = fig4.add_gridspec(ncols=2, nrows=2)
anno_opts = dict(xy=(0.5, 0.5), xycoords='axes fraction',
                 va='center', ha='center')

f4_ax1 = fig4.add_subplot(spec4[0, 0])
f4_ax1.annotate('GridSpec[0, 0]', **anno_opts)
fig4.add_subplot(spec4[0, 1]).annotate('GridSpec[0, 1:]', **anno_opts)
fig4.add_subplot(spec4[1, 0]).annotate('GridSpec[1:, 0]', **anno_opts)
fig4.add_subplot(spec4[1, 1]).annotate('GridSpec[1:, 1:]', **anno_opts)
```

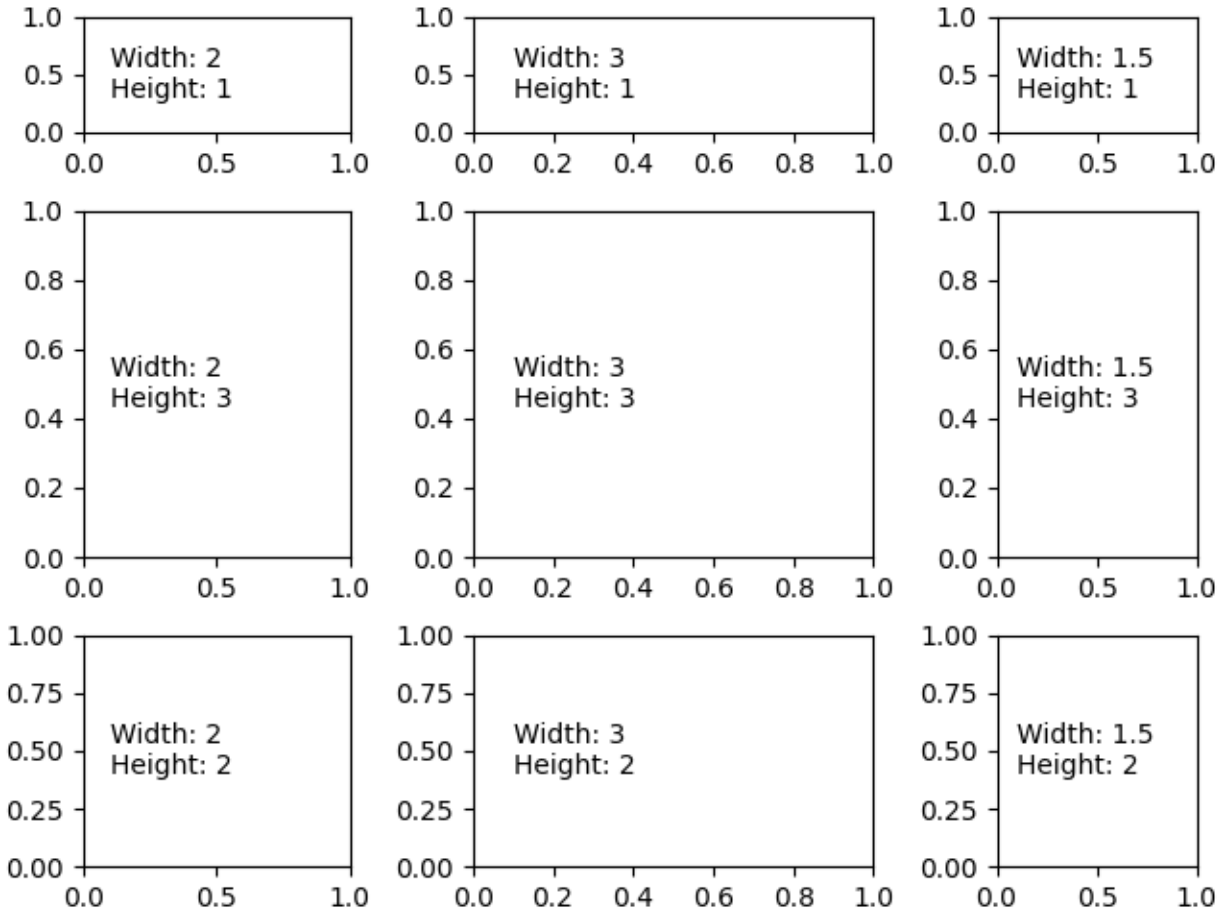


Out:

```
Text(0.5, 0.5, 'GridSpec[1:, 1:]')
```

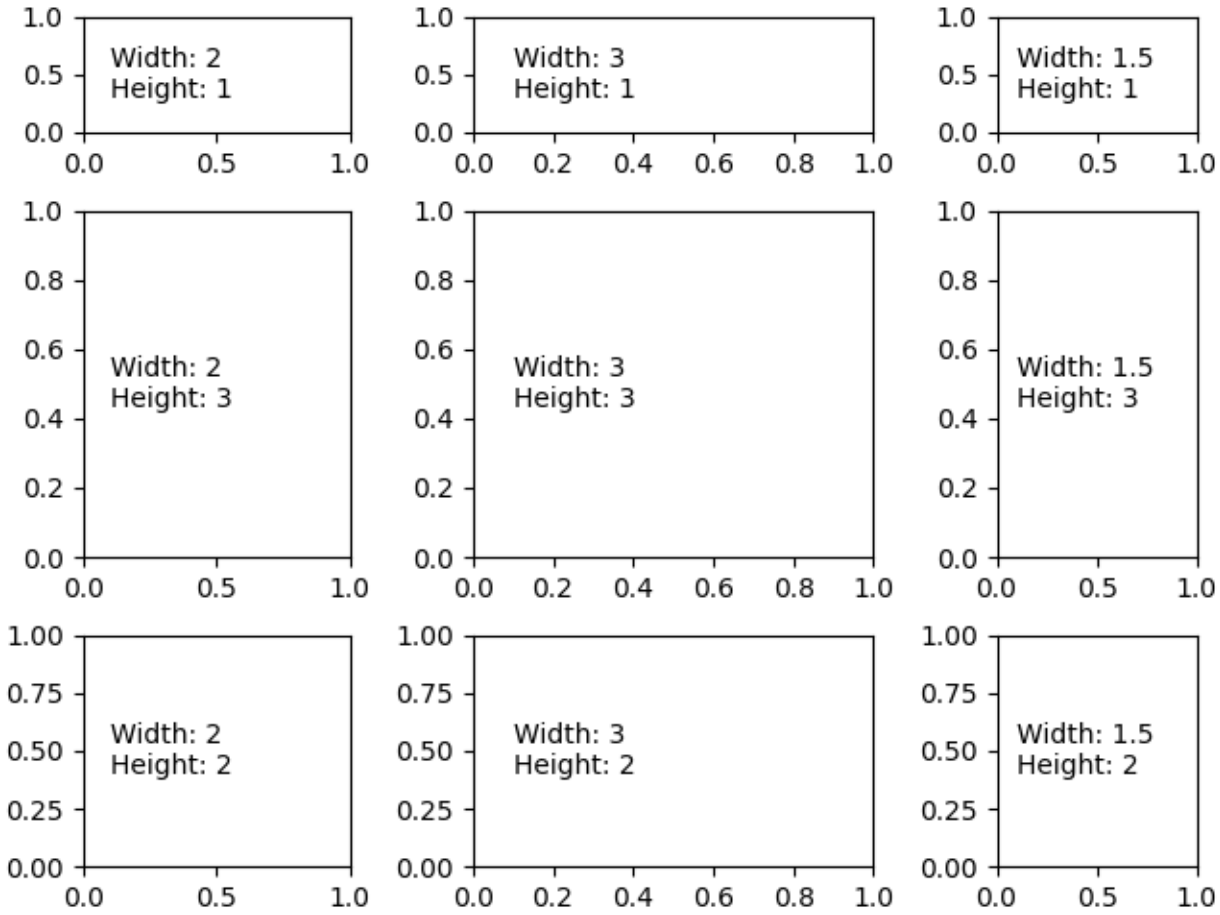
Another option is to use the `width_ratios` and `height_ratios` parameters. These keyword arguments are lists of numbers. Note that absolute values are meaningless, only their relative ratios matter. That means that `width_ratios=[2, 4, 8]` is equivalent to `width_ratios=[1, 2, 4]` within equally wide figures. For the sake of demonstration, we'll blindly create the axes within `for` loops since we won't need them later.

```
fig5 = plt.figure(constrained_layout=True)
widths = [2, 3, 1.5]
heights = [1, 3, 2]
spec5 = fig5.add_gridspec(ncols=3, nrows=3, width_ratios=widths,
                          height_ratios=heights)
for row in range(3):
    for col in range(3):
        ax = fig5.add_subplot(spec5[row, col])
        label = 'Width: {}\nHeight: {}'.format(widths[col], heights[row])
        ax.annotate(label, (0.1, 0.5), xycoords='axes fraction', va='center')
```



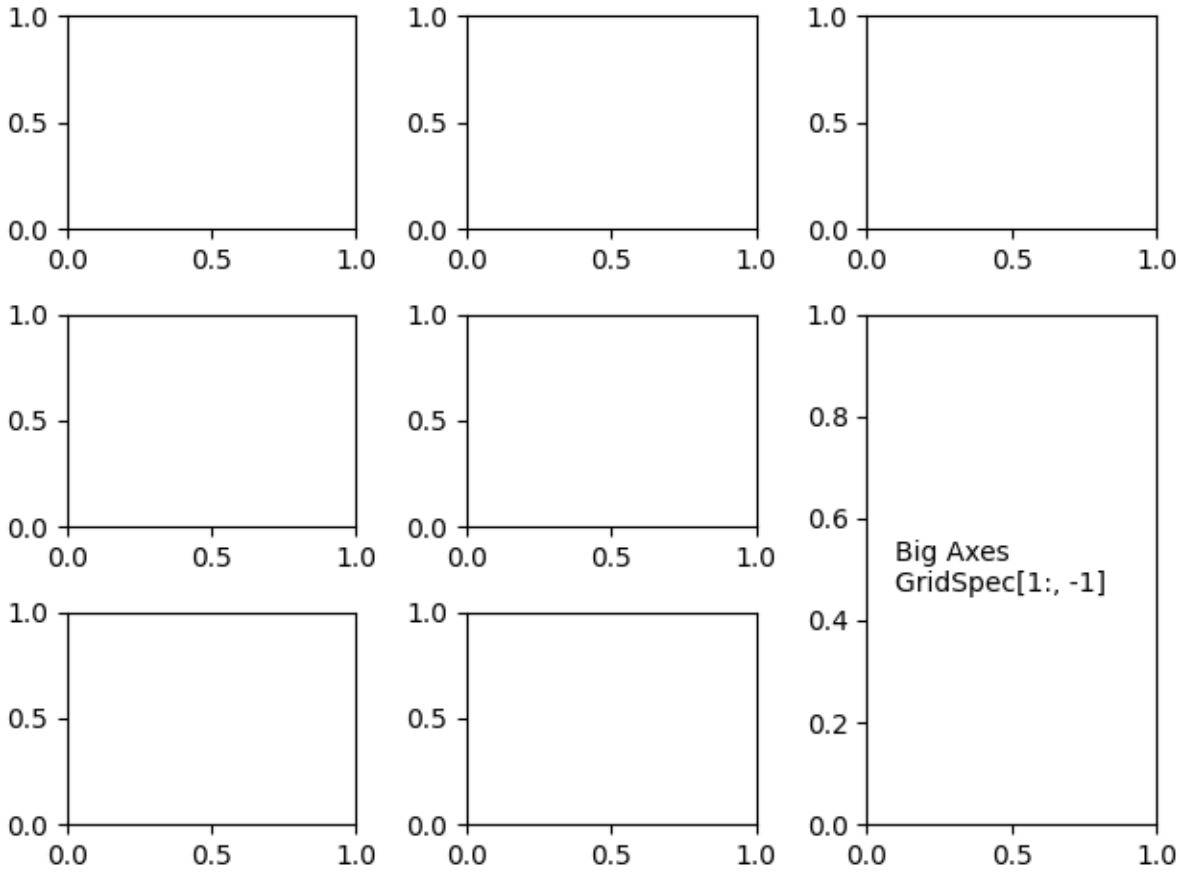
Learning to use `width_ratios` and `height_ratios` is particularly useful since the top-level function *subplots()* accepts them within the `gridspec_kw` parameter. For that matter, any parameter accepted by *GridSpec* can be passed to *subplots()* via the `gridspec_kw` parameter. This example recreates the previous figure without directly using a gridspec instance.

```
gs_kw = dict(width_ratios=widths, height_ratios=heights)
fig6, f6_axes = plt.subplots(ncols=3, nrows=3, constrained_layout=True,
                             gridspec_kw=gs_kw)
for r, row in enumerate(f6_axes):
    for c, ax in enumerate(row):
        label = 'Width: {}\nHeight: {}'.format(widths[c], heights[r])
        ax.annotate(label, (0.1, 0.5), xycoords='axes fraction', va='center')
```



The `subplots` and `get_gridspec` methods can be combined since it is sometimes more convenient to make most of the subplots using `subplots` and then remove some and combine them. Here we create a layout with the bottom two axes in the last column combined.

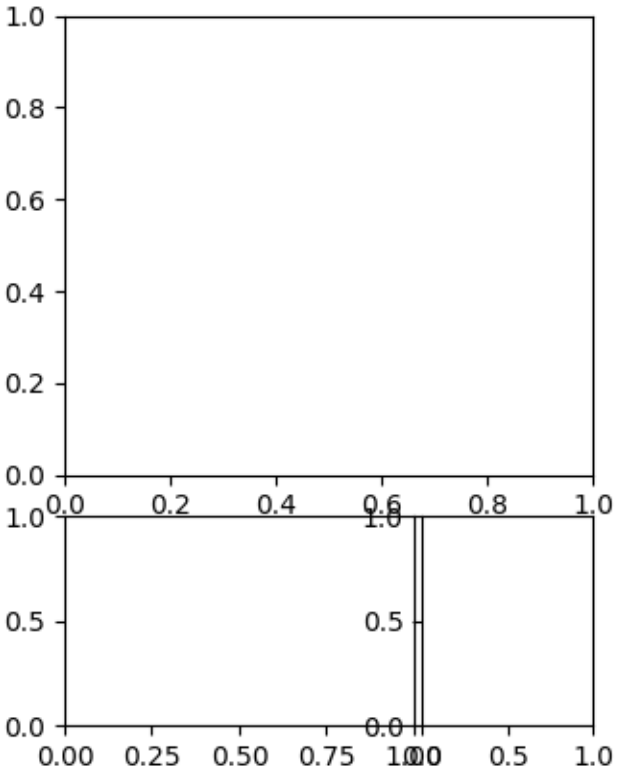
```
fig7, f7_axs = plt.subplots(ncols=3, nrows=3)
gs = f7_axs[1, 2].get_gridspec()
# remove the underlying axes
for ax in f7_axs[1:, -1]:
    ax.remove()
axbig = fig7.add_subplot(gs[1:, -1])
axbig.annotate('Big Axes \nGridSpec[1:, -1]', (0.1, 0.5),
               xycoords='axes fraction', va='center')

fig7.tight_layout()
```



Fine Adjustments to a Gridspec Layout

When a GridSpec is explicitly used, you can adjust the layout parameters of subplots that are created from the GridSpec. Note this option is not compatible with `constrained_layout` or *Figure.tight_layout* which both adjust subplot sizes to fill the figure.

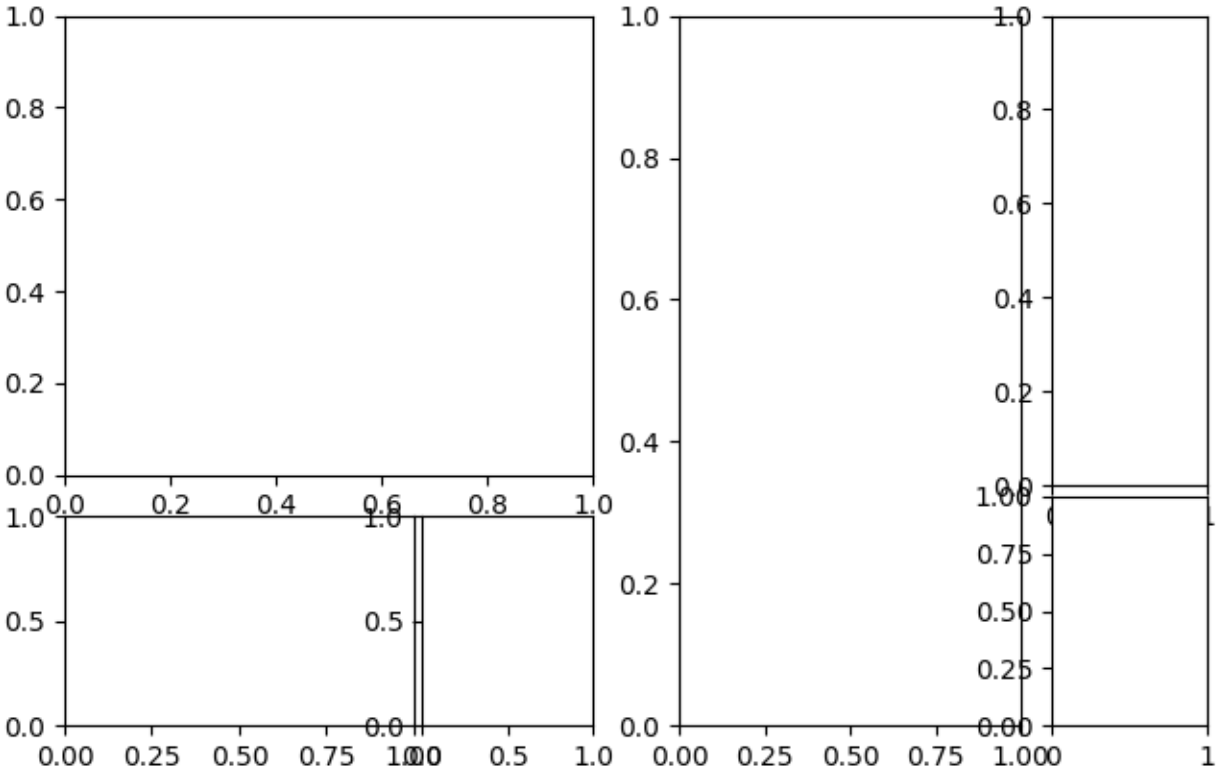
```
fig8 = plt.figure(constrained_layout=False)
gs1 = fig8.add_gridspec(nrows=3, ncols=3, left=0.05, right=0.48, wspace=0.05)
f8_ax1 = fig8.add_subplot(gs1[:-1, :])
f8_ax2 = fig8.add_subplot(gs1[-1, :-1])
f8_ax3 = fig8.add_subplot(gs1[-1, -1])
```



This is similar to *subplots_adjust()*, but it only affects the subplots that are created from the given GridSpec.

For example, compare the left and right sides of this figure:

```
fig9 = plt.figure(constrained_layout=False)
gs1 = fig9.add_gridspec(nrows=3, ncols=3, left=0.05, right=0.48,
                        wspace=0.05)
f9_ax1 = fig9.add_subplot(gs1[:-1, :])
f9_ax2 = fig9.add_subplot(gs1[-1, :-1])
f9_ax3 = fig9.add_subplot(gs1[-1, -1])

gs2 = fig9.add_gridspec(nrows=3, ncols=3, left=0.55, right=0.98,
                        hspace=0.05)
f9_ax4 = fig9.add_subplot(gs2[:, :-1])
f9_ax5 = fig9.add_subplot(gs2[:-1, -1])
f9_ax6 = fig9.add_subplot(gs2[-1, -1])
```

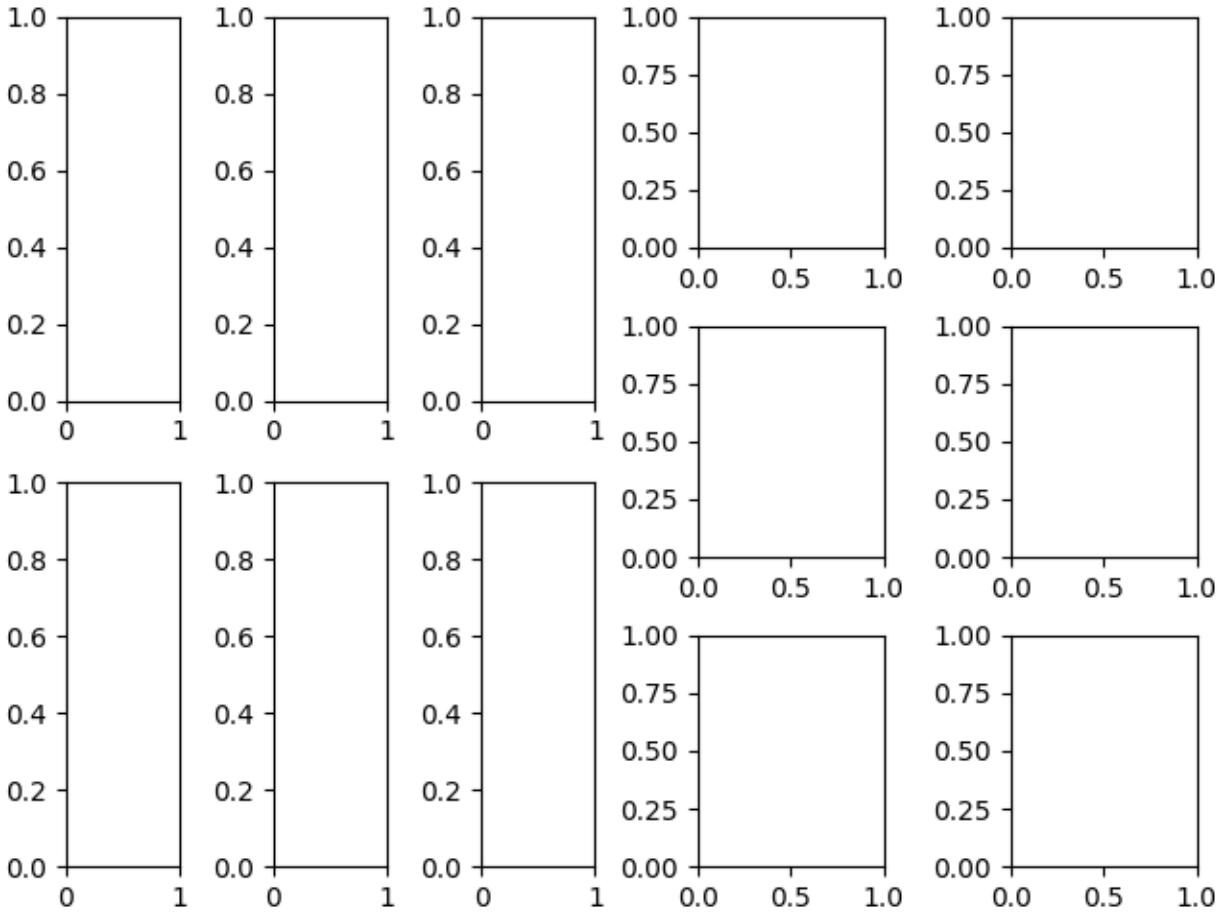


GridSpec using SubplotSpec

You can create GridSpec from the *SubplotSpec*, in which case its layout parameters are set to that of the location of the given SubplotSpec.

Note this is also available from the more verbose *gridspec.GridSpecFromSubplotSpec*.

```
fig10 = plt.figure(constrained_layout=True)
gs0 = fig10.add_gridspec(1, 2)

gs00 = gs0[0].subgridspec(2, 3)
gs01 = gs0[1].subgridspec(3, 2)

for a in range(2):
    for b in range(3):
        fig10.add_subplot(gs00[a, b])
        fig10.add_subplot(gs01[b, a])
```



A Complex Nested GridSpec using SubplotSpec

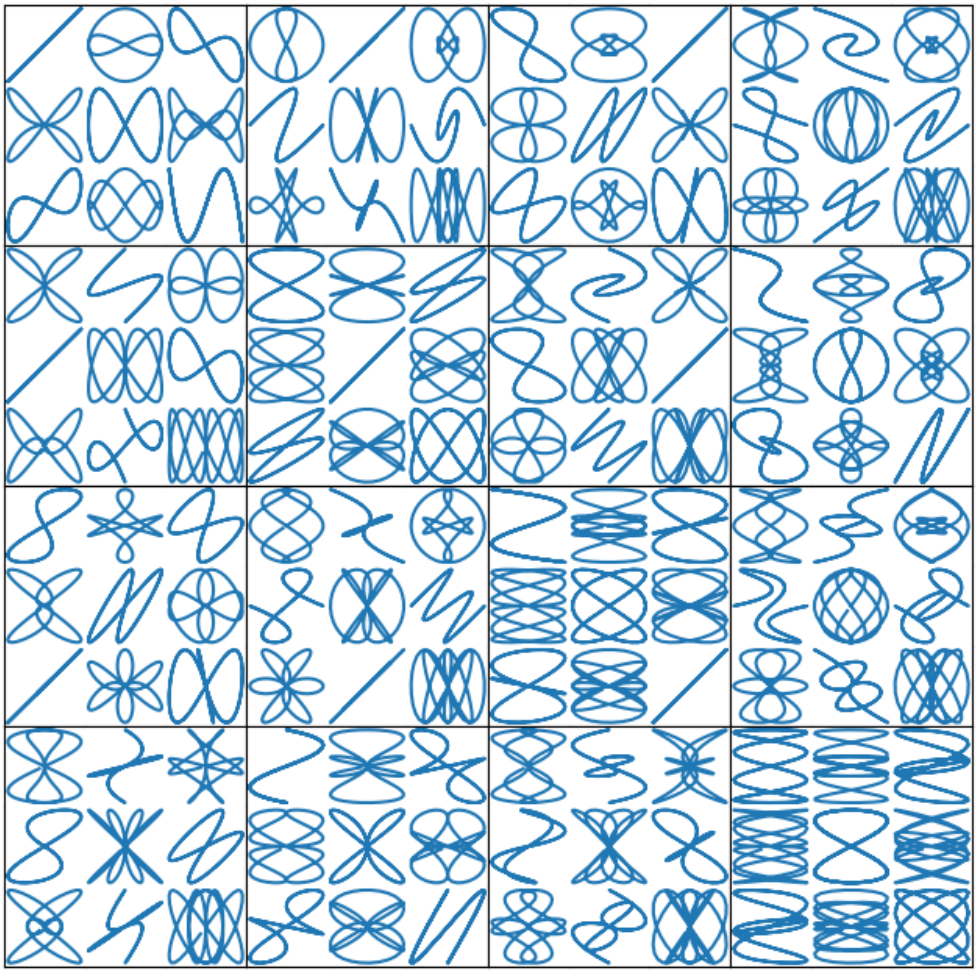
Here's a more sophisticated example of nested GridSpec where we put a box around each cell of the outer 4x4 grid, by hiding appropriate spines in each of the inner 3x3 grids.

```
import numpy as np


def squiggle_xy(a, b, c, d, i=np.arange(0.0, 2*np.pi, 0.05)):
    return np.sin(i*a)*np.cos(i*b), np.sin(i*c)*np.cos(i*d)


fig11 = plt.figure(figsize=(8, 8), constrained_layout=False)
outer_grid = fig11.add_gridspec(4, 4, wspace=0, hspace=0)

for a in range(4):
    for b in range(4):
        # gridspec inside gridspec
        inner_grid = outer_grid[a, b].subgridspec(3, 3, wspace=0, hspace=0)
        axs = inner_grid.subplots()  # Create all subplots for the inner grid.
```

(continues on next page)

(continued from previous page)

```
        for (c, d), ax in np.ndenumerate(axs):
            ax.plot(*squiggle_xy(a + 1, b + 1, c + 1, d + 1))
            ax.set(xticks=[], yticks=[])

# show only the outside spines
for ax in fig11.get_axes():
    ax.spines['top'].set_visible(ax.is_first_row())
    ax.spines['bottom'].set_visible(ax.is_last_row())
    ax.spines['left'].set_visible(ax.is_first_col())
    ax.spines['right'].set_visible(ax.is_last_col())

plt.show()
```



References

The usage of the following functions and methods is shown in this example:

```
matplotlib.pyplot.subplots
matplotlib.figure.Figure.add_gridspec
matplotlib.figure.Figure.add_subplot
matplotlib.gridspec.GridSpec
matplotlib.gridspec.SubplotSpec.subgridspec
matplotlib.gridspec.GridSpecFromSubplotSpec
```

Total running time of the script: (0 minutes 7.312 seconds)

2.2.5 Constrained Layout Guide

How to use constrained-layout to fit plots within your figure cleanly.

constrained_layout automatically adjusts subplots and decorations like legends and colorbars so that they fit in the figure window while still preserving, as best they can, the logical layout requested by the user.

constrained_layout is similar to *tight_layout*, but uses a constraint solver to determine the size of axes that allows them to fit.

constrained_layout needs to be activated before any axes are added to a figure. Two ways of doing so are

- using the respective argument to *subplots()* or *figure()*, e.g.:

```
plt.subplots(constrained_layout=True)
```

- activate it via *rcParams*, like:

```
plt.rcParams['figure.constrained_layout.use'] = True
```

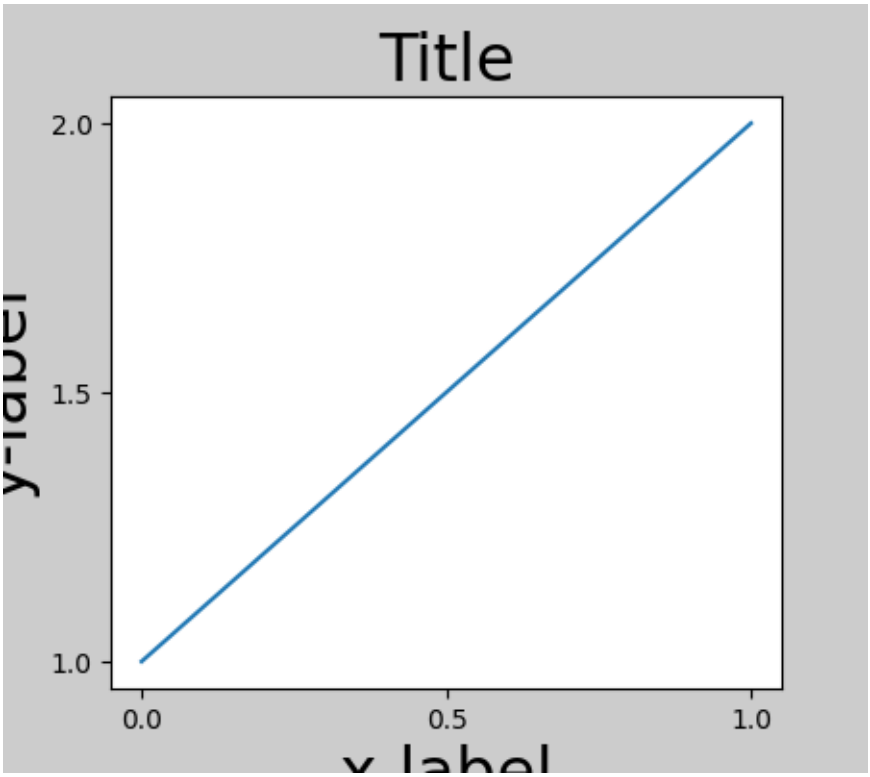
Those are described in detail throughout the following sections.

Warning: Currently Constrained Layout is **experimental**. The behaviour and API are subject to change, or the whole functionality may be removed without a deprecation period. If you *require* your plots to be absolutely reproducible, get the Axes positions after running Constrained Layout and use `ax.set_position()` in your code with `constrained_layout=False`.

Simple Example

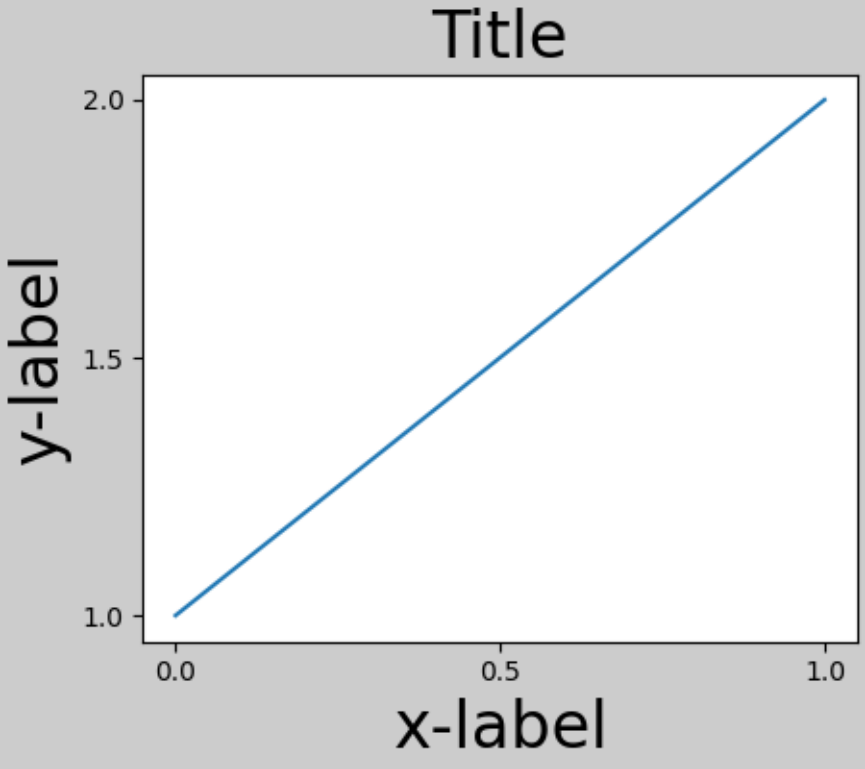
In Matplotlib, the location of axes (including subplots) are specified in normalized figure coordinates. It can happen that your axis labels or titles (or sometimes even ticklabels) go outside the figure area, and are thus clipped.

```
import matplotlib.pyplot as plt
import matplotlib.colors as mcolors
import matplotlib.gridspec as gridspec
import numpy as np


plt.rcParams['savefig.facecolor'] = "0.8"
plt.rcParams['figure.figsize'] = 4.5, 4.
plt.rcParams['figure.max_open_warning'] = 50


def example_plot(ax, fontsize=12, hide_labels=False):
    ax.plot([1, 2])

    ax.locator_params(nbins=3)
    if hide_labels:
        ax.set_xticklabels([])
        ax.set_yticklabels([])
    else:
        ax.set_xlabel('x-label', fontsize=fontsize)
        ax.set_ylabel('y-label', fontsize=fontsize)
        ax.set_title('Title', fontsize=fontsize)


fig, ax = plt.subplots(constrained_layout=False)
example_plot(ax, fontsize=24)
```



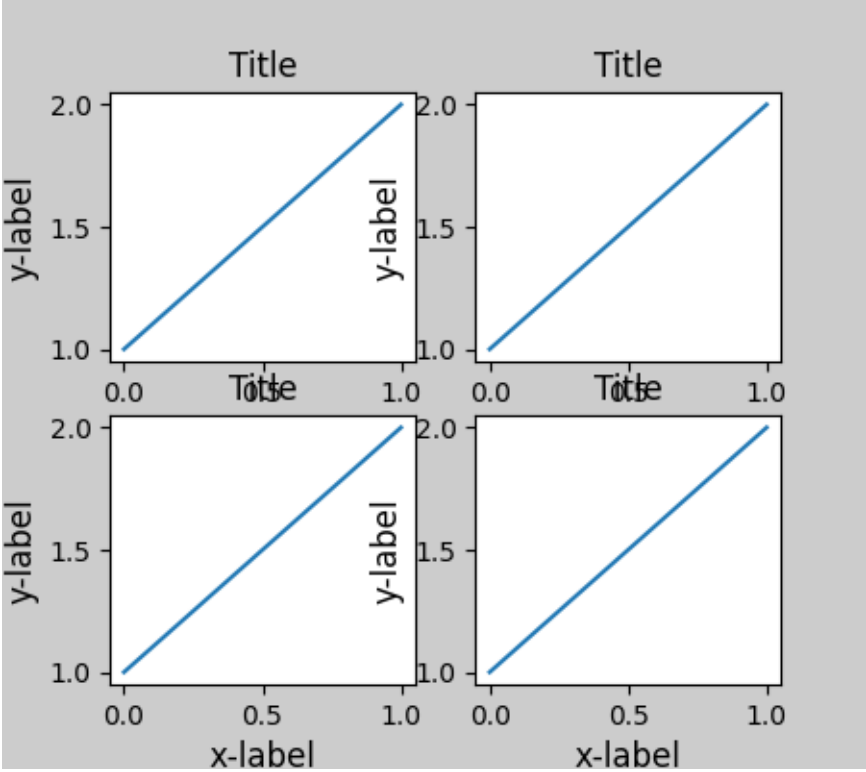
To prevent this, the location of axes needs to be adjusted. For subplots, this can be done by adjusting the subplot params (*Move the edge of an axes to make room for tick labels*). However, specifying your figure with the `constrained_layout=True` kwarg will do the adjusting automatically.

```
fig, ax = plt.subplots(constrained_layout=True)
example_plot(ax, fontsize=24)
```



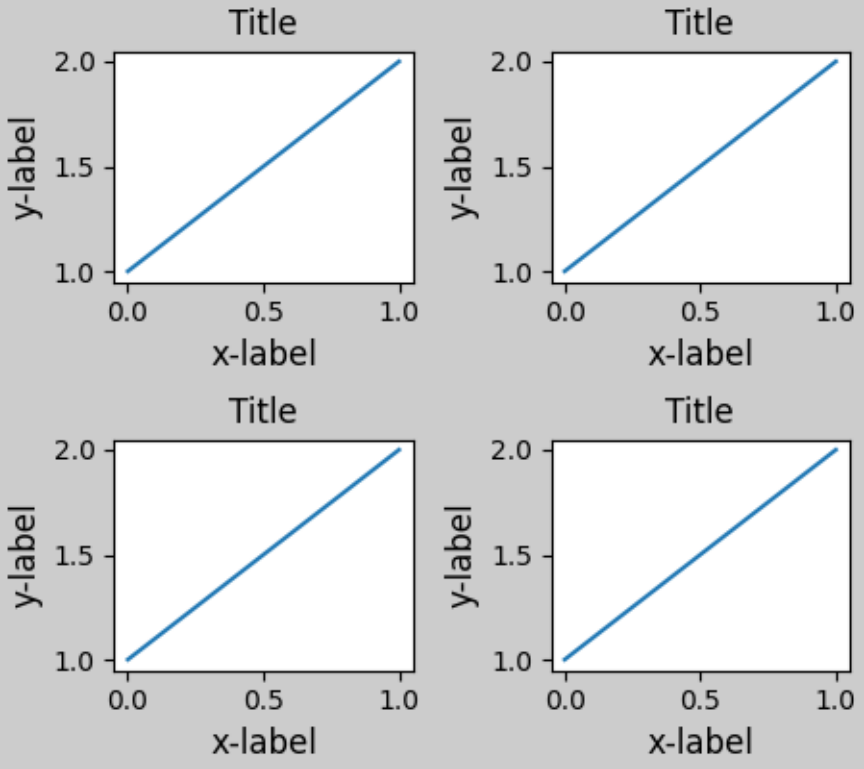
When you have multiple subplots, often you see labels of different axes overlapping each other.

```
fig, axs = plt.subplots(2, 2, constrained_layout=False)
for ax in axs.flat:
    example_plot(ax)
```



Specifying `constrained_layout=True` in the call to `plt.subplots` causes the layout to be properly constrained.

```
fig, axs = plt.subplots(2, 2, constrained_layout=True)
for ax in axs.flat:
    example_plot(ax)
```

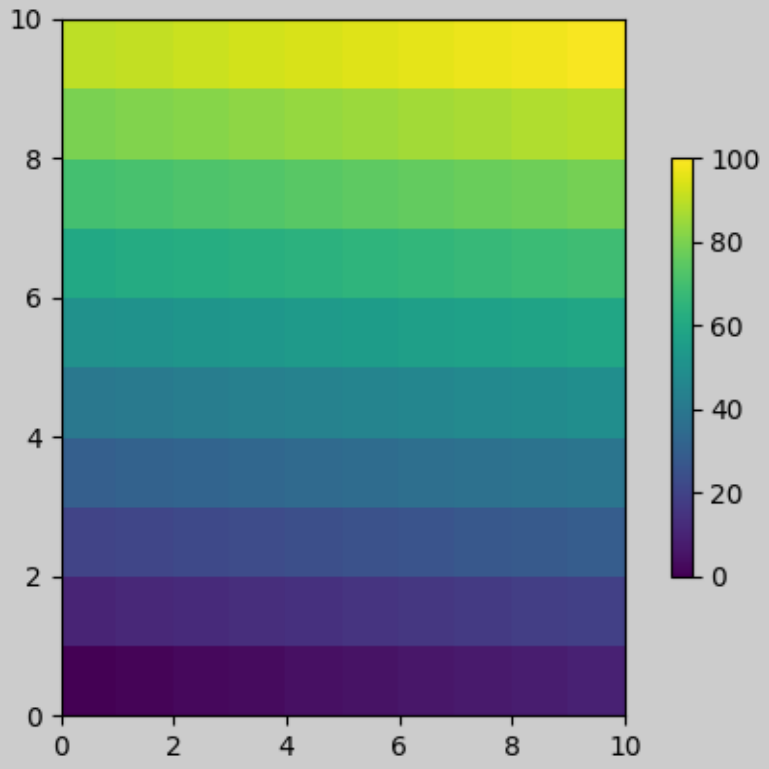


Colorbars

If you create a colorbar with *Figure.colorbar*, you need to make room for it. `constrained_layout` does this automatically. Note that if you specify `use_gridspec=True` it will be ignored because this option is made for improving the layout via `tight_layout`.

Note: For the *pcolormesh* kwargs (`pc_kwargs`) we use a dictionary. Below we will assign one colorbar to a number of axes each containing a *ScalarMappable*; specifying the norm and colormap ensures the colorbar is accurate for all the axes.

```
arr = np.arange(100).reshape((10, 10))
norm = mcolors.Normalize(vmin=0., vmax=100.)
# see note above: this makes all pcolormesh calls consistent:
pc_kwargs = {'rasterized': True, 'cmap': 'viridis', 'norm': norm}
fig, ax = plt.subplots(figsize=(4, 4), constrained_layout=True)
im = ax.pcolormesh(arr, **pc_kwargs)
fig.colorbar(im, ax=ax, shrink=0.6)
```

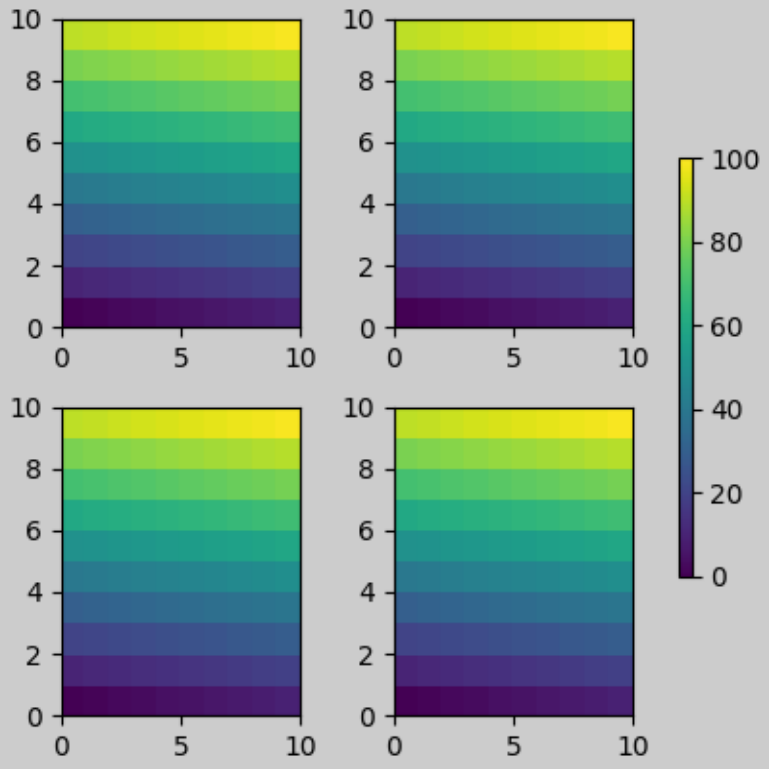


Out:

```
<matplotlib.colorbar.Colorbar object at 0x7f53fac9aa00>
```

If you specify a list of axes (or other iterable container) to the `ax` argument of `colorbar`, constrained_layout will take space from the specified axes.

```
fig, axs = plt.subplots(2, 2, figsize=(4, 4), constrained_layout=True)
for ax in axs.flat:
    im = ax.pcolormesh(arr, **pc_kwargs)
fig.colorbar(im, ax=axs, shrink=0.6)
```

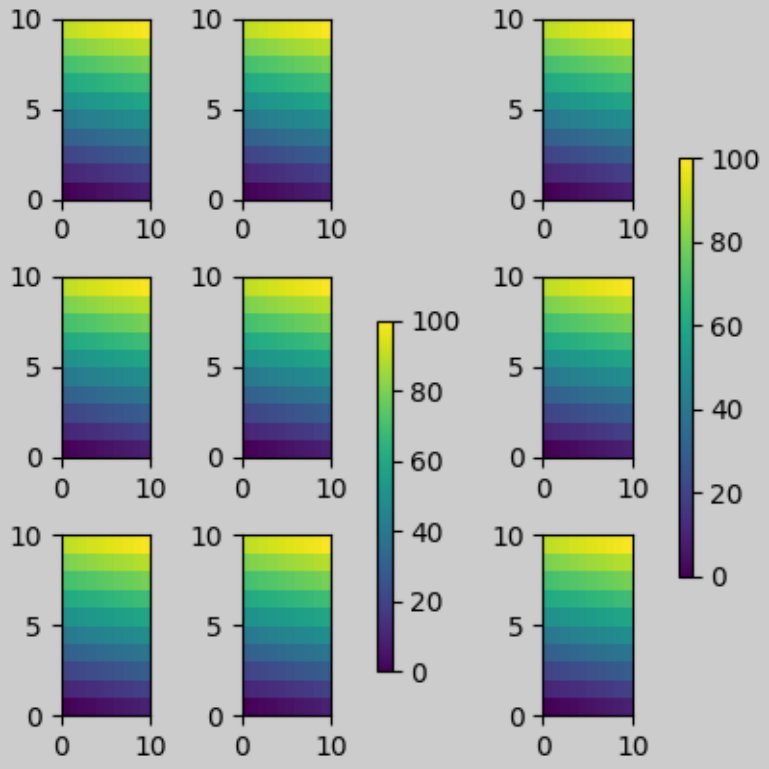


Out:

```
<matplotlib.colorbar.Colorbar object at 0x7f53fedd3c10>
```

If you specify a list of axes from inside a grid of axes, the colorbar will steal space appropriately, and leave a gap, but all subplots will still be the same size.

```
fig, axs = plt.subplots(3, 3, figsize=(4, 4), constrained_layout=True)
for ax in axs.flat:
    im = ax.pcolormesh(arr, **pc_kwargs)
fig.colorbar(im, ax=axs[1:, ][:, 1], shrink=0.8)
fig.colorbar(im, ax=axs[:, -1], shrink=0.6)
```

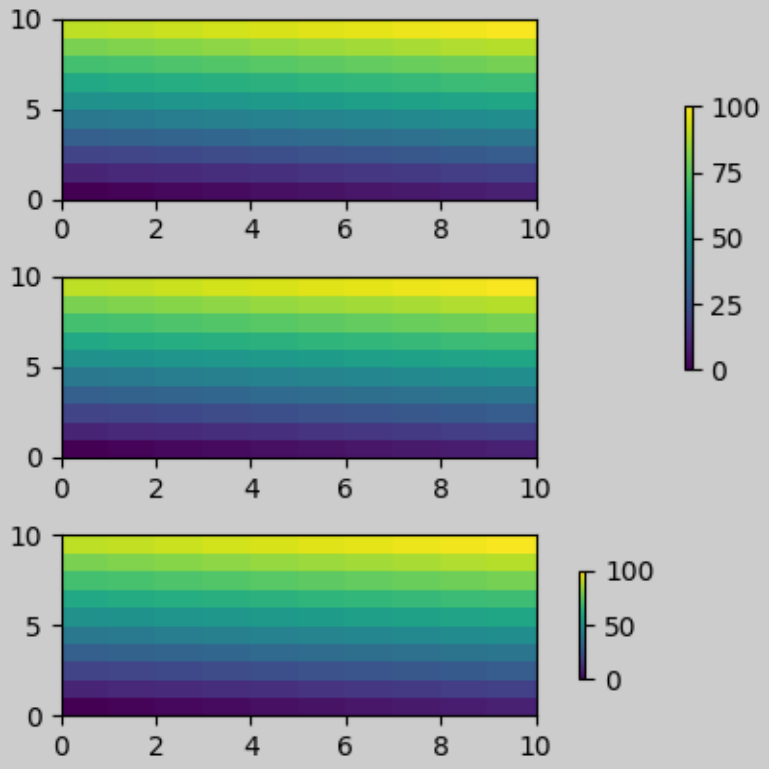


Out:

```
<matplotlib.colorbar.Colorbar object at 0x7f53fe8152b0>
```

Note that there is a bit of a subtlety when specifying a single axes as the parent. In the following, it might be desirable and expected for the colorbars to line up, but they don't because the colorbar paired with the bottom axes is tied to the subplotspec of the axes, and hence shrinks when the gridspec-level colorbar is added.

```
fig, axs = plt.subplots(3, 1, figsize=(4, 4), constrained_layout=True)
for ax in axs[:2]:
    im = ax.pcolormesh(arr, **pc_kwargs)
fig.colorbar(im, ax=axs[:2], shrink=0.6)
im = axs[2].pcolormesh(arr, **pc_kwargs)
fig.colorbar(im, ax=axs[2], shrink=0.6)
```

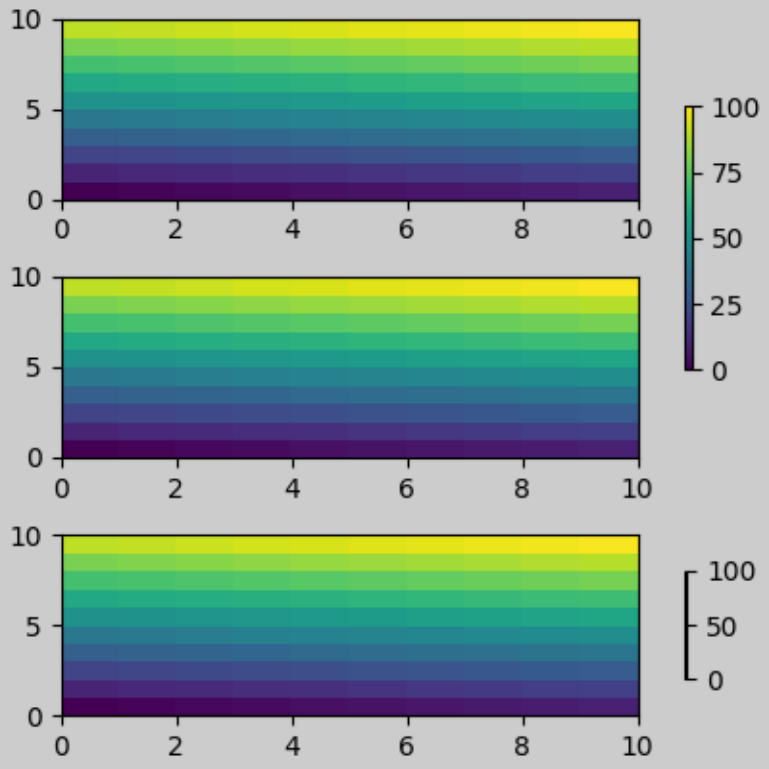


Out:

```
<matplotlib.colorbar.Colorbar object at 0x7f540010d100>
```

The API to make a single-axes behave like a list of axes is to specify it as a list (or other iterable container), as below:

```
fig, axs = plt.subplots(3, 1, figsize=(4, 4), constrained_layout=True)
for ax in axs[:2]:
    im = ax.pcolormesh(arr, **pc_kwargs)
fig.colorbar(im, ax=axs[:2], shrink=0.6)
im = axs[2].pcolormesh(arr, **pc_kwargs)
fig.colorbar(im, ax=[axs[2]], shrink=0.6)
```



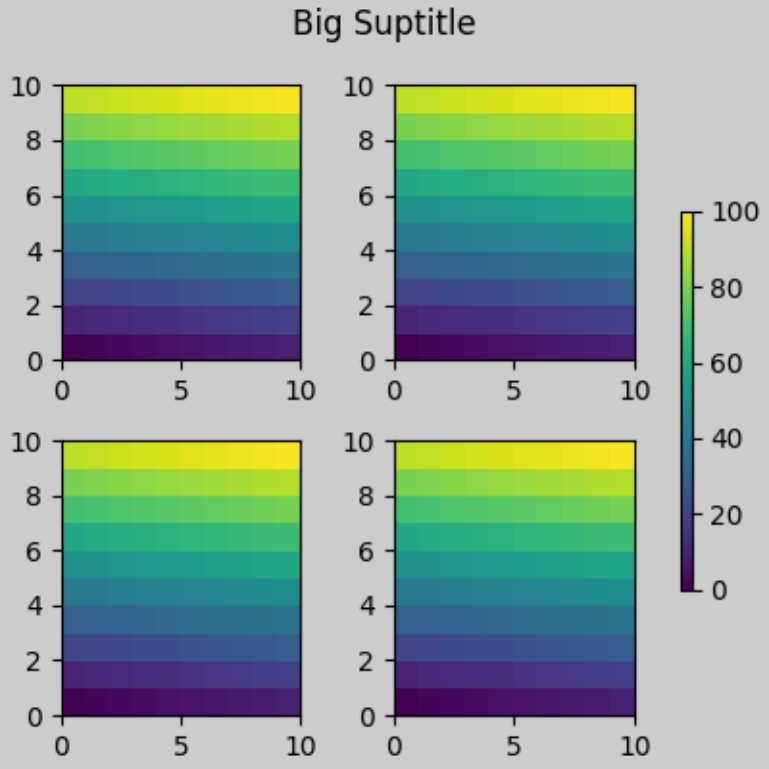
Out:

```
<matplotlib.colorbar.Colorbar object at 0x7f53f9b49e20>
```

Suptitle

`constrained_layout` can also make room for *`suptitle`*.

```
fig, axs = plt.subplots(2, 2, figsize=(4, 4), constrained_layout=True)
for ax in axs.flat:
    im = ax.pcolormesh(arr, **pc_kwargs)
fig.colorbar(im, ax=axs, shrink=0.6)
fig.suptitle('Big Suptitle')
```



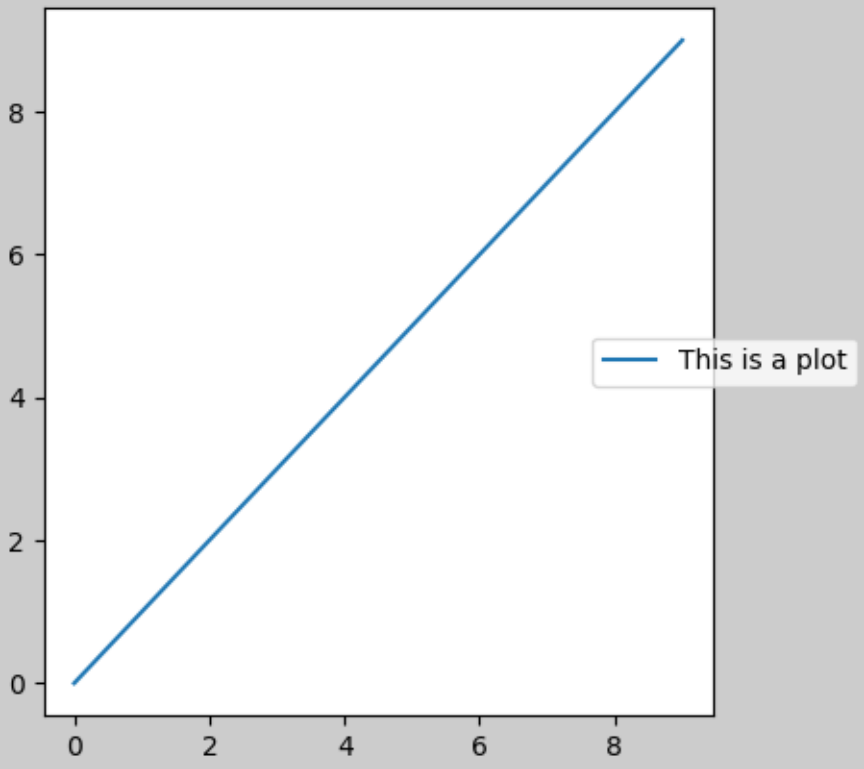
Out:

```
Text(0.5, 0.98, 'Big Suptitle')
```

Legends

Legends can be placed outside of their parent axis. Constrained-layout is designed to handle this for *Axes.legend()*. However, constrained-layout does *not* handle legends being created via *Figure.legend()* (yet).

```
fig, ax = plt.subplots(constrained_layout=True)
ax.plot(np.arange(10), label='This is a plot')
ax.legend(loc='center left', bbox_to_anchor=(0.8, 0.5))
```

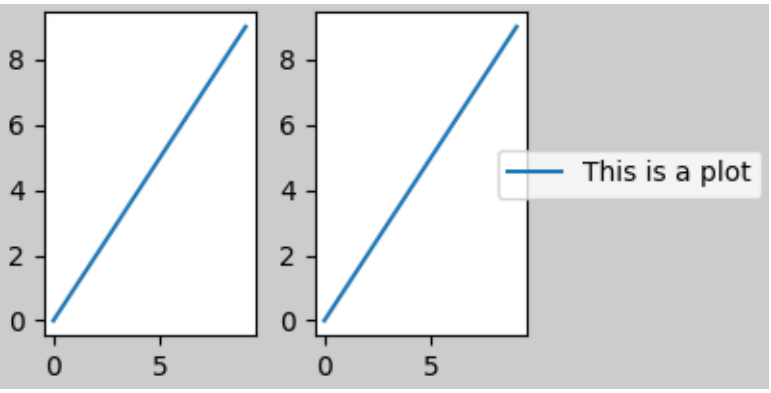


Out:

```
<matplotlib.legend.Legend object at 0x7f53f9b037c0>
```

However, this will steal space from a subplot layout:

```
fig, axs = plt.subplots(1, 2, figsize=(4, 2), constrained_layout=True)
axs[0].plot(np.arange(10))
axs[1].plot(np.arange(10), label='This is a plot')
axs[1].legend(loc='center left', bbox_to_anchor=(0.8, 0.5))
```



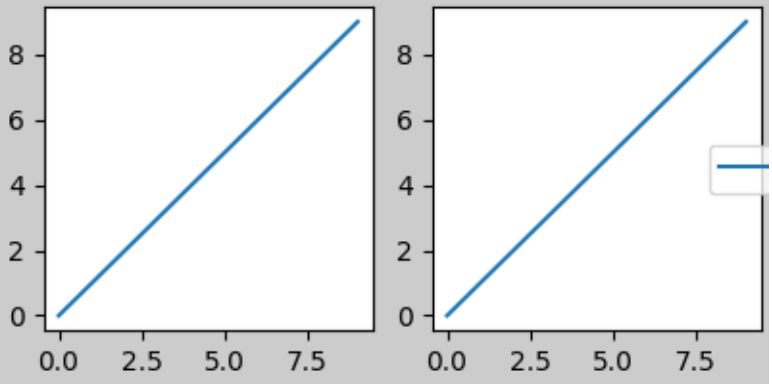
Out:

```
<matplotlib.legend.Legend object at 0x7f53f9a93b80>
```

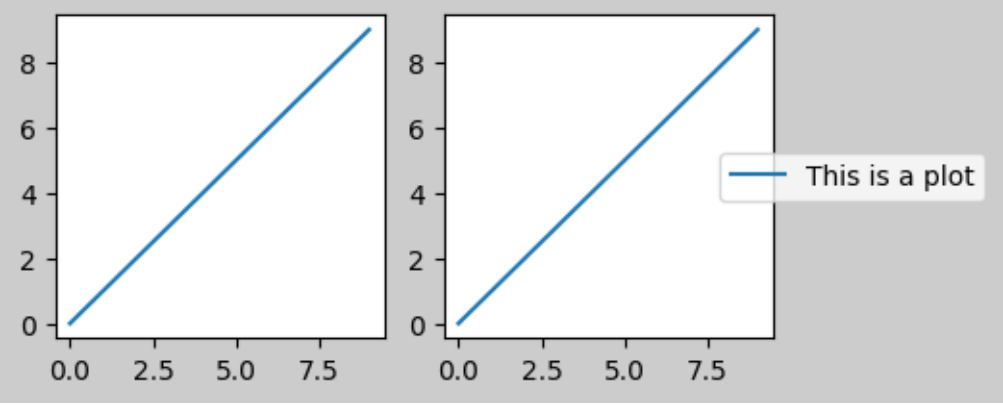
In order for a legend or other artist to *not* steal space from the subplot layout, we

can `leg.set_in_layout(False)`. Of course this can mean the legend ends up cropped, but can be useful if the plot is subsequently called with `fig.savefig('outname.png', bbox_inches='tight')`. Note, however, that the legend's `get_in_layout` status will have to be toggled again to make the saved file work, and we must manually trigger a draw if we want constrained_layout to adjust the size of the axes before printing.

```
fig, axs = plt.subplots(1, 2, figsize=(4, 2), constrained_layout=True)

axs[0].plot(np.arange(10))
axs[1].plot(np.arange(10), label='This is a plot')
leg = axs[1].legend(loc='center left', bbox_to_anchor=(0.8, 0.5))
leg.set_in_layout(False)
# trigger a draw so that constrained_layout is executed once
# before we turn it off when printing....
fig.canvas.draw()
# we want the legend included in the bbox_inches='tight' calcs.
leg.set_in_layout(True)
# we don't want the layout to change at this point.
fig.set_constrained_layout(False)
fig.savefig('../../doc/_static/constrained_layout_1b.png',
            bbox_inches='tight', dpi=100)
```

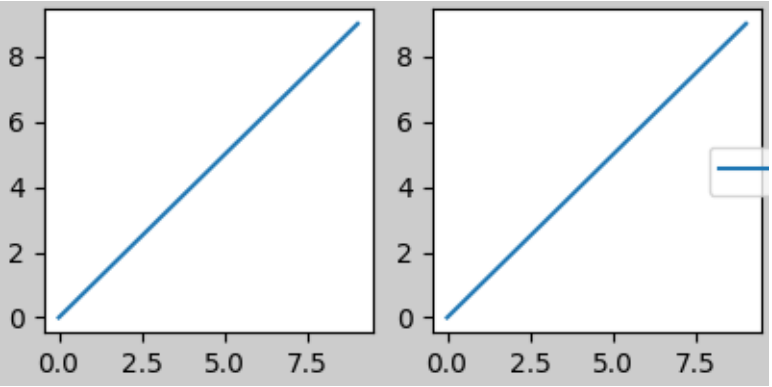


The saved file looks like:

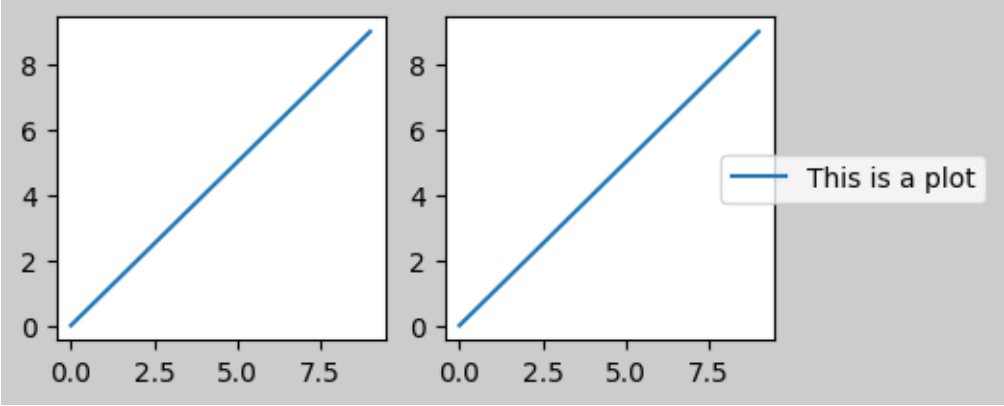


A better way to get around this awkwardness is to simply use the legend method provided by *Figure.legend*:

```
fig, axs = plt.subplots(1, 2, figsize=(4, 2), constrained_layout=True)
axs[0].plot(np.arange(10))
lines = axs[1].plot(np.arange(10), label='This is a plot')
labels = [l.get_label() for l in lines]
leg = fig.legend(lines, labels, loc='center left',
                 bbox_to_anchor=(0.8, 0.5), bbox_transform=axs[1].transAxes)
fig.savefig('../../doc/_static/constrained_layout_2b.png',
            bbox_inches='tight', dpi=100)
```



The saved file looks like:



Padding and Spacing

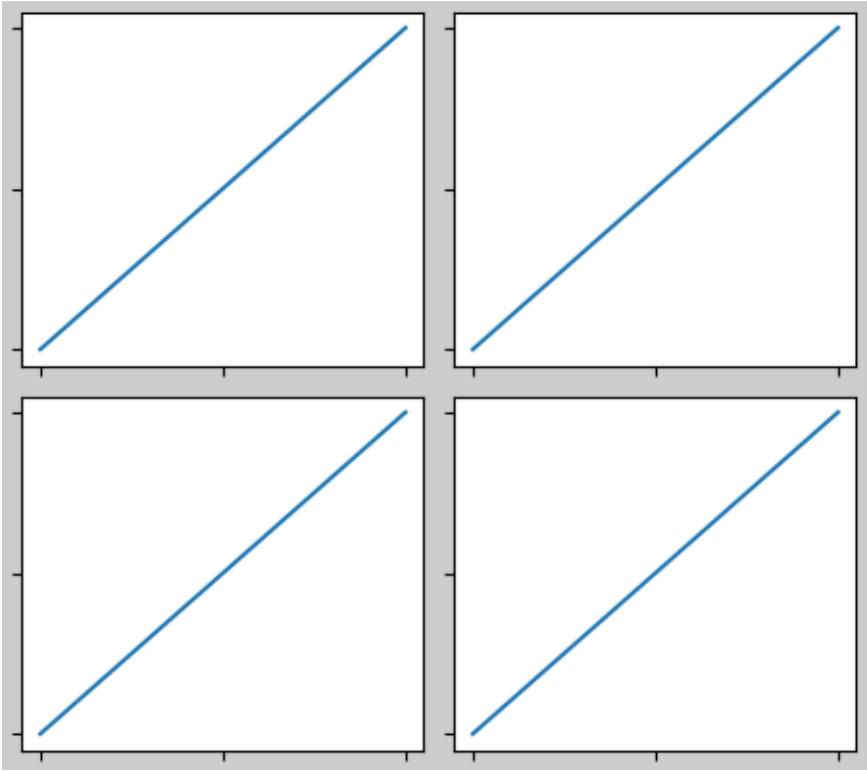
For constrained_layout, we have implemented a padding around the edge of each axes. This padding sets the distance from the edge of the plot, and the minimum distance between adjacent plots. It is specified in inches by the keyword arguments `w_pad` and `h_pad` to the function *set_constrained_layout_pads*:

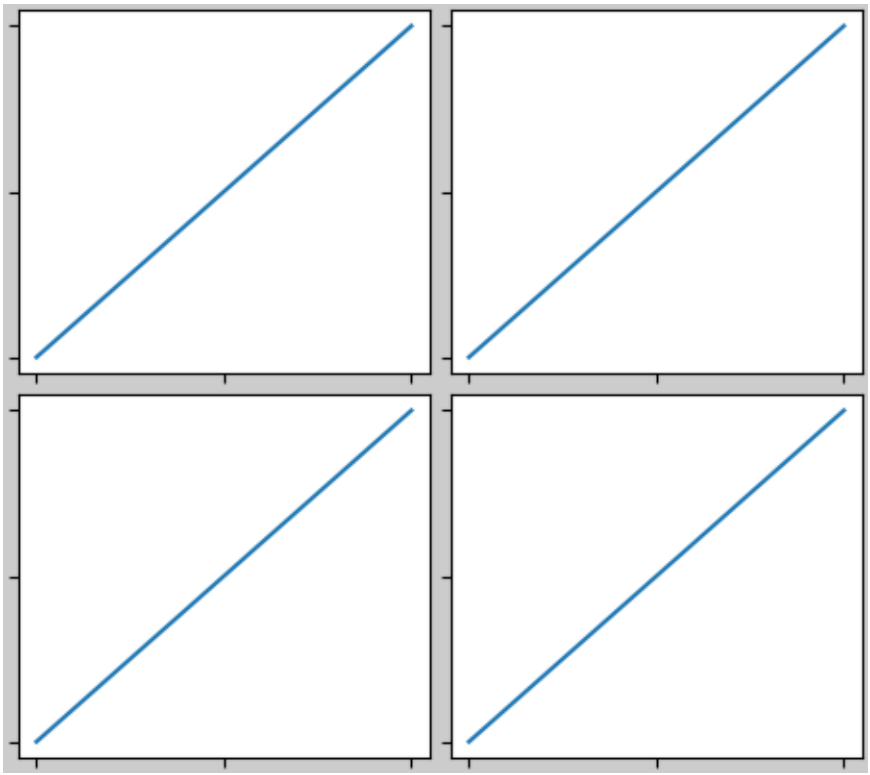
```
fig, axs = plt.subplots(2, 2, constrained_layout=True)
for ax in axs.flat:
    example_plot(ax, hide_labels=True)
fig.set_constrained_layout_pads(w_pad=4/72, h_pad=4/72, hspace=0, wspace=0)

fig, axs = plt.subplots(2, 2, constrained_layout=True)
```

(continues on next page)

(continued from previous page)

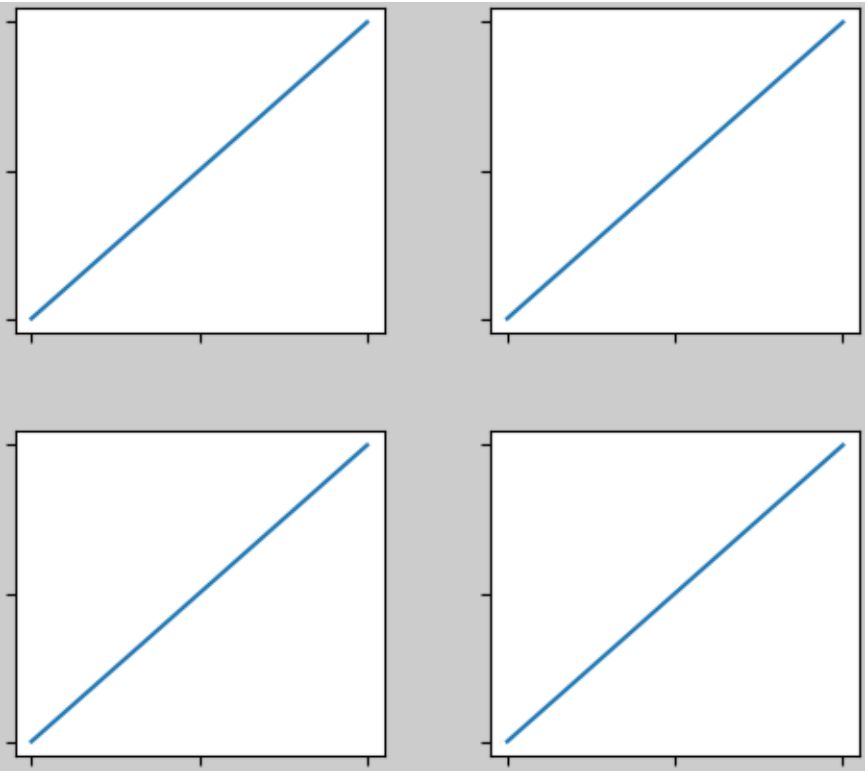
```
for ax in axs.flat:
    example_plot(ax, hide_labels=True)
fig.set_constrained_layout_pads(w_pad=2/72, h_pad=2/72, hspace=0, wspace=0)
```

- 

- 

Spacing between subplots is set by `wspace` and `hspace`. There are specified as a fraction of the size of the subplot group as a whole. If the size of the figure is changed, then these spaces change in proportion. Note in the below how the space at the edges doesn't change from the above, but the space between subplots does.

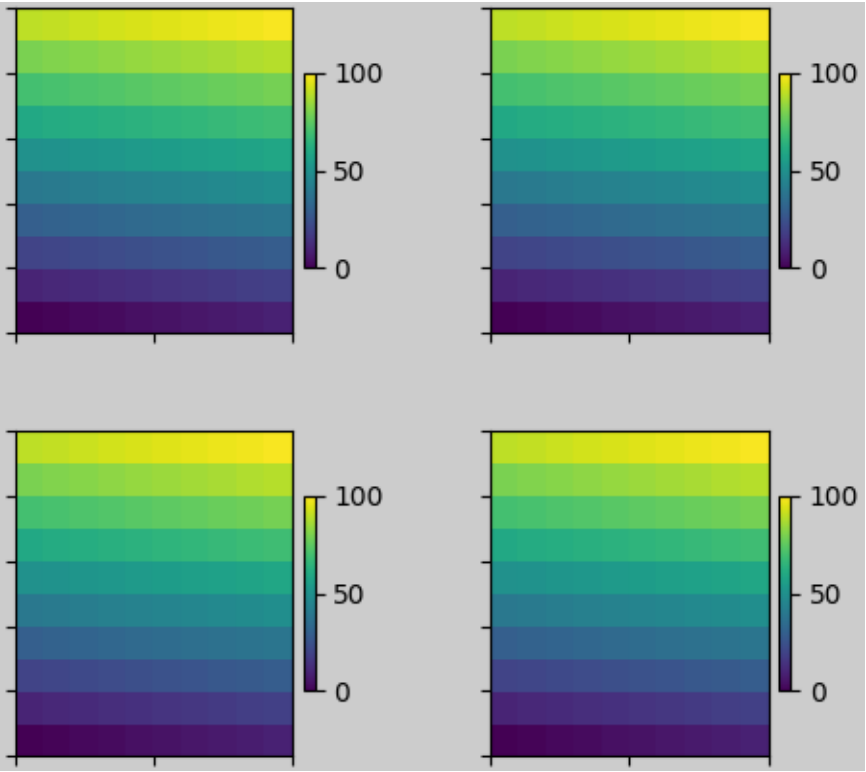
```
fig, axs = plt.subplots(2, 2, constrained_layout=True)
for ax in axs.flat:
    example_plot(ax, hide_labels=True)
fig.set_constrained_layout_pads(w_pad=2/72, h_pad=2/72, hspace=0.2, wspace=0.2)
```



Spacing with colorbars

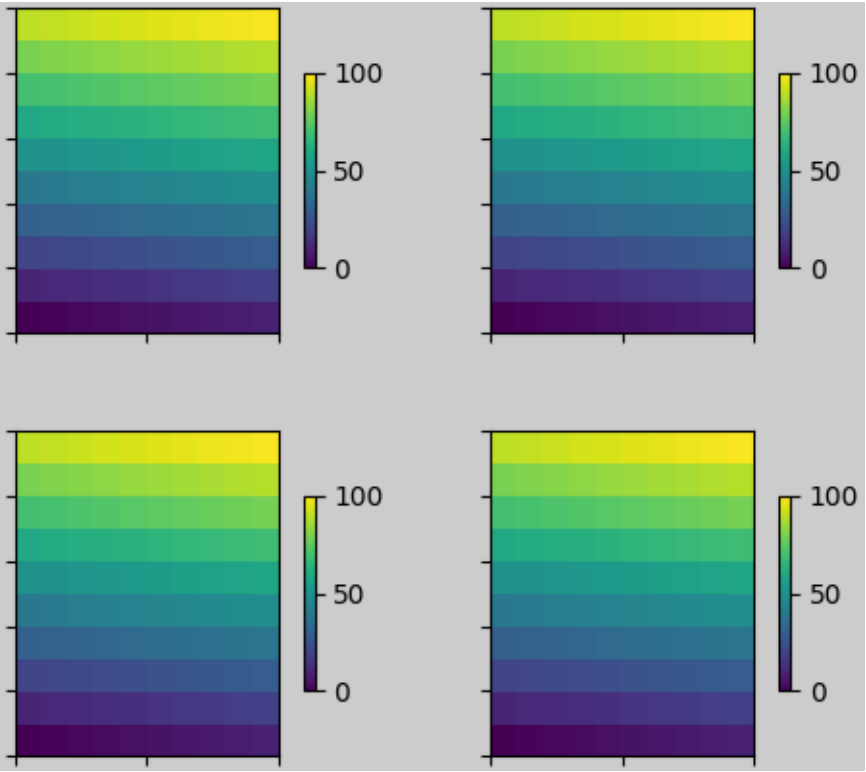
Colorbars will be placed `wspace` and `hsapce` apart from other subplots. The padding between the colorbar and the axis it is attached to will never be less than `w_pad` (for a vertical colorbar) or `h_pad` (for a horizontal colorbar). Note the use of the `pad` kwarg here in the `colorbar` call. It defaults to 0.02 of the size of the axis it is attached to.

```
fig, axs = plt.subplots(2, 2, constrained_layout=True)
for ax in axs.flat:
    pc = ax.pcolormesh(arr, **pc_kwargs)
    fig.colorbar(pc, ax=ax, shrink=0.6, pad=0)
    ax.set_xticklabels('')
    ax.set_yticklabels('')
fig.set_constrained_layout_pads(w_pad=2/72, h_pad=2/72, hspace=0.2, wspace=0.2)
```



In the above example, the colorbar will not ever be closer than 2 pts to the plot, but if we want it a bit further away, we can specify its value for `pad` to be non-zero.

```
fig, axs = plt.subplots(2, 2, constrained_layout=True)
for ax in axs.flat:
    pc = ax.pcolormesh(arr, **pc_kwargs)
    fig.colorbar(im, ax=ax, shrink=0.6, pad=0.05)
    ax.set_xticklabels('')
    ax.set_yticklabels('')
fig.set_constrained_layout_pads(w_pad=2/72, h_pad=2/72, hspace=0.2, wspace=0.2)
```



rcParams

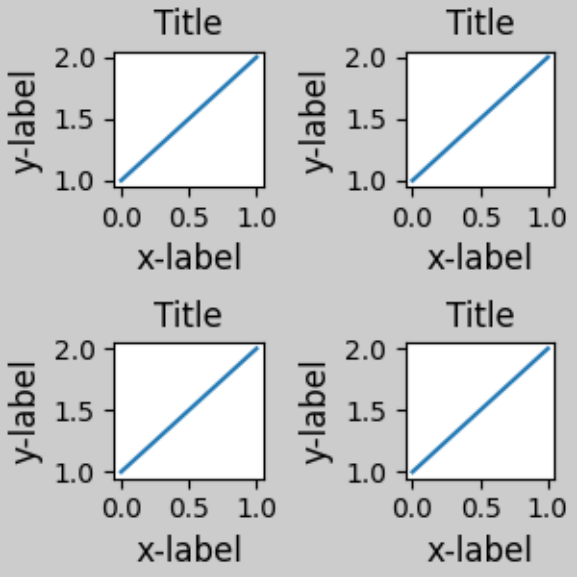
There are five *rcParams* that can be set, either in a script or in the `matplotlibrc` file. They all have the prefix `figure.constrained_layout`:

- `use`: Whether to use constrained_layout. Default is False
- **`w_pad`, `h_pad`: Padding around axes objects.**

  Float representing inches. Default is 3./72. inches (3 pts)
- **`wspace`, `hspace`: Space between subplot groups.**

  Float representing a fraction of the subplot widths being separated. Default is 0.02.

```
plt.rcParams['figure.constrained_layout.use'] = True
fig, axs = plt.subplots(2, 2, figsize=(3, 3))
for ax in axs.flat:
    example_plot(ax)
```

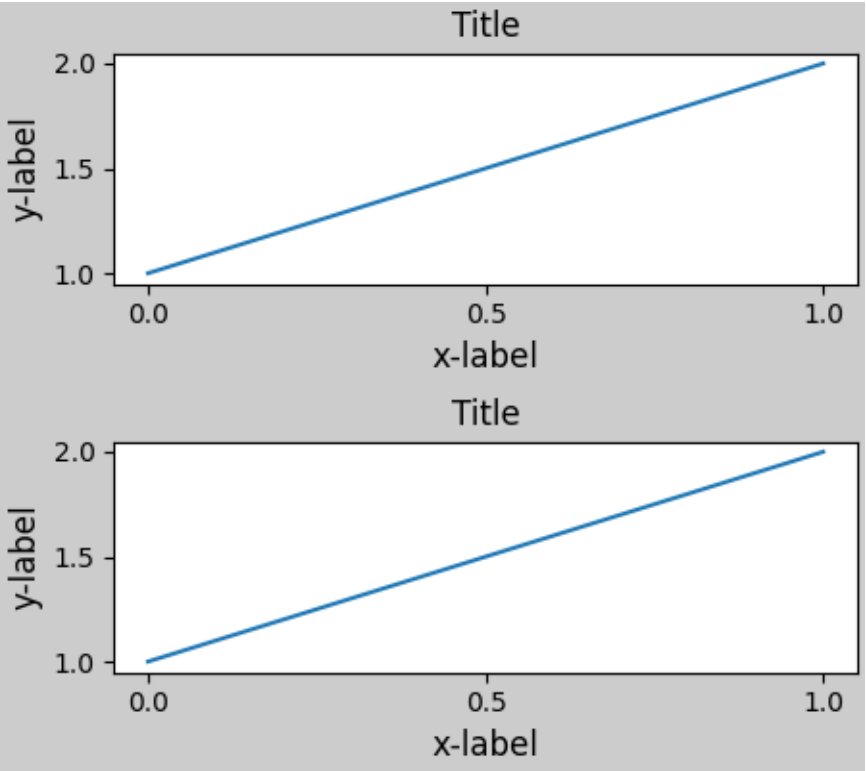


Use with GridSpec

constrained_layout is meant to be used with *subplots()* or *GridSpec()* and *add_subplot()*.

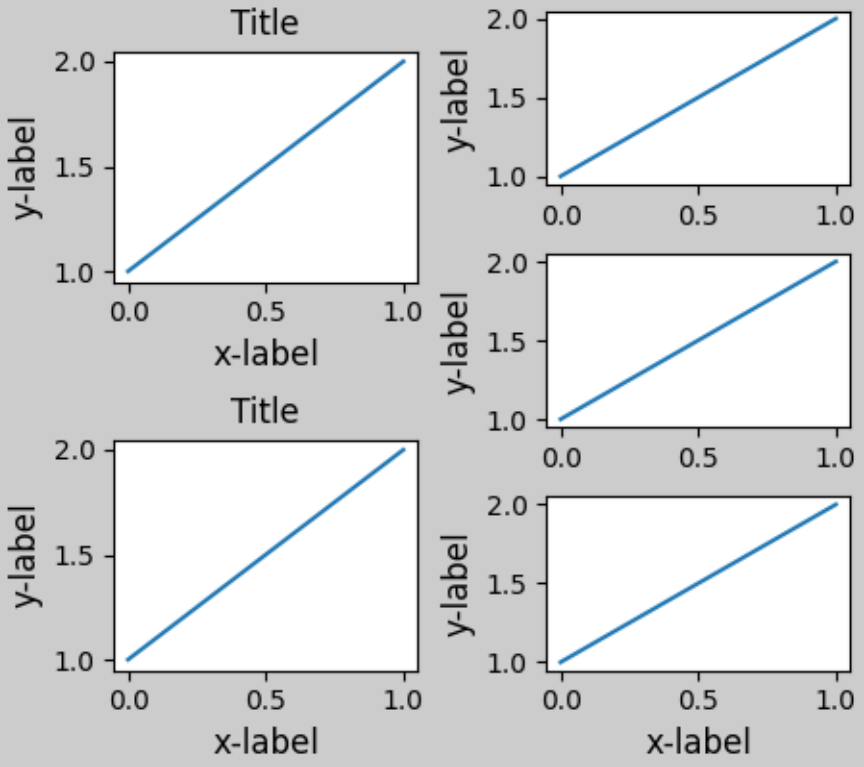
Note that in what follows `constrained_layout=True`

```
fig = plt.figure()

gs1 = gridspec.GridSpec(2, 1, figure=fig)
ax1 = fig.add_subplot(gs1[0])
ax2 = fig.add_subplot(gs1[1])

example_plot(ax1)
example_plot(ax2)
```



More complicated gridspec layouts are possible. Note here we use the convenience functions `add_gridspec` and `subgridspec`.

```
fig = plt.figure()

gs0 = fig.add_gridspec(1, 2)

gs1 = gs0[0].subgridspec(2, 1)
ax1 = fig.add_subplot(gs1[0])
ax2 = fig.add_subplot(gs1[1])

example_plot(ax1)
example_plot(ax2)

gs2 = gs0[1].subgridspec(3, 1)

for ss in gs2:
    ax = fig.add_subplot(ss)
    example_plot(ax)
    ax.set_title("")
    ax.set_xlabel("")

ax.set_xlabel("x-label", fontsize=12)
```

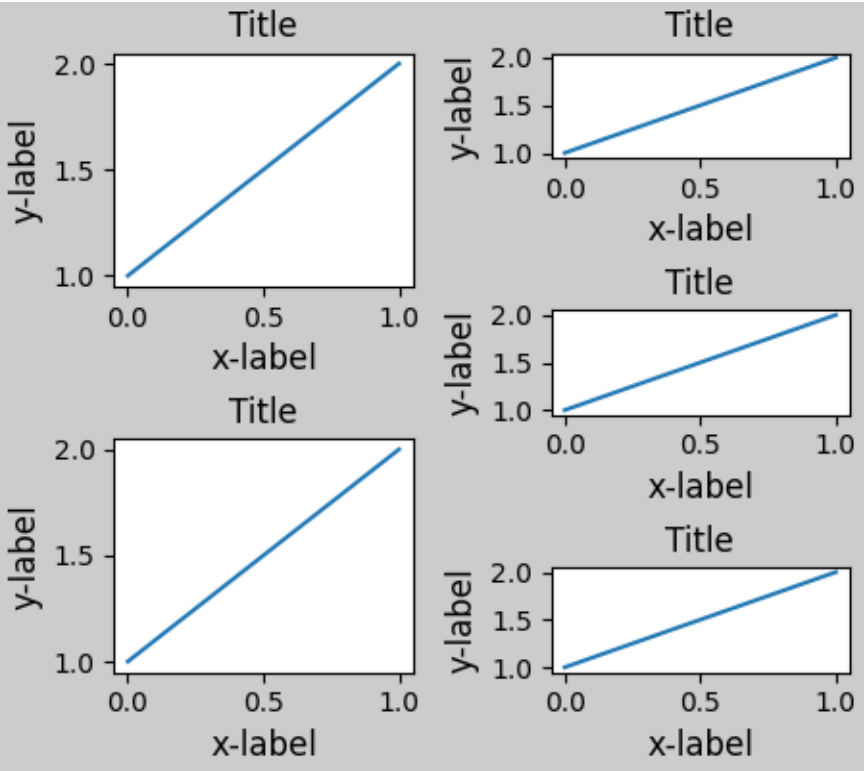


Out:

```
Text(0.5, 0, 'x-label')
```

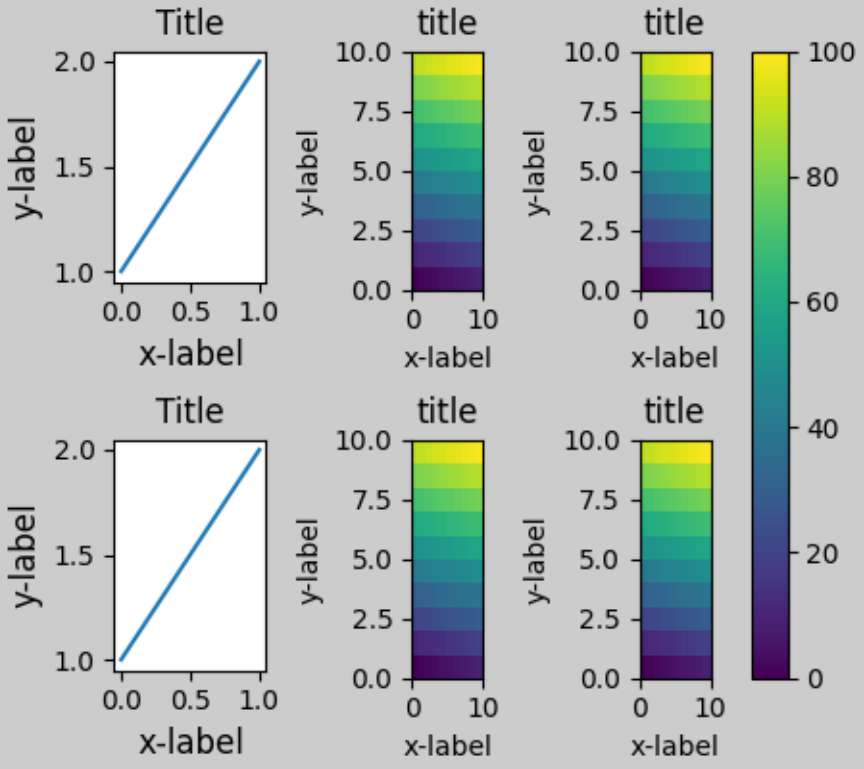
Note that in the above the left and columns don't have the same vertical extent. If we want the top and bottom of the two grids to line up then they need to be in the same gridspec:

```
fig = plt.figure()

gs0 = fig.add_gridspec(6, 2)

ax1 = fig.add_subplot(gs0[:3, 0])
ax2 = fig.add_subplot(gs0[3:, 0])

example_plot(ax1)
example_plot(ax2)

ax = fig.add_subplot(gs0[0:2, 1])
example_plot(ax)
ax = fig.add_subplot(gs0[2:4, 1])
example_plot(ax)
ax = fig.add_subplot(gs0[4:, 1])
example_plot(ax)
```



This example uses two gridspecs to have the colorbar only pertain to one set of pcolors. Note how the left column is wider than the two right-hand columns because of this. Of course, if you wanted the subplots to be the same size you only needed one gridspec.

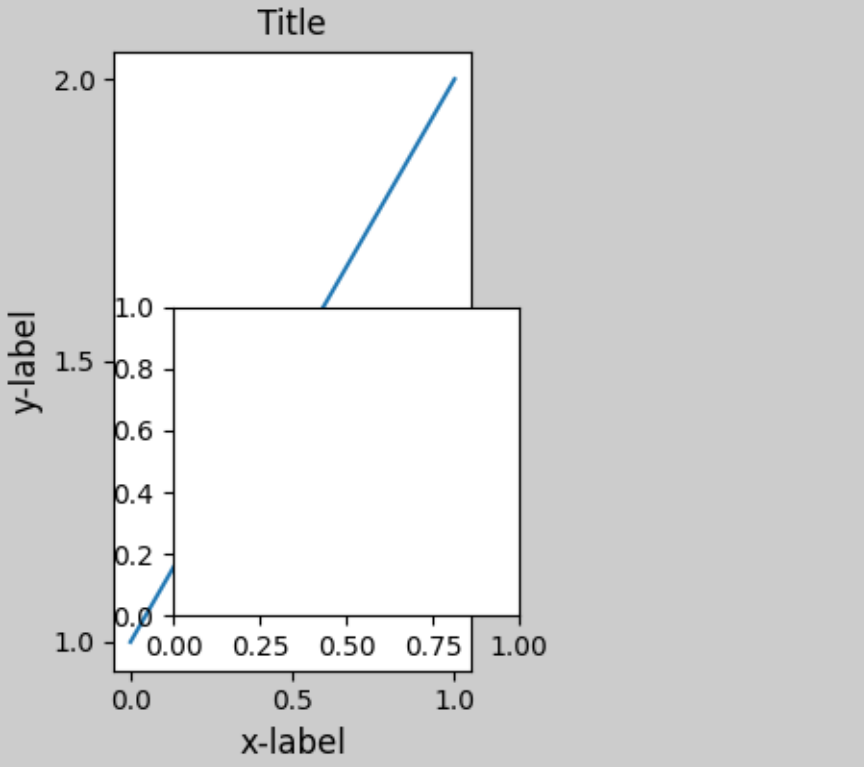
```
def docomplicated(suptitle=None):
    fig = plt.figure()
    gs0 = fig.add_gridspec(1, 2, figure=fig, width_ratios=[1., 2.])
    gsl = gs0[0].subgridspec(2, 1)
    gsr = gs0[1].subgridspec(2, 2)

    for gs in gsl:
        ax = fig.add_subplot(gs)
        example_plot(ax)
    axs = []
    for gs in gsr:
        ax = fig.add_subplot(gs)
        pcm = ax.pcolormesh(arr, **pc_kwargs)
        ax.set_xlabel('x-label')
        ax.set_ylabel('y-label')
        ax.set_title('title')

        axs += [ax]
    fig.colorbar(pcm, ax=axs)
    if suptitle is not None:
        fig.suptitle(suptitle)

docomplicated()
```



Manually setting axes positions

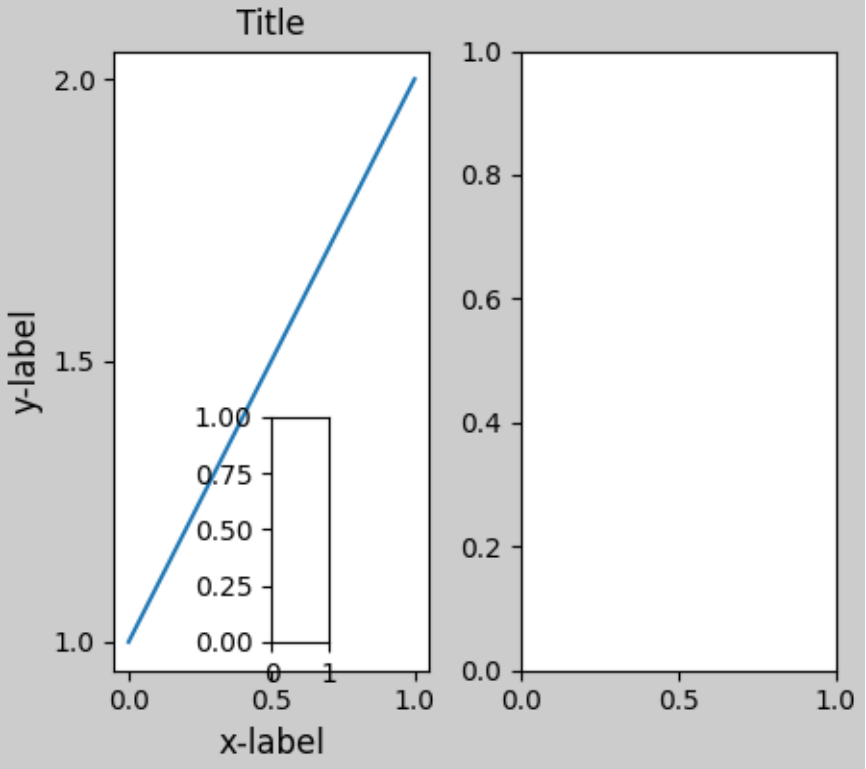
There can be good reasons to manually set an axes position. A manual call to *set_position* will set the axes so constrained_layout has no effect on it anymore. (Note that constrained_layout still leaves the space for the axes that is moved).

```
fig, axs = plt.subplots(1, 2)
example_plot(axs[0], fontsize=12)
axs[1].set_position([0.2, 0.2, 0.4, 0.4])
```



If you want an inset axes in data-space, you need to manually execute the layout using `fig.execute_constrained_layout()` call. The inset figure will then be properly positioned. However, it will not be properly positioned if the size of the figure is subsequently changed. Similarly, if the figure is printed to another backend, there may be slight changes of location due to small differences in how the backends render fonts.

```
from matplotlib.transforms import Bbox

fig, axs = plt.subplots(1, 2)
example_plot(axs[0], fontsize=12)
fig.execute_constrained_layout()
# put into data-space:
bb_data_ax2 = Bbox.from_bounds(0.5, 1., 0.2, 0.4)
disp_coords = axs[0].transData.transform(bb_data_ax2)
fig_coords_ax2 = fig.transFigure.inverted().transform(disp_coords)
bb_ax2 = Bbox(fig_coords_ax2)
ax2 = fig.add_axes(bb_ax2)
```



Manually turning off `constrained_layout`

`constrained_layout` usually adjusts the axes positions on each draw of the figure. If you want to get the spacing provided by `constrained_layout` but not have it update, then do the initial draw and then call `fig.set_constrained_layout(False)`. This is potentially useful for animations where the tick labels may change length.

Note that `constrained_layout` is turned off for `ZOOM` and `PAN` GUI events for the backends that use the toolbar. This prevents the axes from changing position during zooming and panning.

Limitations

Incompatible functions

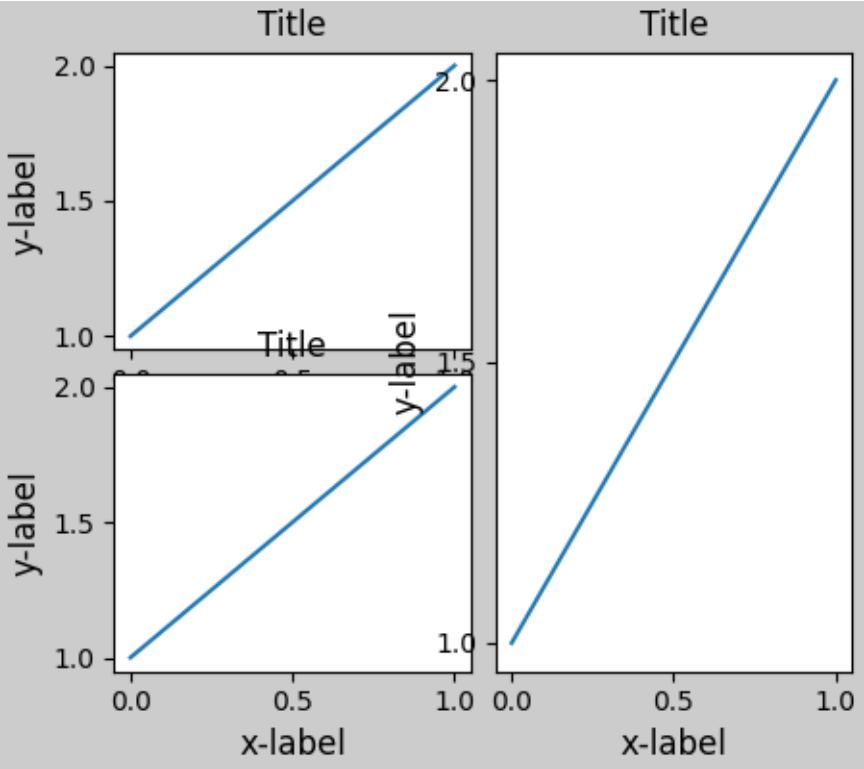
`constrained_layout` will not work on subplots created via *`pyplot.subplot`*. The reason is that each call to *`pyplot.subplot`* creates a separate *`GridSpec`* instance and `constrained_layout` uses (nested) gridspecs to carry out the layout. So the following fails to yield a nice layout:

```
fig = plt.figure()

ax1 = plt.subplot(221)
```

(continues on next page)

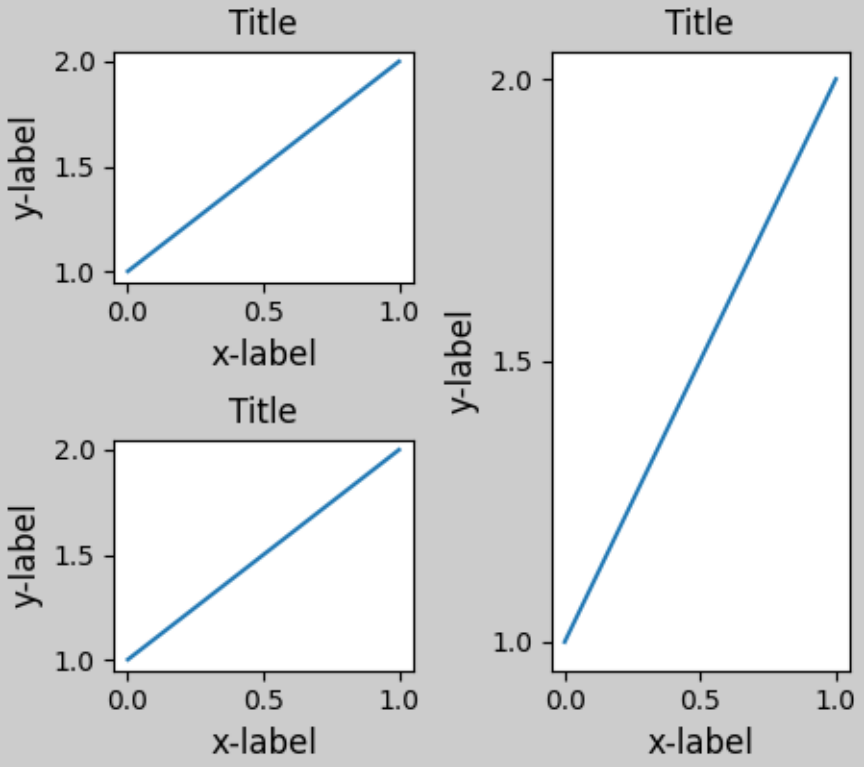
(continued from previous page)

```
ax2 = plt.subplot(223)
ax3 = plt.subplot(122)

example_plot(ax1)
example_plot(ax2)
example_plot(ax3)
```



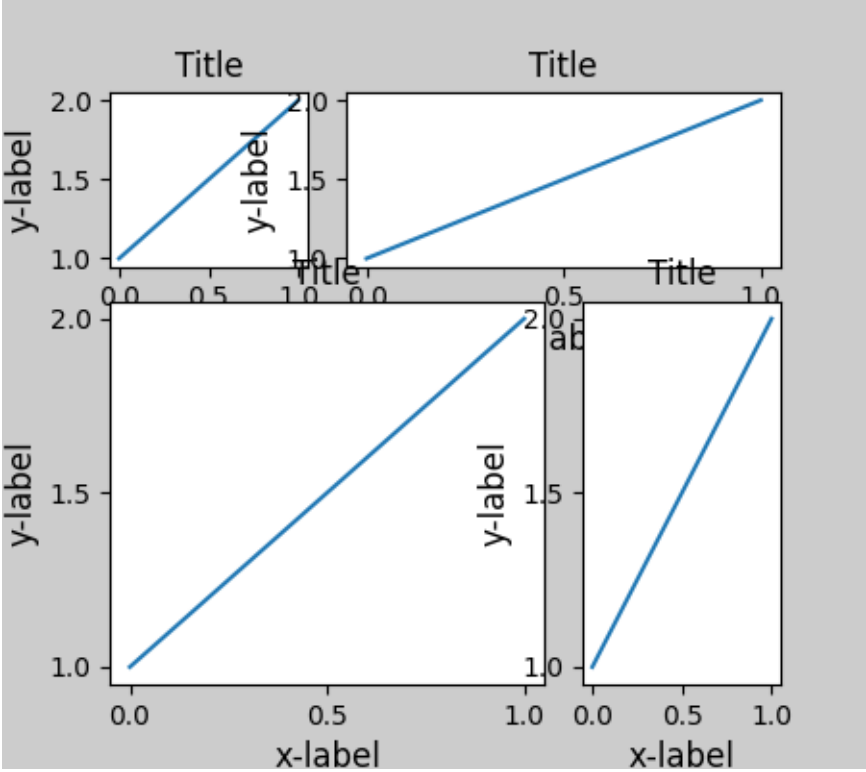
Of course that layout is possible using a gridspec:

```
fig = plt.figure()
gs = fig.add_gridspec(2, 2)

ax1 = fig.add_subplot(gs[0, 0])
ax2 = fig.add_subplot(gs[1, 0])
ax3 = fig.add_subplot(gs[:, 1])

example_plot(ax1)
example_plot(ax2)
example_plot(ax3)
```



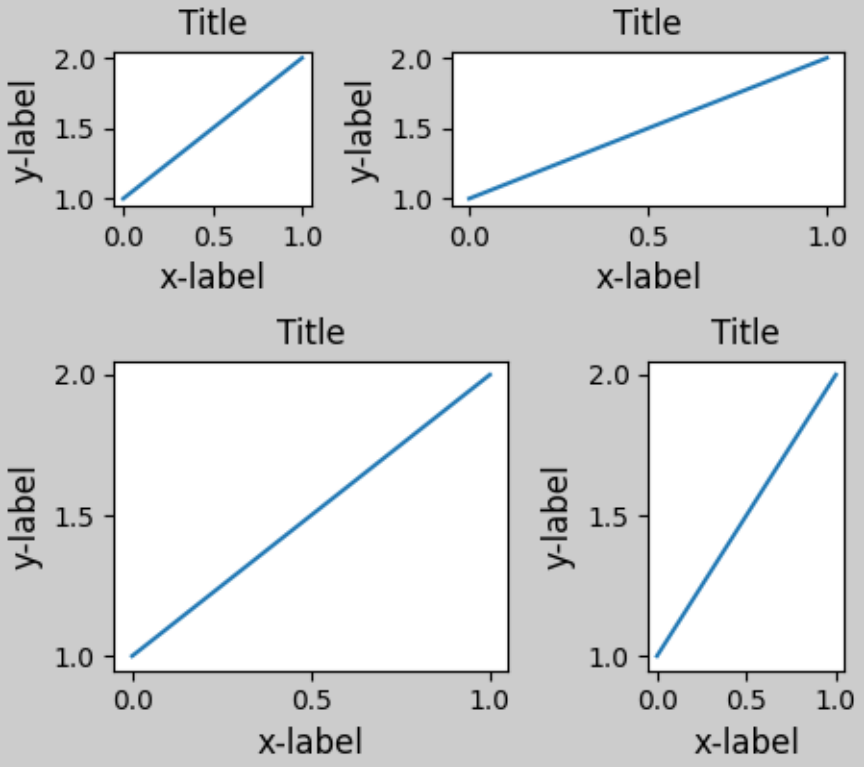
Similarly, *subplot2grid()* doesn't work for the same reason: each call creates a different parent gridspec.

```
fig = plt.figure()

ax1 = plt.subplot2grid((3, 3), (0, 0))
ax2 = plt.subplot2grid((3, 3), (0, 1), colspan=2)
ax3 = plt.subplot2grid((3, 3), (1, 0), colspan=2, rowspan=2)
ax4 = plt.subplot2grid((3, 3), (1, 2), rowspan=2)

example_plot(ax1)
example_plot(ax2)
example_plot(ax3)
example_plot(ax4)
```



The way to make this plot compatible with `constrained_layout` is again to use `gridspec` directly

```
fig = plt.figure()
gs = fig.add_gridspec(3, 3)

ax1 = fig.add_subplot(gs[0, 0])
ax2 = fig.add_subplot(gs[0, 1:])
ax3 = fig.add_subplot(gs[1:, 0:2])
ax4 = fig.add_subplot(gs[1:, -1])

example_plot(ax1)
example_plot(ax2)
example_plot(ax3)
example_plot(ax4)
```



Other Caveats

- `constrained_layout` only considers ticklabels, axis labels, titles, and legends. Thus, other artists may be clipped and also may overlap.
- It assumes that the extra space needed for ticklabels, axis labels, and titles is independent of original location of axes. This is often true, but there are rare cases where it is not.
- There are small differences in how the backends handle rendering fonts, so the results will not be pixel-identical.
- An artist using axes coordinates that extend beyond the axes boundary will result in unusual layouts when added to an axes. This can be avoided by adding the artist directly to the *Figure* using *add_artist()*. See *ConnectionPatch* for an example.

Debugging

Constrained-layout can fail in somewhat unexpected ways. Because it uses a constraint solver the solver can find solutions that are mathematically correct, but that aren't at all what the user wants. The usual failure mode is for all sizes to collapse to their smallest allowable value. If this happens, it is for one of two reasons:

1. There was not enough room for the elements you were requesting to draw.
2. There is a bug - in which case open an issue at https://github.com/matplotlib/matplotlib/issues.

If there is a bug, please report with a self-contained example that does not require outside data or dependencies (other than numpy).

Notes on the algorithm

The algorithm for the constraint is relatively straightforward, but has some complexity due to the complex ways we can layout a figure.

Figure layout

Figures are laid out in a hierarchy:

1. Figure: `fig = plt.figure()`

a. Gridspec `gs0 = gridspec.GridSpec(1, 2, figure=fig)`

   i. Subplotspec: `ss = gs[0, 0]`

      1. Axes: `ax0 = fig.add_subplot(ss)`

   ii. Subplotspec: `ss = gs[0, 1]`

   1. Gridspec: `gsR = gridspec.GridSpecFromSubplotSpec(2, 1, ss)`

      - Subplotspec: `ss = gsR[0, 0]`

        - Axes: `axR0 = fig.add_subplot(ss)`

      - Subplotspec: `ss = gsR[1, 0]`

        - Axes: `axR1 = fig.add_subplot(ss)`

Each item has a layoutbox associated with it. The nesting of gridspecs created with *GridSpecFromSubplotSpec* can be arbitrarily deep.

Each *Axes* has *two* layoutboxes. The first one, `ax._layoutbox` represents the outside of the Axes and all its decorations (i.e. ticklabels, axis labels, etc.). The second layoutbox corresponds to the Axes' `ax.position`, which sets where in the figure the spines are placed.

Why so many stacked containers? Ideally, all that would be needed are the Axes layout boxes. For the Gridspec case, a container is needed if the Gridspec is nested via *GridSpecFromSubplotSpec*. At the top level, it is desirable for symmetry, but it also makes room for *suptitle*.

For the Subplotspec/Axes case, Axes often have colorbars or other annotations that need to be packaged inside the Subplotspec, hence the need for the outer layer.

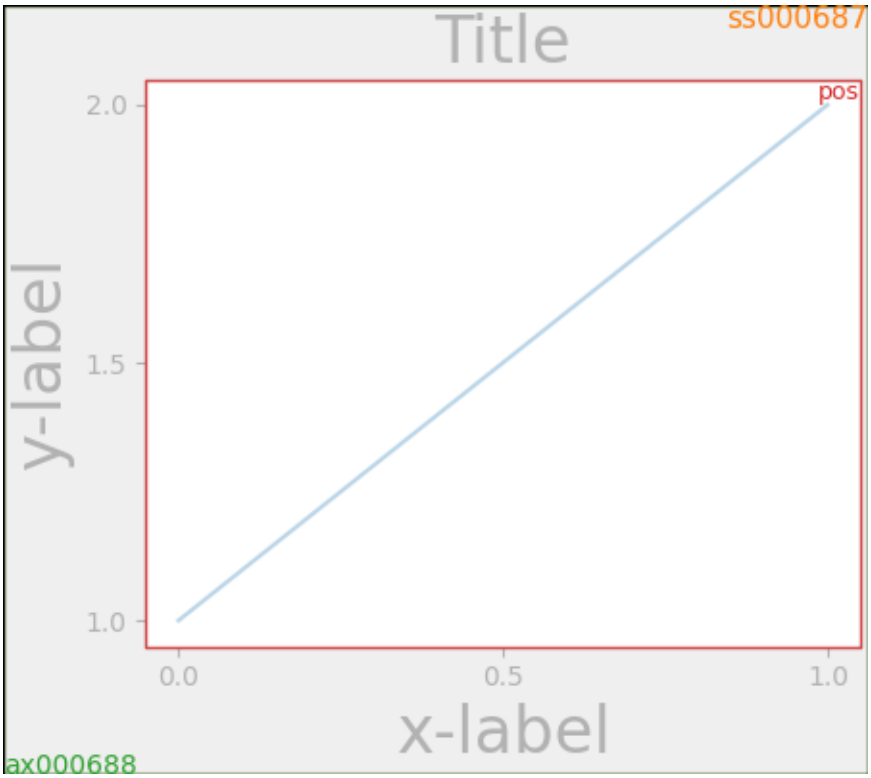
Simple case: one Axes

For a single Axes the layout is straight forward. The Figure and outer Gridspec layoutboxes coincide. The Subplotspec and Axes boxes also coincide because the Axes has no colorbar. Note the difference between the red `pos` box and the green `ax` box is set by the size of the decorations around the Axes.

In the code, this is accomplished by the entries in `do_constrained_layout()` like:

```
ax._poslayoutbox.edit_left_margin_min(-bbox.x0 + pos.x0 + w_padt)
```

```
from matplotlib._layoutbox import plot_children

fig, ax = plt.subplots(constrained_layout=True)
example_plot(ax, fontsize=24)
plot_children(fig, fig._layoutbox, printit=False)
```



Simple case: two Axes

For this case, the Axes layoutboxes and the Subplotspec boxes still co-incide. However, because the decorations in the right-hand plot are so much smaller than the left-hand, so the right-hand layoutboxes are smaller.

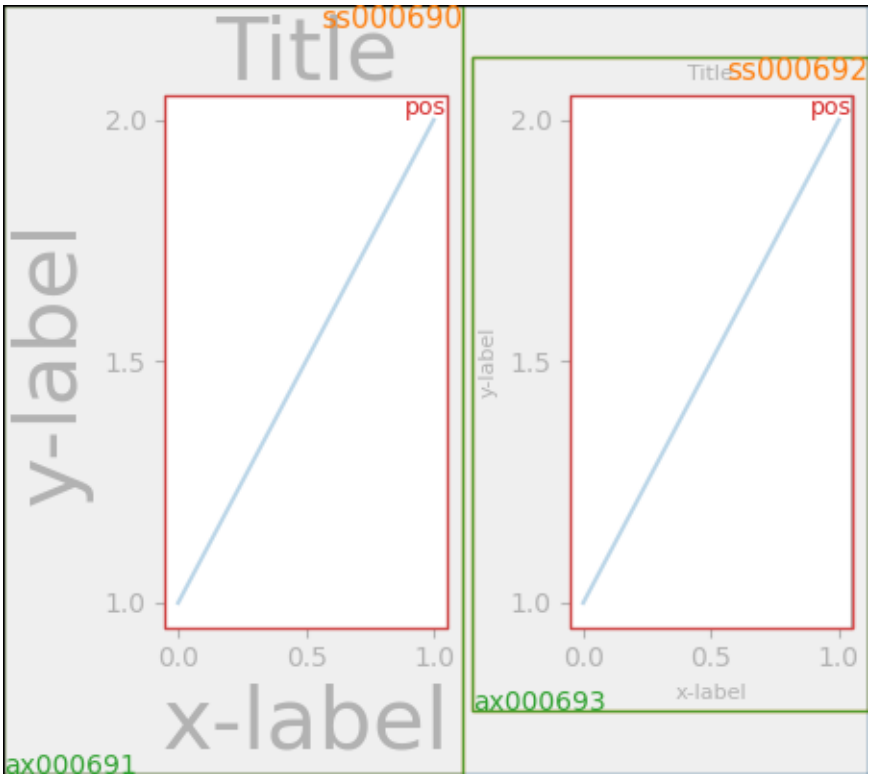
The Subplotspec boxes are laid out in the code in the subroutine `arange_subplotspecs()`, which simply checks the subplotspecs in the code against one another and stacks them appropriately.

The two `pos` axes are lined up. Because they have the same minimum row, they are lined up at the top. Because they have the same maximum row they are lined up at the bottom. In the code this is accomplished via the calls to `layoutbox.align`. If there was more than one row, then the same horizontal alignment would occur between the rows.

The two `pos` axes are given the same width because the subplotspecs occupy the same number of columns. This is accomplished in the code where `dcols0` is compared to `dcolsC`. If they are equal, then their widths are constrained to be equal.

While it is a bit subtle in this case, note that the division between the Subplotspecs is *not* centered, but has been moved to the right to make space for the larger labels on the left-hand plot.

```
fig, ax = plt.subplots(1, 2, constrained_layout=True)
example_plot(ax[0], fontsize=32)
example_plot(ax[1], fontsize=8)
plot_children(fig, fig._layoutbox, printit=False)
```



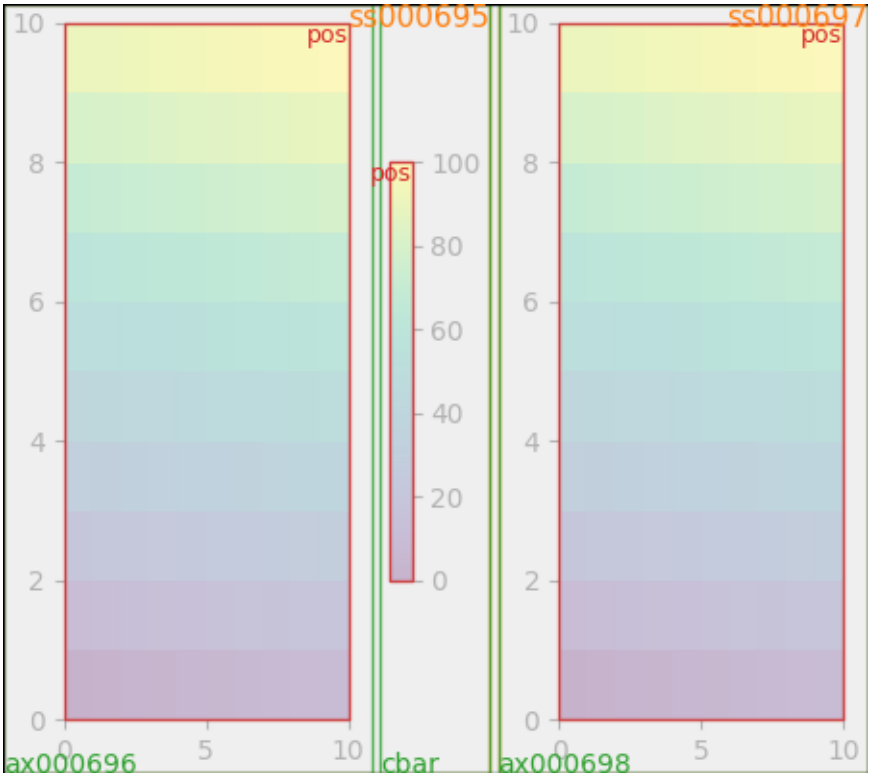
Two Axes and colorbar

Adding a colorbar makes it clear why the Subplotspec layoutboxes must be different from the axes layoutboxes. Here we see the left-hand subplotspec has more room to accommodate the *colorbar*, and that there are two green ax boxes inside the ss box.

Note that the width of the pos boxes is still the same because of the constraint on their widths because their subplotspecs occupy the same number of columns (one in this example).

The colorbar layout logic is contained in *make_axes* which calls `_constrained_layout.layoutcolorbarsingle()` for cbars attached to a single axes, and `_constrained_layout.layoutcolorbargridspec()` if the colorbar is associated with a gridspec.

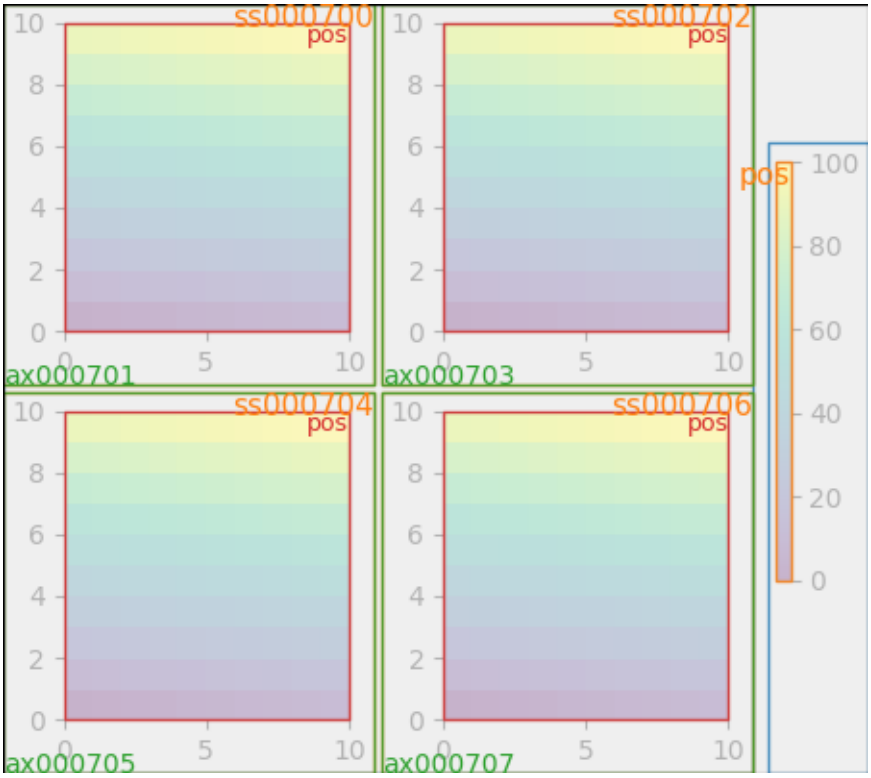
```
fig, ax = plt.subplots(1, 2, constrained_layout=True)
im = ax[0].pcolormesh(arr, **pc_kwargs)
fig.colorbar(im, ax=ax[0], shrink=0.6)
im = ax[1].pcolormesh(arr, **pc_kwargs)
plot_children(fig, fig._layoutbox, printit=False)
```



Colorbar associated with a Gridspec

This example shows the Subplotspec layoutboxes being made smaller by a colorbar layoutbox. The size of the colorbar layoutbox is set to be `shrink` smaller than the vertical extent of the `pos` layoutboxes in the gridspec, and it is made to be centered between those two points.

```
fig, axs = plt.subplots(2, 2, constrained_layout=True)
for ax in axs.flat:
    im = ax.pcolormesh(arr, **pc_kwargs)
fig.colorbar(im, ax=axs, shrink=0.6)
plot_children(fig, fig._layoutbox, printit=False)
```



Uneven sized Axes

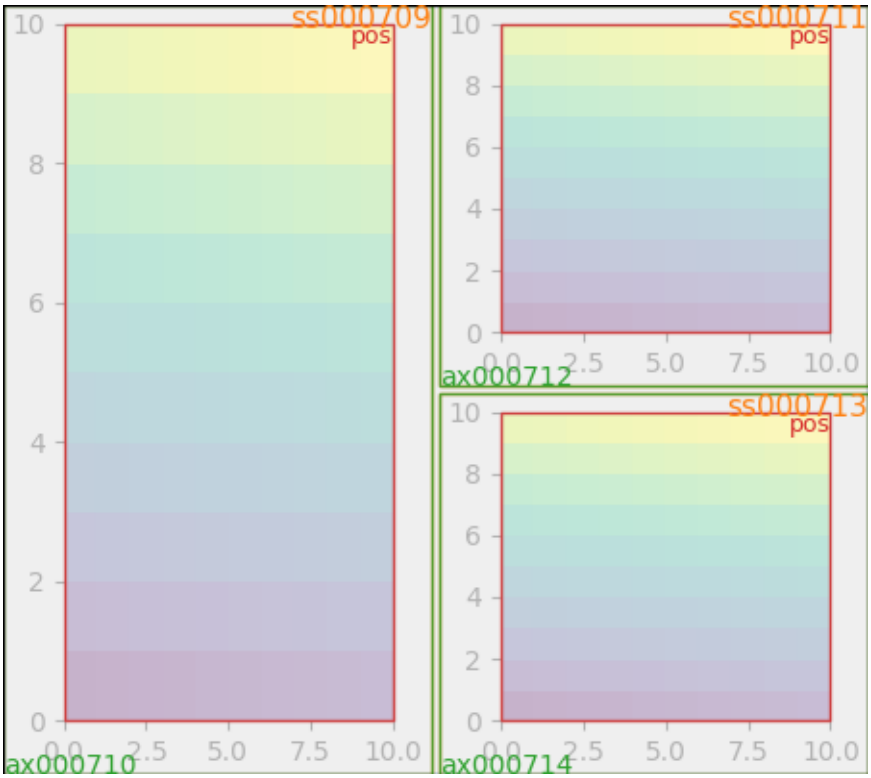
There are two ways to make axes have an uneven size in a Gridspec layout, either by specifying them to cross Gridspecs rows or columns, or by specifying width and height ratios.

The first method is used here. The constraint that makes the heights be correct is in the code where `drowsC < drows0` which in this case would be 1 is less than 2. So we constrain the height of the 1-row Axes to be less than half the height of the 2-row Axes.

Note: This algorithm can be wrong if the decorations attached to the smaller axes

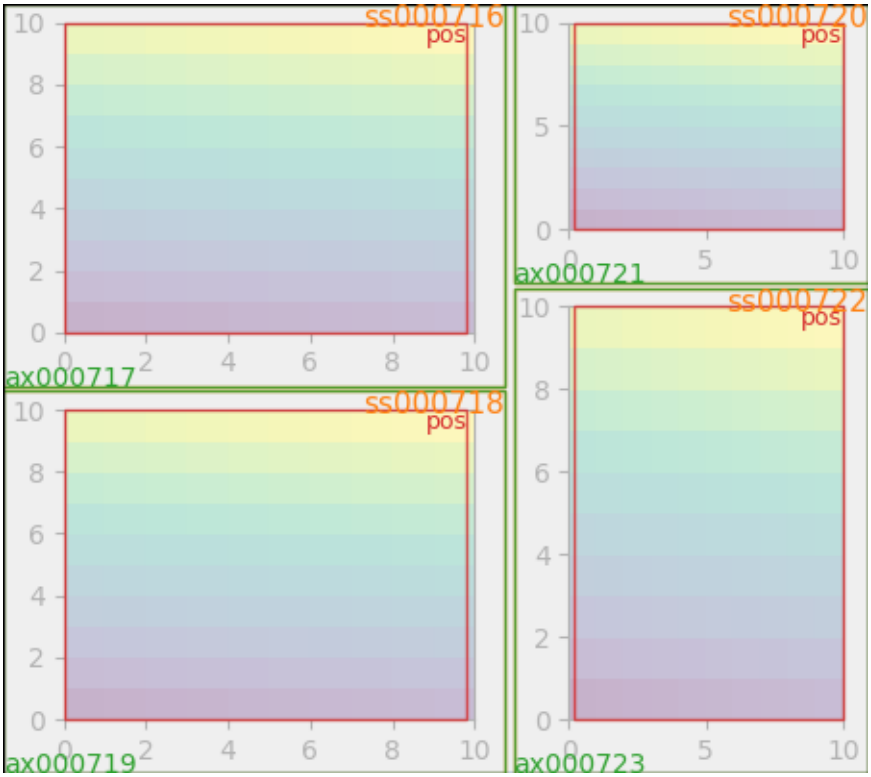
are very large, so there is an unaccounted-for edge case.

```
fig = plt.figure(constrained_layout=True)
gs = gridspec.GridSpec(2, 2, figure=fig)
ax = fig.add_subplot(gs[:, 0])
im = ax.pcolormesh(arr, **pc_kwargs)
ax = fig.add_subplot(gs[0, 1])
im = ax.pcolormesh(arr, **pc_kwargs)
ax = fig.add_subplot(gs[1, 1])
im = ax.pcolormesh(arr, **pc_kwargs)
plot_children(fig, fig._layoutbox, printit=False)
```



Height and width ratios are accommodated with the same part of the code with the smaller axes always constrained to be less in size than the larger.

```
fig = plt.figure(constrained_layout=True)
gs = gridspec.GridSpec(3, 2, figure=fig,
                       height_ratios=[1., 0.5, 1.5], width_ratios=[1.2, 0.8])
ax = fig.add_subplot(gs[:2, 0])
im = ax.pcolormesh(arr, **pc_kwargs)
ax = fig.add_subplot(gs[2, 0])
im = ax.pcolormesh(arr, **pc_kwargs)
ax = fig.add_subplot(gs[0, 1])
im = ax.pcolormesh(arr, **pc_kwargs)
ax = fig.add_subplot(gs[1:, 1])
im = ax.pcolormesh(arr, **pc_kwargs)
plot_children(fig, fig._layoutbox, printit=False)
```

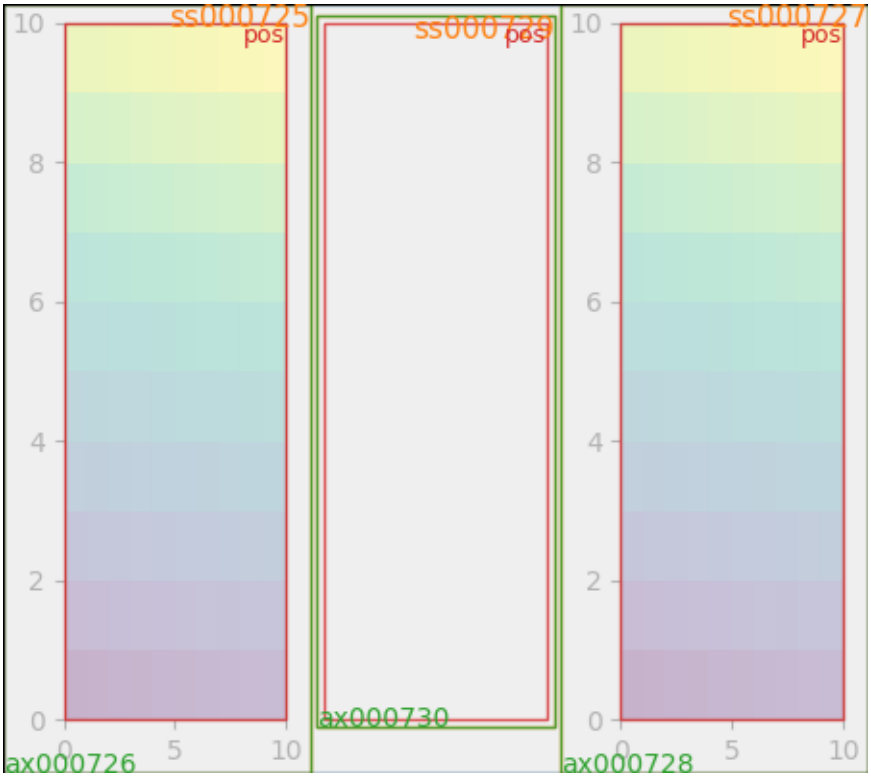


Empty gridspec slots

The final piece of the code that has not been explained is what happens if there is an empty gridspec opening. In that case a fake invisible axes is added and we proceed as before. The empty gridspec has no decorations, but the axes position in made the same size as the occupied Axes positions.

This is done at the start of `_constrained_layout.do_constrained_layout()` (`hassubplotspec`).

```
fig = plt.figure(constrained_layout=True)
gs = gridspec.GridSpec(1, 3, figure=fig)
ax = fig.add_subplot(gs[0])
im = ax.pcolormesh(arr, **pc_kwargs)
ax = fig.add_subplot(gs[-1])
im = ax.pcolormesh(arr, **pc_kwargs)
plot_children(fig, fig._layoutbox, printit=False)
plt.show()
```



Other notes

The layout is called only once. This is OK if the original layout was pretty close (which it should be in most cases). However, if the layout changes a lot from the default layout then the decorators can change size. In particular the x and ytick labels can change. If this happens, then we should probably call the whole routine twice.

Total running time of the script: (0 minutes 13.819 seconds)

2.2.6 Tight Layout guide

How to use tight-layout to fit plots within your figure cleanly.

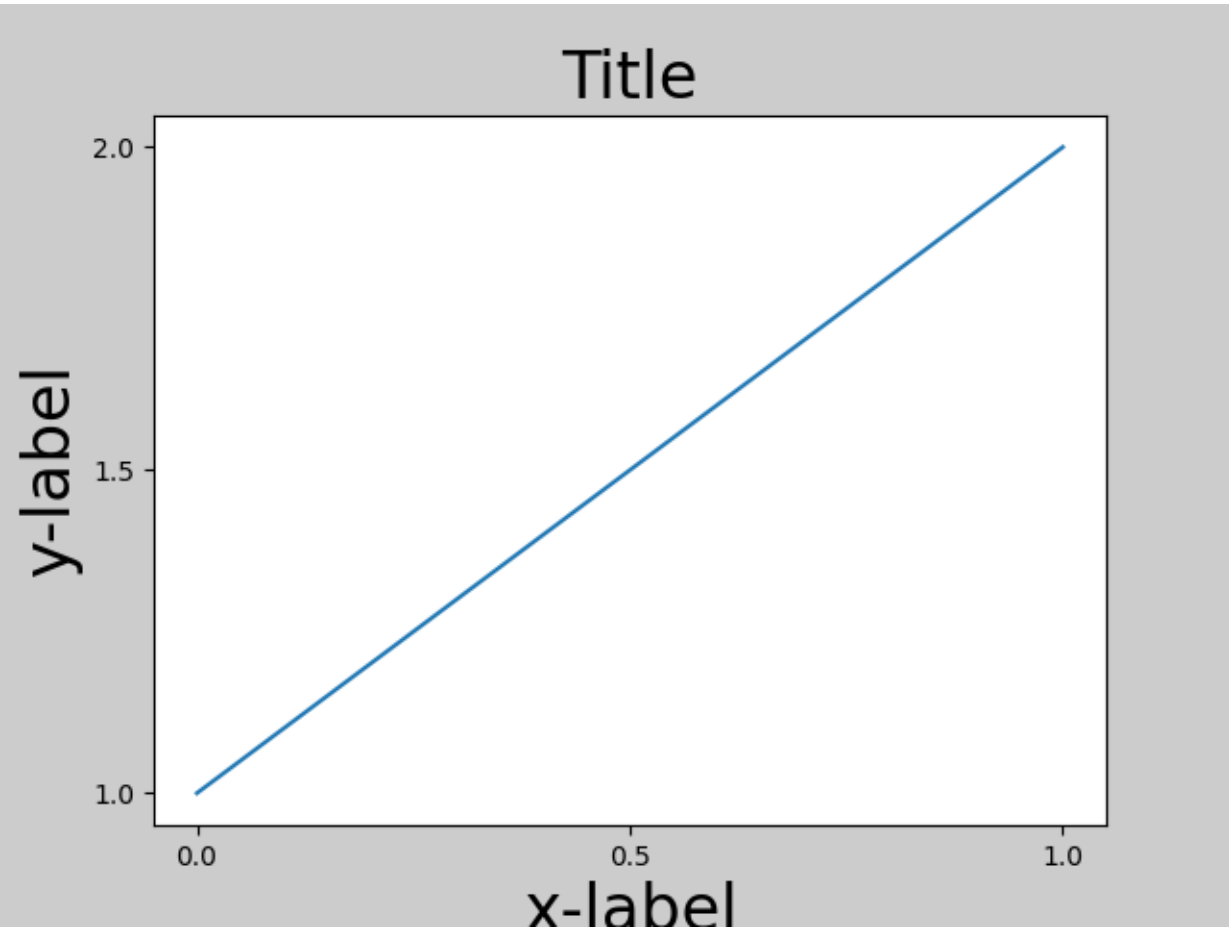
tight_layout automatically adjusts subplot params so that the subplot(s) fits in to the figure area. This is an experimental feature and may not work for some cases. It only checks the extents of ticklabels, axis labels, and titles.

An alternative to *tight_layout* is *constrained_layout*.

Simple Example

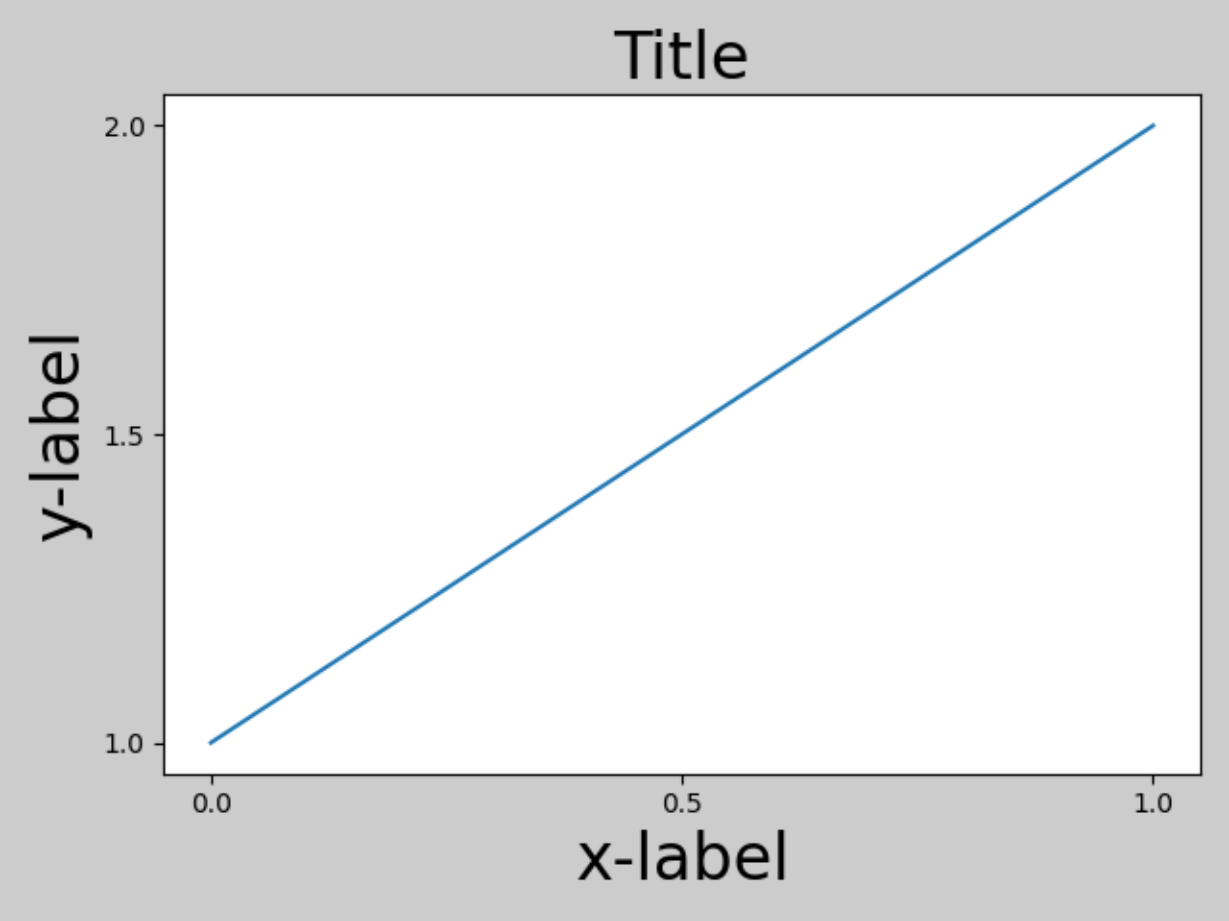
In matplotlib, the location of axes (including subplots) are specified in normalized figure coordinates. It can happen that your axis labels or titles (or sometimes even ticklabels) go outside the figure area, and are thus clipped.

```
import matplotlib.pyplot as plt
import numpy as np

plt.rcParams['savefig.facecolor'] = "0.8"


def example_plot(ax, fontsize=12):
    ax.plot([1, 2])

    ax.locator_params(nbins=3)
    ax.set_xlabel('x-label', fontsize=fontsize)
    ax.set_ylabel('y-label', fontsize=fontsize)
    ax.set_title('Title', fontsize=fontsize)

plt.close('all')
fig, ax = plt.subplots()
example_plot(ax, fontsize=24)
```



To prevent this, the location of axes needs to be adjusted. For subplots, this can be done by adjusting the subplot params (*Move the edge of an axes to make room for tick labels*). Matplotlib v1.1 introduced *Figure.tight_layout* that does this automatically for you.

```
fig, ax = plt.subplots()
example_plot(ax, fontsize=24)
plt.tight_layout()
```



Note that *matplotlib.pyplot.tight_layout()* will only adjust the subplot params when it is called. In order to perform this adjustment each time the figure is redrawn, you can call `fig.set_tight_layout(True)`, or, equivalently, set `rcParams["figure.autolayout"]` (default: `False`) to `True`.

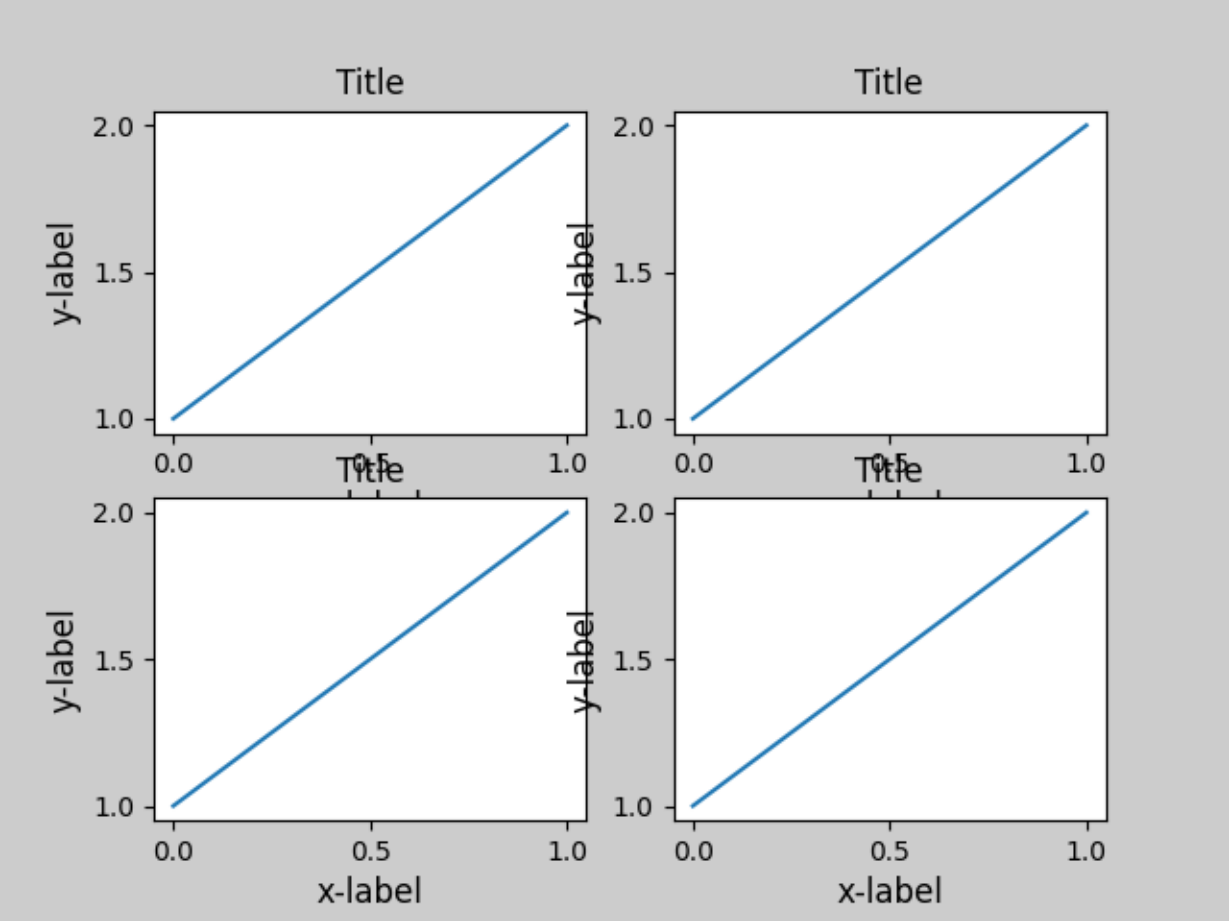
When you have multiple subplots, often you see labels of different axes overlapping each other.

```
plt.close('all')

fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(nrows=2, ncols=2)
example_plot(ax1)
example_plot(ax2)
example_plot(ax3)
```

(continues on next page)

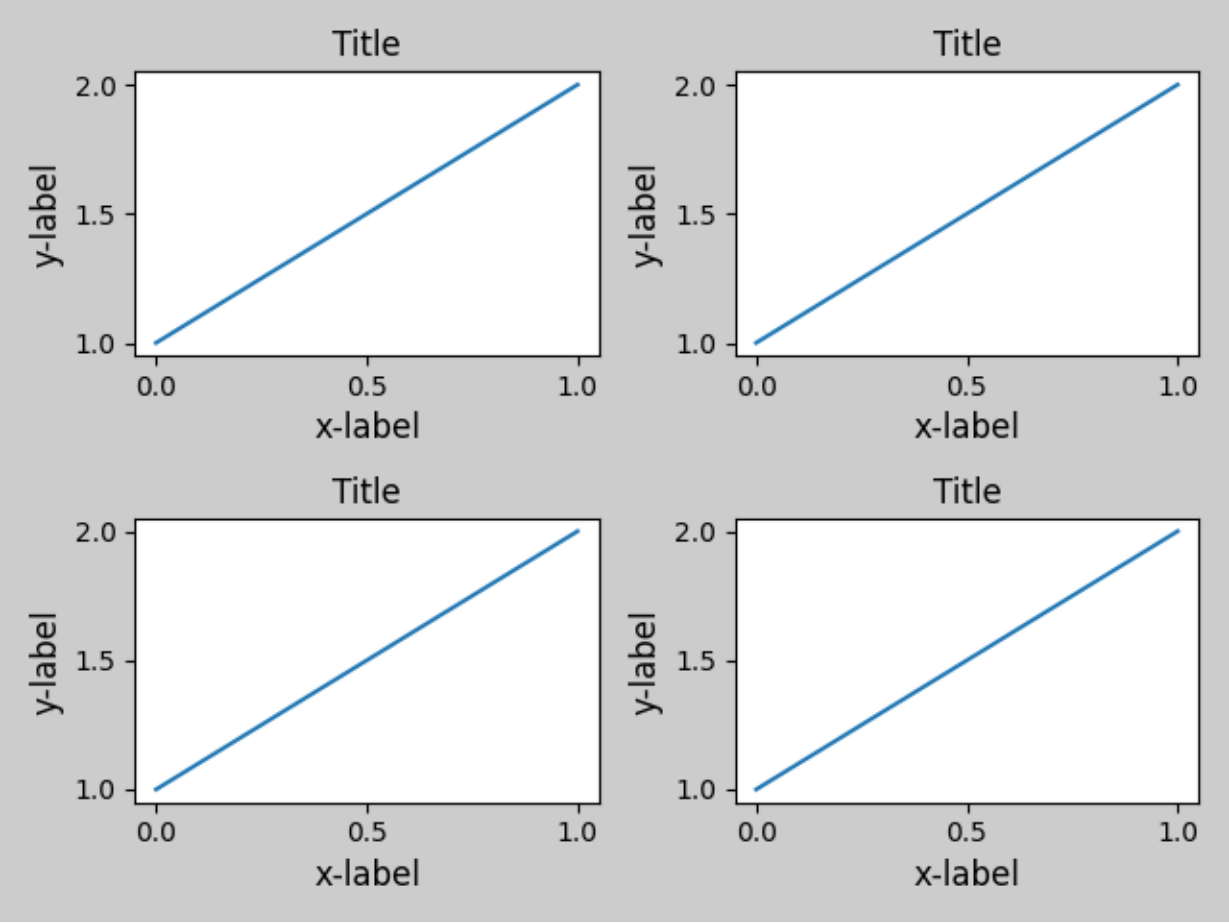
(continued from previous page)

```
example_plot(ax4)
```



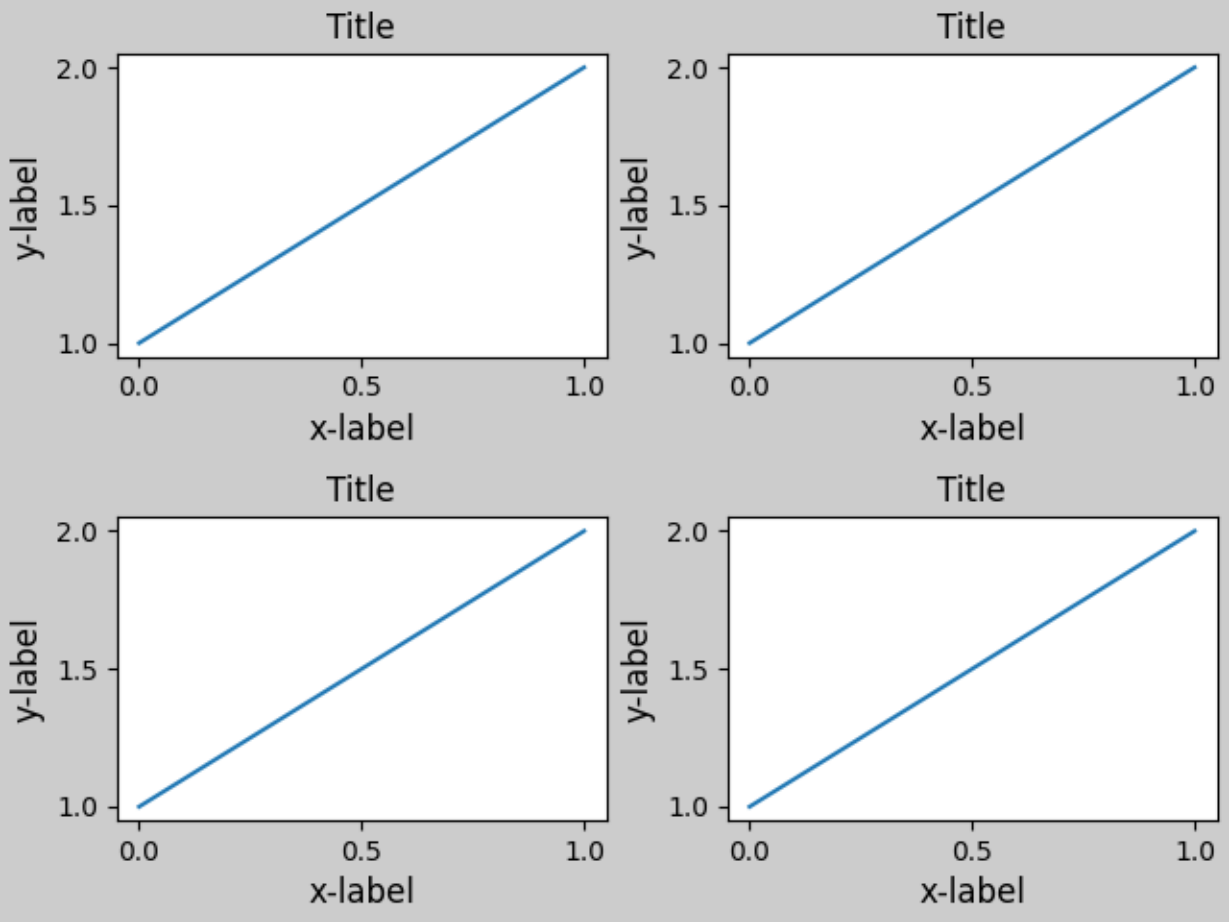
tight_layout() will also adjust spacing between subplots to minimize the overlaps.

```
fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(nrows=2, ncols=2)
example_plot(ax1)
example_plot(ax2)
example_plot(ax3)
example_plot(ax4)
plt.tight_layout()
```



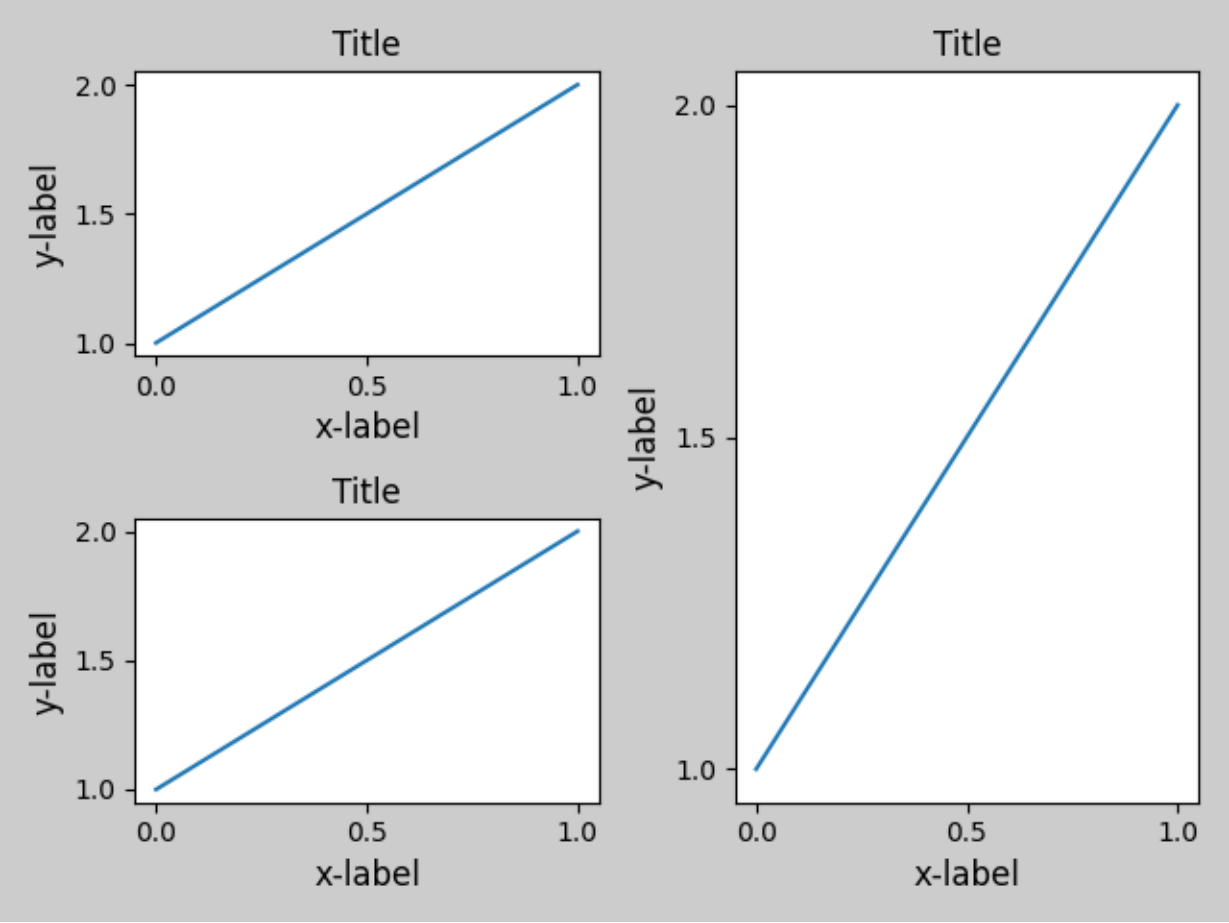
tight_layout() can take keyword arguments of *pad*, *w_pad* and *h_pad*. These control the extra padding around the figure border and between subplots. The pads are specified in fraction of fontsize.

```
fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(nrows=2, ncols=2)
example_plot(ax1)
example_plot(ax2)
example_plot(ax3)
example_plot(ax4)
plt.tight_layout(pad=0.4, w_pad=0.5, h_pad=1.0)
```



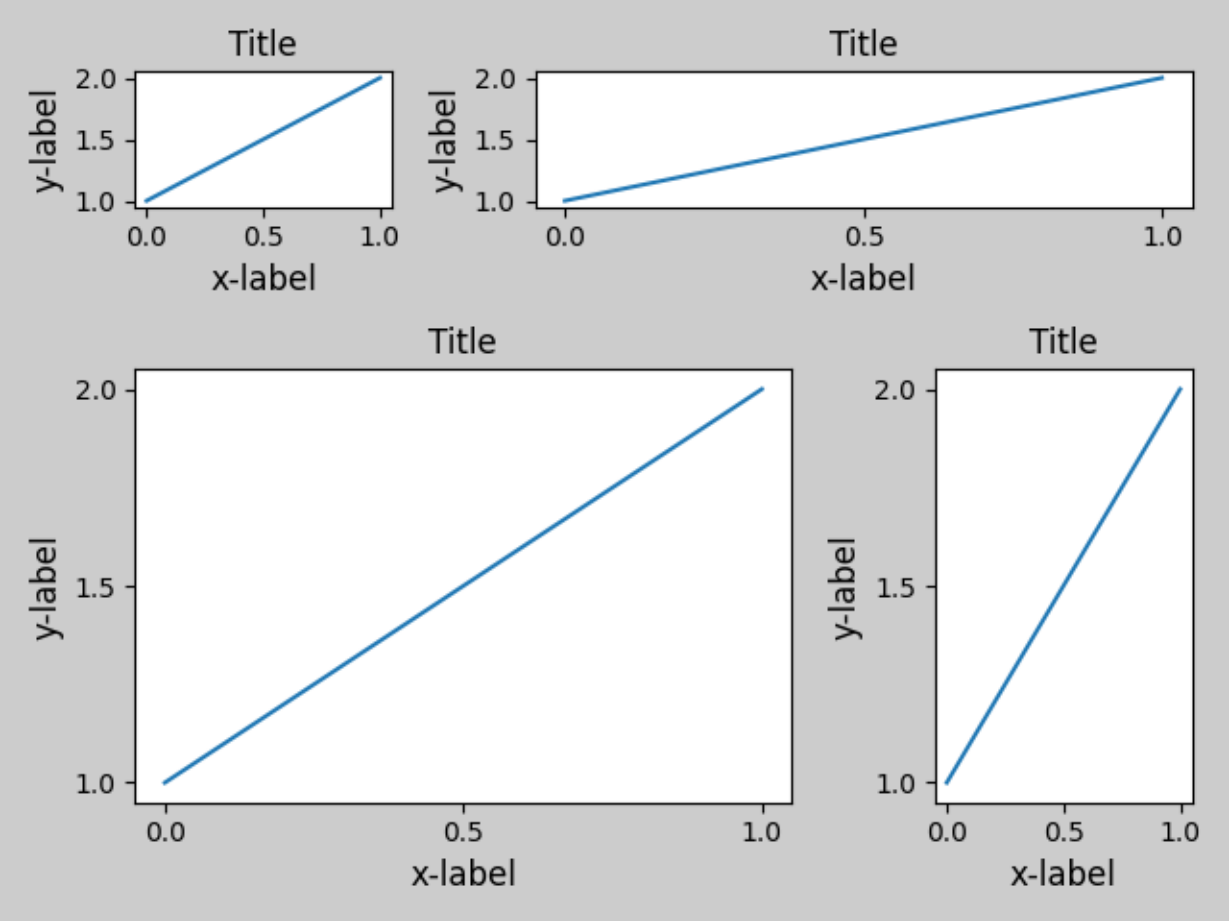
tight_layout() will work even if the sizes of subplots are different as far as their grid specification is compatible. In the example below, *ax1* and *ax2* are subplots of a 2x2 grid, while *ax3* is of a 1x2 grid.

```
plt.close('all')
fig = plt.figure()

ax1 = plt.subplot(221)
ax2 = plt.subplot(223)
ax3 = plt.subplot(122)

example_plot(ax1)
example_plot(ax2)
example_plot(ax3)

plt.tight_layout()
```



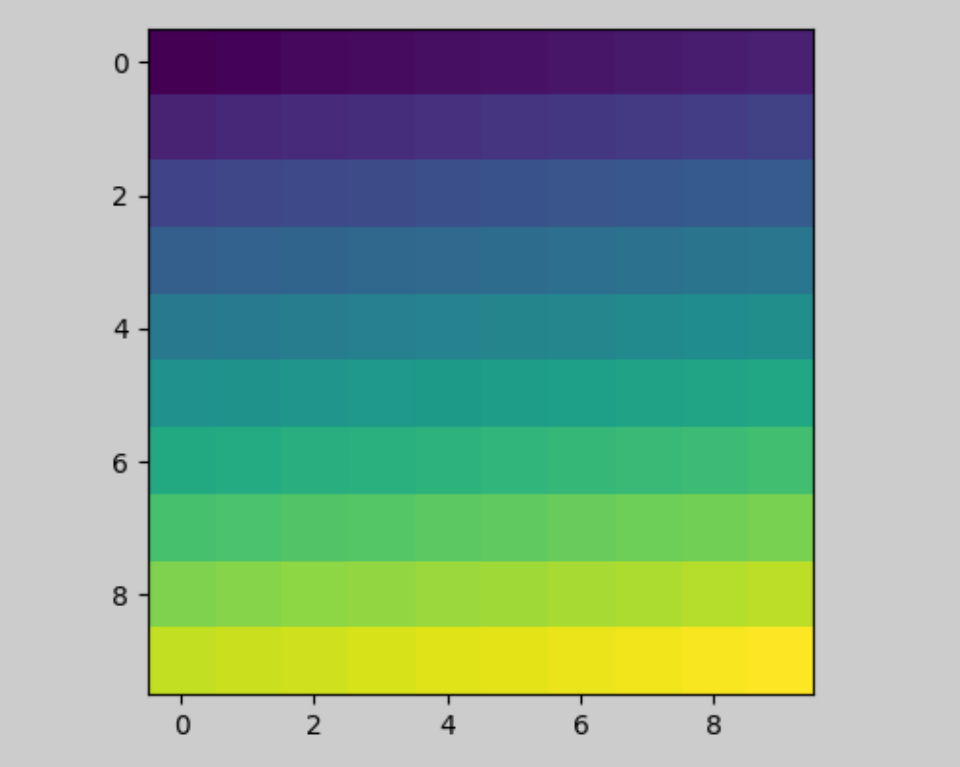
It works with subplots created with *subplot2grid()*. In general, subplots created from the gridspec (*Customizing Figure Layouts Using GridSpec and Other Functions*) will work.

```
plt.close('all')
fig = plt.figure()

ax1 = plt.subplot2grid((3, 3), (0, 0))
ax2 = plt.subplot2grid((3, 3), (0, 1), colspan=2)
ax3 = plt.subplot2grid((3, 3), (1, 0), colspan=2, rowspan=2)
ax4 = plt.subplot2grid((3, 3), (1, 2), rowspan=2)

example_plot(ax1)
example_plot(ax2)
example_plot(ax3)
example_plot(ax4)

plt.tight_layout()
```



Although not thoroughly tested, it seems to work for subplots with aspect != "auto" (e.g., axes with images).

```
arr = np.arange(100).reshape((10, 10))

plt.close('all')
fig = plt.figure(figsize=(5, 4))

ax = plt.subplot(111)
im = ax.imshow(arr, interpolation="none")

plt.tight_layout()
```



Caveats

- *tight_layout* considers all artists on the axes by default. To remove an artist from the layout calculation you can call *Artist.set_in_layout*.
- `tight_layout` assumes that the extra space needed for artists is independent of the original location of axes. This is often true, but there are rare cases where it is not.
- `pad=0` can clip some texts by a few pixels. This may be a bug or a limitation of the current algorithm and it is not clear why it happens. Meanwhile, use of pad larger than 0.3 is recommended.

Use with GridSpec

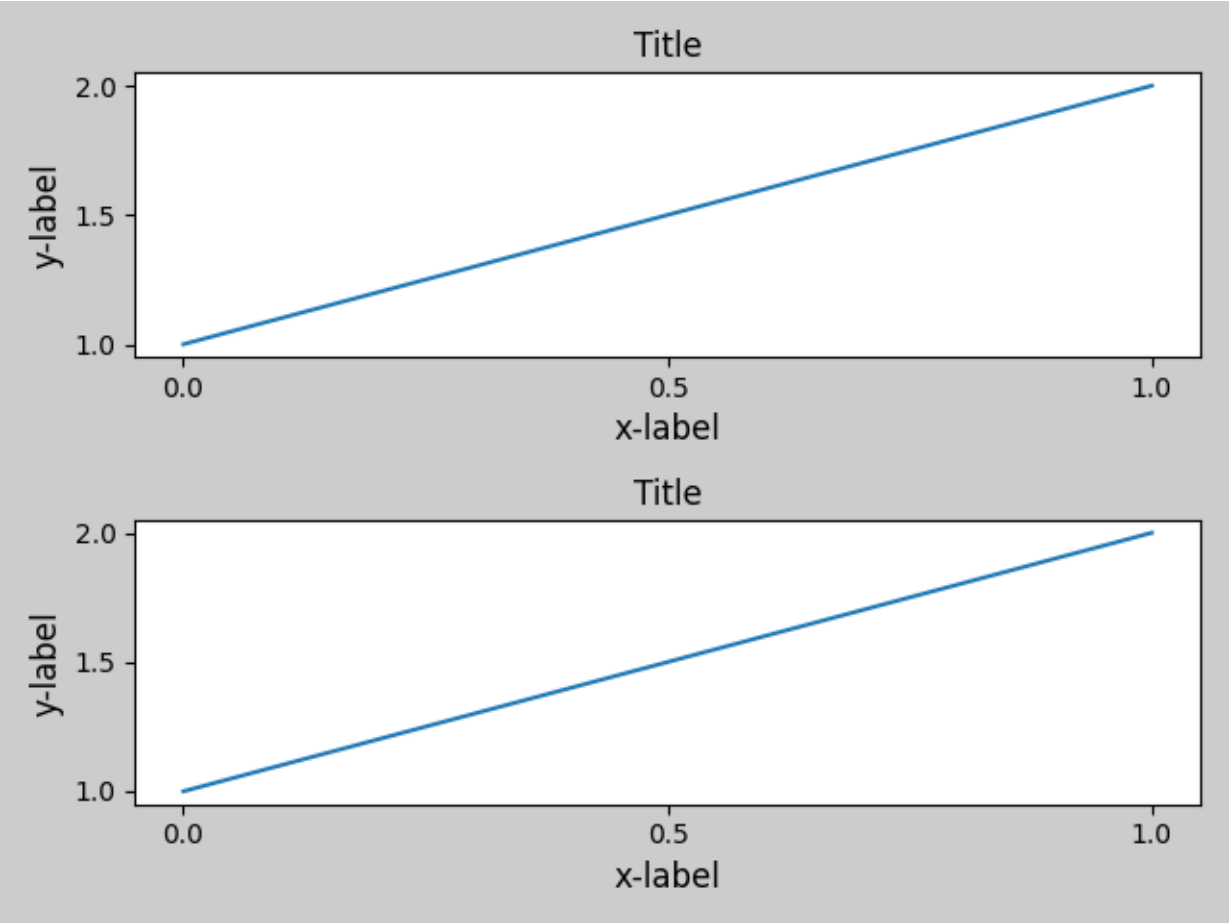
GridSpec has its own *GridSpec.tight_layout* method (the pyplot api *pyplot.tight_layout* also works).

```
import matplotlib.gridspec as gridspec

plt.close('all')
fig = plt.figure()

gs1 = gridspec.GridSpec(2, 1)
```

(continues on next page)

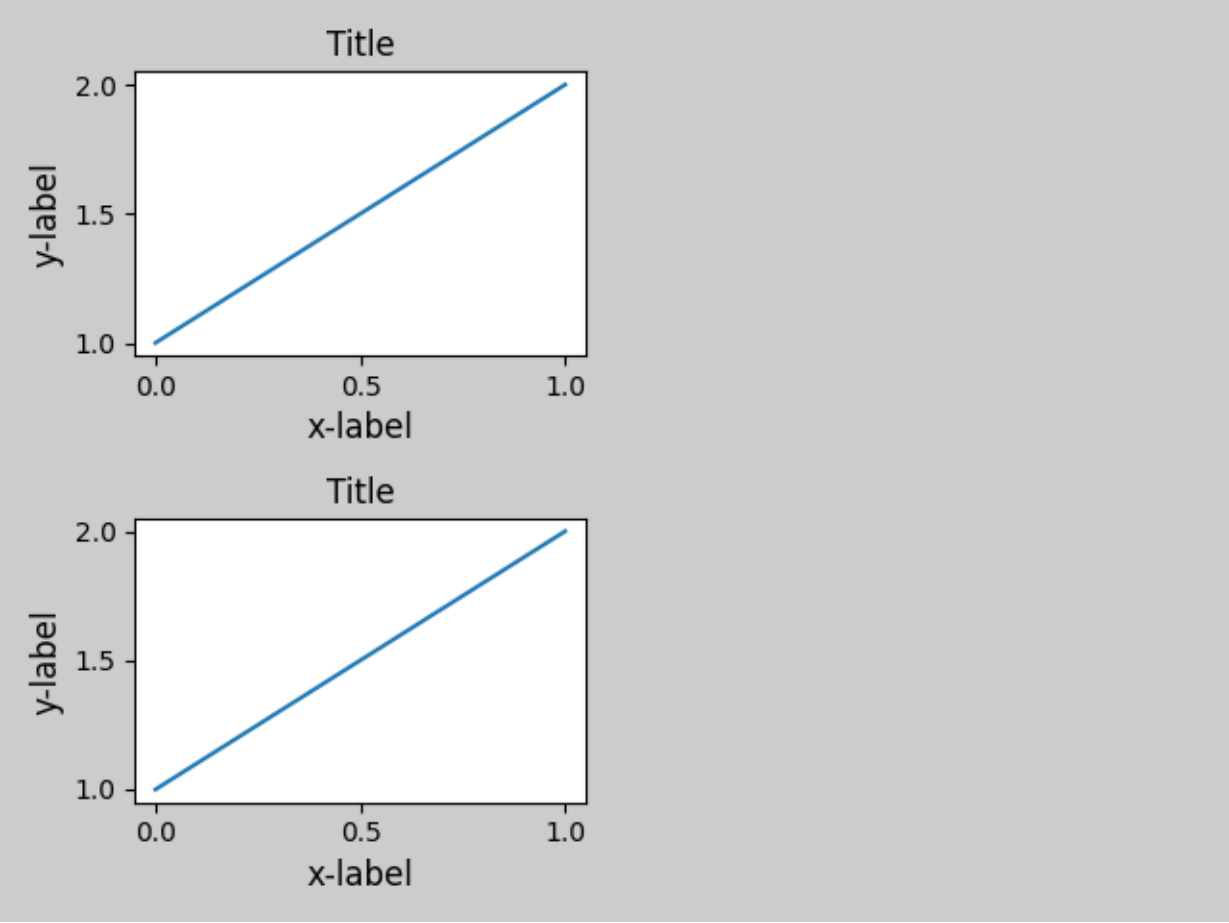
(continued from previous page)

```
ax1 = fig.add_subplot(gs1[0])
ax2 = fig.add_subplot(gs1[1])

example_plot(ax1)
example_plot(ax2)

gs1.tight_layout(fig)
```



You may provide an optional *rect* parameter, which specifies the bounding box that the subplots will be fit inside. The coordinates must be in normalized figure coordinates and the default is (0, 0, 1, 1).

```
fig = plt.figure()

gs1 = gridspec.GridSpec(2, 1)
ax1 = fig.add_subplot(gs1[0])
ax2 = fig.add_subplot(gs1[1])

example_plot(ax1)
example_plot(ax2)

gs1.tight_layout(fig, rect=[0, 0, 0.5, 1])
```



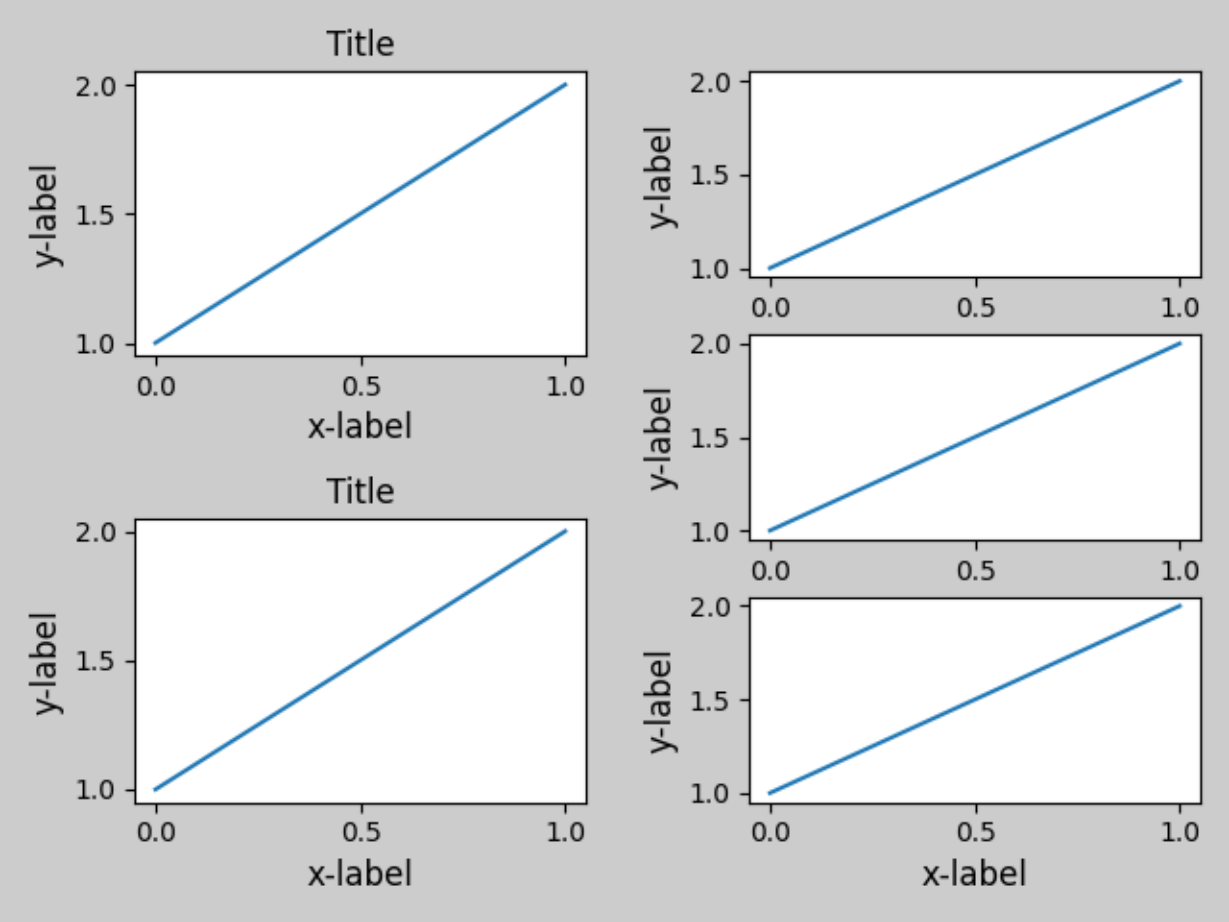
For example, this can be used for a figure with multiple gridspecs.

```
fig = plt.figure()

gs1 = gridspec.GridSpec(2, 1)
ax1 = fig.add_subplot(gs1[0])
ax2 = fig.add_subplot(gs1[1])

example_plot(ax1)
example_plot(ax2)

gs1.tight_layout(fig, rect=[0, 0, 0.5, 1])

gs2 = gridspec.GridSpec(3, 1)

for ss in gs2:
    ax = fig.add_subplot(ss)
    example_plot(ax)
    ax.set_title("")
    ax.set_xlabel("")

ax.set_xlabel("x-label", fontsize=12)
```

(continues on next page)

(continued from previous page)

```
gs2.tight_layout(fig, rect=[0.5, 0, 1, 1], h_pad=0.5)

# We may try to match the top and bottom of two grids ::
top = min(gs1.top, gs2.top)
bottom = max(gs1.bottom, gs2.bottom)

gs1.update(top=top, bottom=bottom)
gs2.update(top=top, bottom=bottom)
plt.show()
```



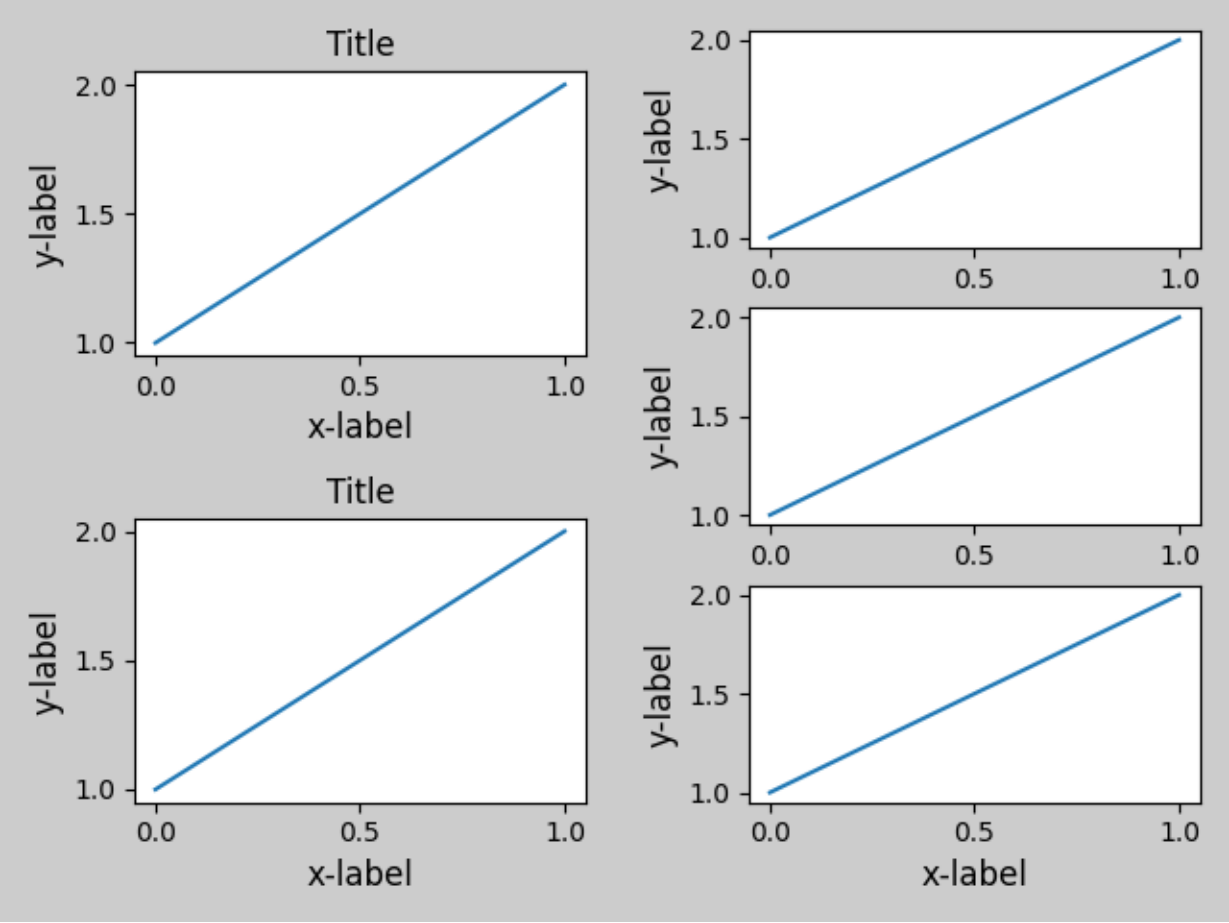
Out:

```
/root/matplotlib/tutorials/intermediate/tight_layout_guide.py:227: UserWarning: This␣
↪figure includes Axes that are not compatible with tight_layout, so results might be␣
↪incorrect.
  gs2.tight_layout(fig, rect=[0.5, 0, 1, 1], h_pad=0.5)
```

While this should be mostly good enough, adjusting top and bottom may require adjustment of hspace also. To update hspace & vspace, we call *GridSpec.tight_layout* again with updated rect argument. Note that the rect argument specifies the area including the ticklabels, etc. Thus, we will increase the bottom (which is 0 for the normal case) by the difference between the *bottom* from above and the bottom of

each gridspec. Same thing for the top.

```
fig = plt.gcf()

gs1 = gridspec.GridSpec(2, 1)
ax1 = fig.add_subplot(gs1[0])
ax2 = fig.add_subplot(gs1[1])

example_plot(ax1)
example_plot(ax2)

gs1.tight_layout(fig, rect=[0, 0, 0.5, 1])

gs2 = gridspec.GridSpec(3, 1)

for ss in gs2:
    ax = fig.add_subplot(ss)
    example_plot(ax)
    ax.set_title("")
    ax.set_xlabel("")

ax.set_xlabel("x-label", fontsize=12)

gs2.tight_layout(fig, rect=[0.5, 0, 1, 1], h_pad=0.5)

top = min(gs1.top, gs2.top)
bottom = max(gs1.bottom, gs2.bottom)

gs1.update(top=top, bottom=bottom)
gs2.update(top=top, bottom=bottom)

top = min(gs1.top, gs2.top)
bottom = max(gs1.bottom, gs2.bottom)

gs1.tight_layout(fig, rect=[None, 0 + (bottom-gs1.bottom),
                            0.5, 1 - (gs1.top-top)])
gs2.tight_layout(fig, rect=[0.5, 0 + (bottom-gs2.bottom),
                            None, 1 - (gs2.top-top)],
                 h_pad=0.5)
```



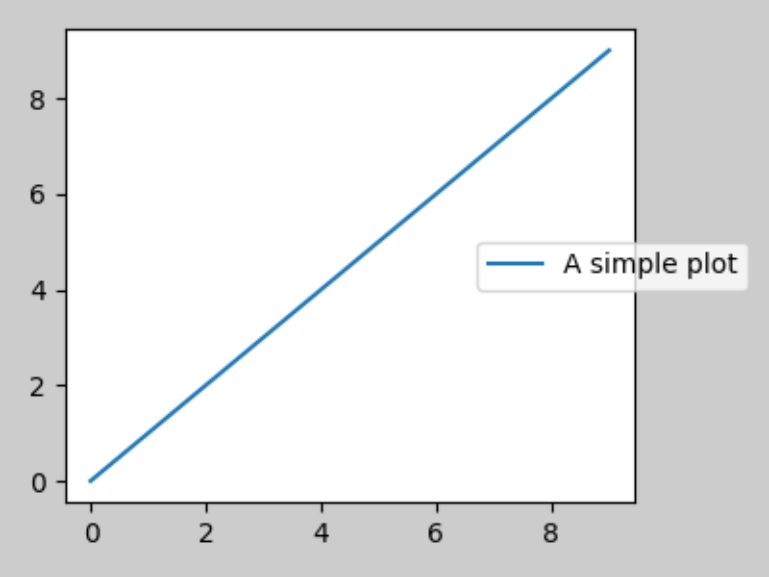
Out:

```
/root/matplotlib/tutorials/intermediate/tight_layout_guide.py:267: UserWarning: This␣
↪figure includes Axes that are not compatible with tight_layout, so results might be␣
↪incorrect.
  gs2.tight_layout(fig, rect=[0.5, 0, 1, 1], h_pad=0.5)
/root/matplotlib/tutorials/intermediate/tight_layout_guide.py:278: UserWarning: This␣
↪figure includes Axes that are not compatible with tight_layout, so results might be␣
↪incorrect.
  gs1.tight_layout(fig, rect=[None, 0 + (bottom-gs1.bottom),
/root/matplotlib/tutorials/intermediate/tight_layout_guide.py:280: UserWarning: This␣
↪figure includes Axes that are not compatible with tight_layout, so results might be␣
↪incorrect.
  gs2.tight_layout(fig, rect=[0.5, 0 + (bottom-gs2.bottom),
```

Legends and Annotations

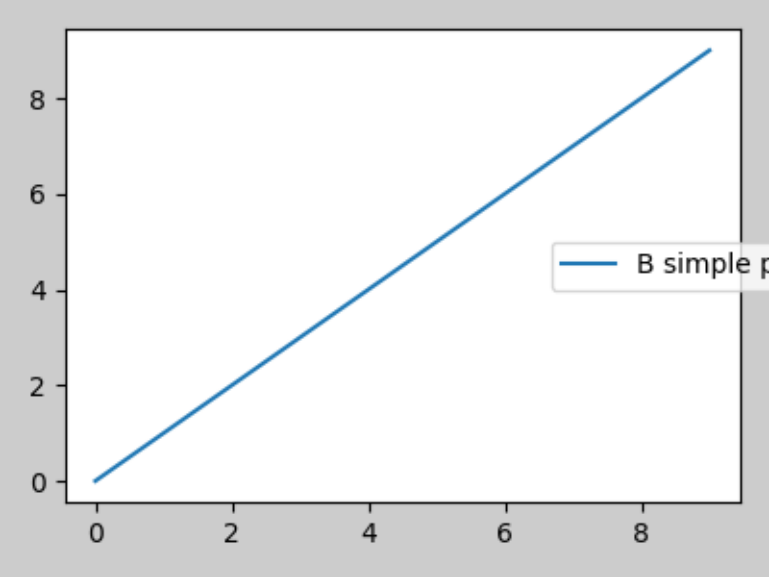
Pre Matplotlib 2.2, legends and annotations were excluded from the bounding box calculations that decide the layout. Subsequently these artists were added to the calculation, but sometimes it is undesirable to include them. For instance in this case it might be good to have the axes shring a bit to make room for the legend:

```
fig, ax = plt.subplots(figsize=(4, 3))
lines = ax.plot(range(10), label='A simple plot')
ax.legend(bbox_to_anchor=(0.7, 0.5), loc='center left',)
fig.tight_layout()
plt.show()
```



However, sometimes this is not desired (quite often when using `fig.savefig('outname.png', bbox_inches='tight')`). In order to remove the legend from the bounding box calculation, we simply set its bounding `leg.set_in_layout(False)` and the legend will be ignored.

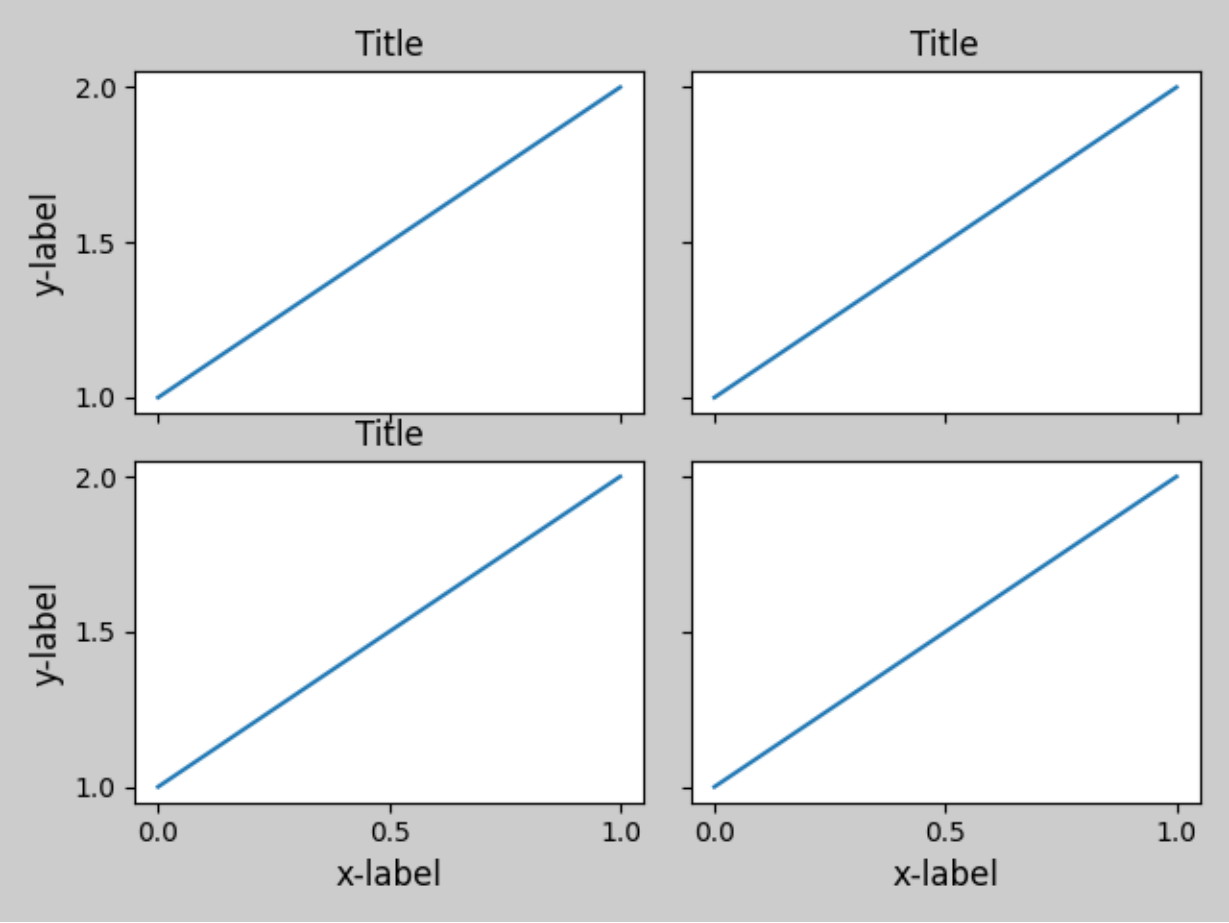
```
fig, ax = plt.subplots(figsize=(4, 3))
lines = ax.plot(range(10), label='B simple plot')
leg = ax.legend(bbox_to_anchor=(0.7, 0.5), loc='center left',)
leg.set_in_layout(False)
fig.tight_layout()
plt.show()
```



Use with AxesGrid1

While limited, *mpl_toolkits.axes_grid1* is also supported.

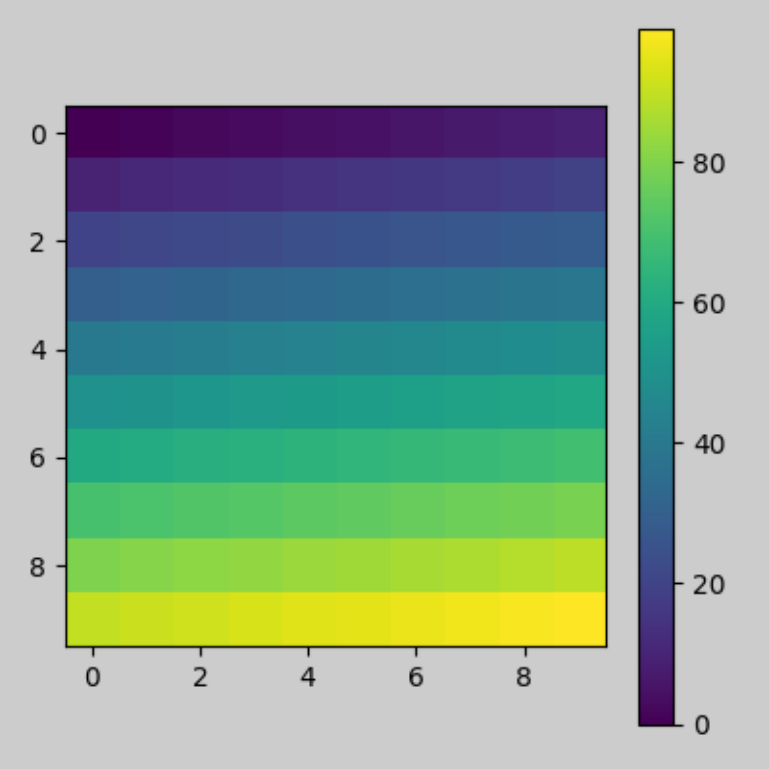
```
from mpl_toolkits.axes_grid1 import Grid

plt.close('all')
fig = plt.figure()
grid = Grid(fig, rect=111, nrows_ncols=(2, 2),
            axes_pad=0.25, label_mode='L',
            )

for ax in grid:
    example_plot(ax)
ax.title.set_visible(False)

plt.tight_layout()
```



Colorbar

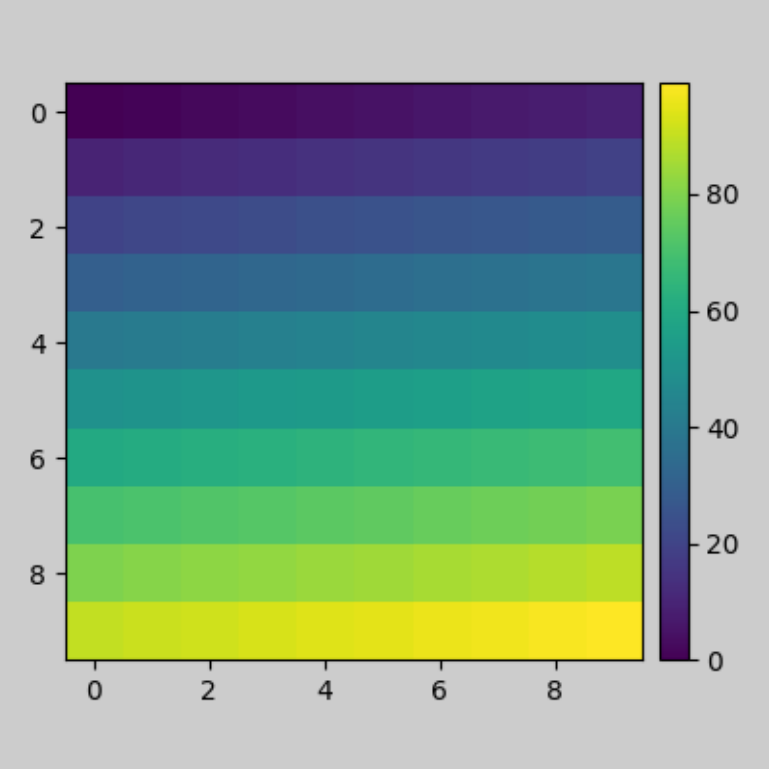
If you create a colorbar with *Figure.colorbar*, the created colorbar is drawn in a Subplot as long as the parent axes is also a Subplot, so *Figure.tight_layout* will work.

```
plt.close('all')
arr = np.arange(100).reshape((10, 10))
fig = plt.figure(figsize=(4, 4))
im = plt.imshow(arr, interpolation="none")

plt.colorbar(im)

plt.tight_layout()
```



Another option is to use AxesGrid1 toolkit to explicitly create an axes for colorbar.

```
from mpl_toolkits.axes_grid1 import make_axes_locatable

plt.close('all')
arr = np.arange(100).reshape((10, 10))
fig = plt.figure(figsize=(4, 4))
im = plt.imshow(arr, interpolation="none")

divider = make_axes_locatable(plt.gca())
cax = divider.append_axes("right", "5%", pad="3%")
plt.colorbar(im, cax=cax)

plt.tight_layout()
```



Total running time of the script: (0 minutes 5.914 seconds)

2.2.7 *origin* and *extent* in `imshow`

imshow() allows you to render an image (either a 2D array which will be color-mapped (based on *norm* and *cmap*) or a 3D RGB(A) array which will be used as-is) to a rectangular region in dataspace. The orientation of the image in the final rendering is controlled by the *origin* and *extent* kwargs (and attributes on the resulting *AxesImage* instance) and the data limits of the axes.

The *extent* kwarg controls the bounding box in data coordinates that the image will fill specified as `(left, right, bottom, top)` in **data coordinates**, the *origin* kwarg controls how the image fills that bounding box, and the orientation in the final rendered image is also affected by the axes limits.

Hint: Most of the code below is used for adding labels and informative text to the plots. The described effects of *origin* and *extent* can be seen in the plots without the need to follow all code details.

For a quick understanding, you may want to skip the code details below and directly continue with the discussion of the results.

```
import numpy as np
import matplotlib.pyplot as plt
```

(continues on next page)

(continued from previous page)

```
from matplotlib.gridspec import GridSpec


def index_to_coordinate(index, extent, origin):
    """Return the pixel center of an index."""
    left, right, bottom, top = extent

    hshift = 0.5 * np.sign(right - left)
    left, right = left + hshift, right - hshift
    vshift = 0.5 * np.sign(top - bottom)
    bottom, top = bottom + vshift, top - vshift

    if origin == 'upper':
        bottom, top = top, bottom

    return {
        "[0, 0]": (left, bottom),
        "[M', 0]": (left, top),
        "[0, N']": (right, bottom),
        "[M', N']": (right, top),
    }[index]


def get_index_label_pos(index, extent, origin, inverted_xindex):
    """
    Return the desired position and horizontal alignment of an index label.
    """
    if extent is None:
        extent = lookup_extent(origin)
    left, right, bottom, top = extent
    x, y = index_to_coordinate(index, extent, origin)

    is_x0 = index[-2:] == "0]"
    halign = 'left' if is_x0 ^ inverted_xindex else 'right'
    hshift = 0.5 * np.sign(left - right)
    x += hshift * (1 if is_x0 else -1)
    return x, y, halign


def get_color(index, data, cmap):
    """Return the data color of an index."""
    val = {
        "[0, 0]": data[0, 0],
        "[0, N']": data[0, -1],
        "[M', 0]": data[-1, 0],
        "[M', N']": data[-1, -1],
    }[index]
    return cmap(val / data.max())


def lookup_extent(origin):
```

(continues on next page)

(continued from previous page)

```
    """Return extent for label positioning when not given explicitly."""
    if origin == 'lower':
        return (-0.5, 6.5, -0.5, 5.5)
    else:
        return (-0.5, 6.5, 5.5, -0.5)


def set_extent_None_text(ax):
    ax.text(3, 2.5, 'equals\nextent=None', size='large',
            ha='center', va='center', color='w')


def plot_imshow_with_labels(ax, data, extent, origin, xlim, ylim):
    """Actually run ``imshow()`` and add extent and index labels."""
    im = ax.imshow(data, origin=origin, extent=extent)

    # extent labels (left, right, bottom, top)
    left, right, bottom, top = im.get_extent()
    if xlim is None or top > bottom:
        upper_string, lower_string = 'top', 'bottom'
    else:
        upper_string, lower_string = 'bottom', 'top'
    if ylim is None or left < right:
        port_string, starboard_string = 'left', 'right'
        inverted_xindex = False
    else:
        port_string, starboard_string = 'right', 'left'
        inverted_xindex = True
    bbox_kwargs = {'fc': 'w', 'alpha': .75, 'boxstyle': "round4"}
    ann_kwargs = {'xycoords': 'axes fraction',
                  'textcoords': 'offset points',
                  'bbox': bbox_kwargs}
    ax.annotate(upper_string, xy=(.5, 1), xytext=(0, -1),
                ha='center', va='top', **ann_kwargs)
    ax.annotate(lower_string, xy=(.5, 0), xytext=(0, 1),
                ha='center', va='bottom', **ann_kwargs)
    ax.annotate(port_string, xy=(0, .5), xytext=(1, 0),
                ha='left', va='center', rotation=90,
                **ann_kwargs)
    ax.annotate(starboard_string, xy=(1, .5), xytext=(-1, 0),
                ha='right', va='center', rotation=-90,
                **ann_kwargs)
    ax.set_title('origin: {origin}'.format(origin=origin))

    # index labels
    for index in ["[0, 0]", "[0, N']", "[M', 0]", "[M', N']"]:
        tx, ty, halign = get_index_label_pos(index, extent, origin,
                                             inverted_xindex)
        facecolor = get_color(index, data, im.get_cmap())
        ax.text(tx, ty, index, color='white', ha=halign, va='center',
                bbox={'boxstyle': 'square', 'facecolor': facecolor})
```

(continues on next page)

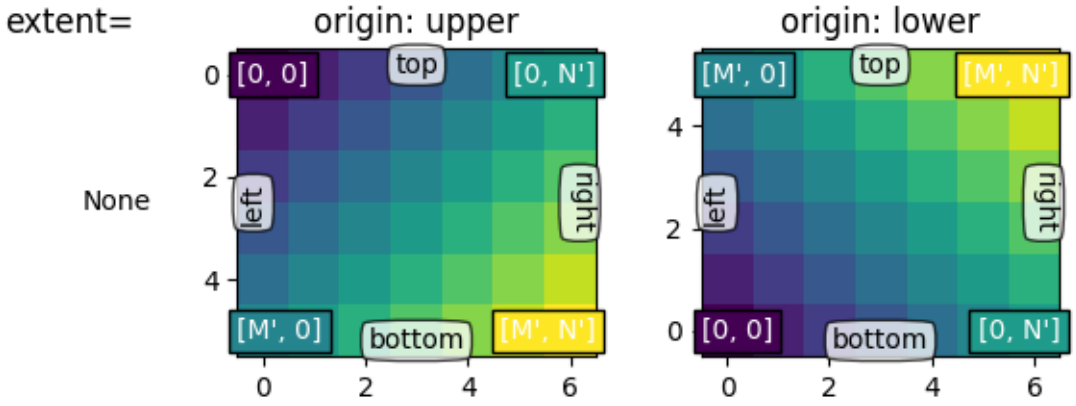
(continued from previous page)

```
    if xlim:
        ax.set_xlim(*xlim)
    if ylim:
        ax.set_ylim(*ylim)


def generate_imshow_demo_grid(extents, xlim=None, ylim=None):
    N = len(extents)
    fig = plt.figure(tight_layout=True)
    fig.set_size_inches(6, N * (11.25) / 5)
    gs = GridSpec(N, 5, figure=fig)

    columns = {'label': [fig.add_subplot(gs[j, 0]) for j in range(N)],
               'upper': [fig.add_subplot(gs[j, 1:3]) for j in range(N)],
               'lower': [fig.add_subplot(gs[j, 3:5]) for j in range(N)]}
    x, y = np.ogrid[0:6, 0:7]
    data = x + y

    for origin in ['upper', 'lower']:
        for ax, extent in zip(columns[origin], extents):
            plot_imshow_with_labels(ax, data, extent, origin, xlim, ylim)

    columns['label'][0].set_title('extent=')
    for ax, extent in zip(columns['label'], extents):
        if extent is None:
            text = 'None'
        else:
            left, right, bottom, top = extent
            text = (f'left: {left:0.1f}\nright: {right:0.1f}\n'
                    f'bottom: {bottom:0.1f}\ntop: {top:0.1f}\n')
        ax.text(1., .5, text, transform=ax.transAxes, ha='right', va='center')
        ax.axis('off')
    return columns
```

Default extent

First, let's have a look at the default `extent=None`

```
generate_imshow_demo_grid(extents=[None])
```



Out:

```
{'label': [<AxesSubplot:title={'center':'extent='}>], 'upper': [<AxesSubplot:title={
↪'center':'origin: upper'}>], 'lower': [<AxesSubplot:title={'center':'origin: lower'}>]}
```

Generally, for an array of shape (M, N), the first index runs along the vertical, the second index runs along the horizontal. The pixel centers are at integer positions ranging from 0 to `N' = N - 1` horizontally and from 0 to `M' = M - 1` vertically. *origin* determines how to the data is filled in the bounding box.

For `origin='lower'`:

- [0, 0] is at (left, bottom)
- [M', 0] is at (left, top)
- [0, N'] is at (right, bottom)
- [M', N'] is at (right, top)

`origin='upper'` reverses the vertical axes direction and filling:

- [0, 0] is at (left, top)
- [M', 0] is at (left, bottom)
- [0, N'] is at (right, top)
- [M', N'] is at (right, bottom)

In summary, the position of the [0, 0] index as well as the extent are influenced by *origin*:

origin	[0, 0] position	extent
upper	top left	`(-0.5, numcols-0.5, numrows-0.5, -0.5)`
lower	bottom left	`(-0.5, numcols-0.5, -0.5, numrows-0.5)`

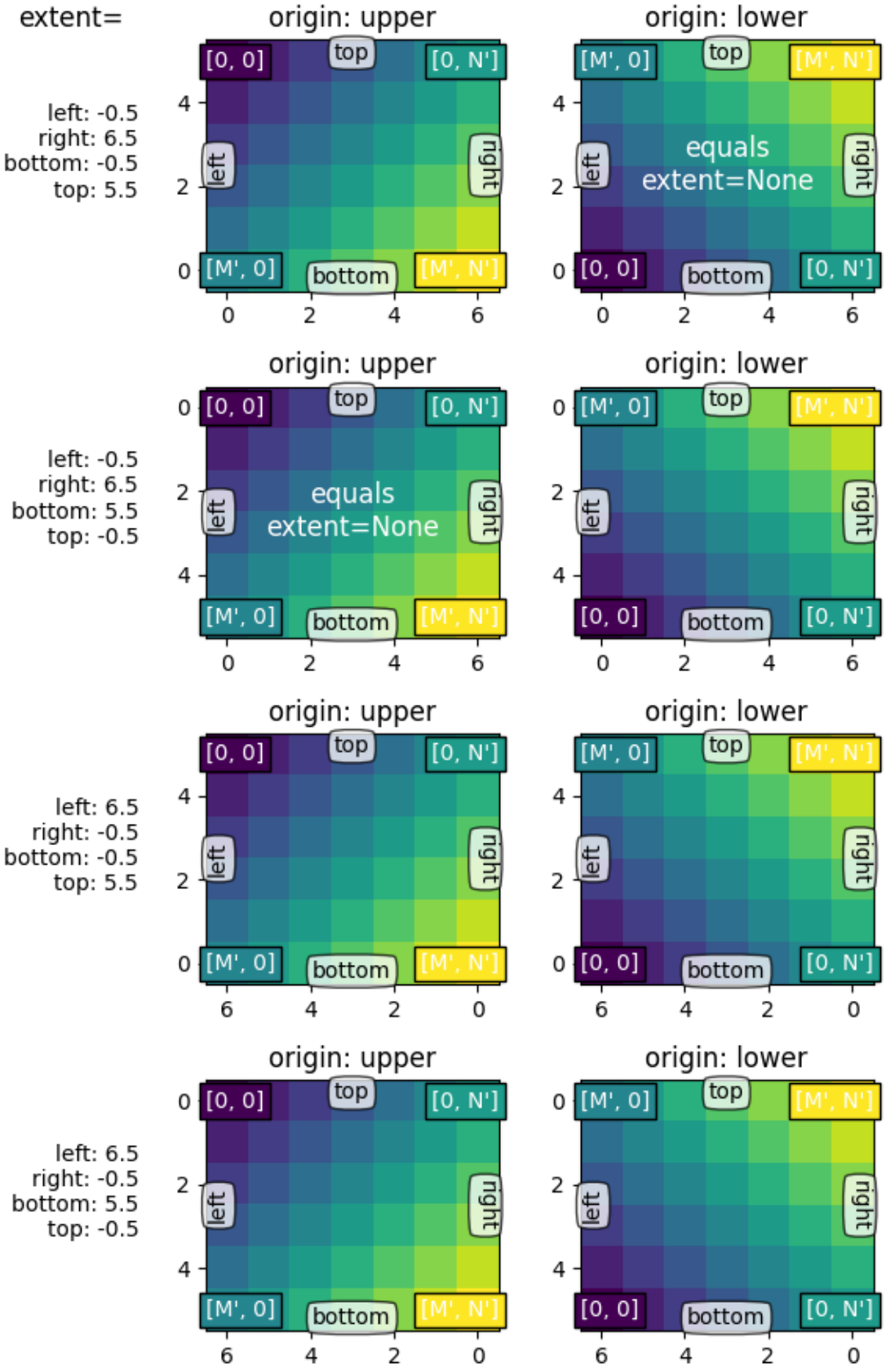
The default value of *origin* is set by `rcParams["image.origin"]` (default: `'upper'`) which defaults to `'upper'` to match the matrix indexing conventions in math and computer graphics image indexing conventions.

Explicit extent

By setting *extent* we define the coordinates of the image area. The underlying image data is interpolated/resampled to fill that area.

If the axes is set to autoscale, then the view limits of the axes are set to match the *extent* which ensures that the coordinate set by `(left, bottom)` is at the bottom left of the axes! However, this may invert the axis so they do not increase in the 'natural' direction.

```
extents = [(-0.5, 6.5, -0.5, 5.5),
           (-0.5, 6.5, 5.5, -0.5),
           (6.5, -0.5, -0.5, 5.5),
           (6.5, -0.5, 5.5, -0.5)]

columns = generate_imshow_demo_grid(extents)
set_extent_None_text(columns['upper'][1])
set_extent_None_text(columns['lower'][0])
```


extent=
origin: upper
origin: lower
left: -0.5
right: 6.5
bottom: -0.5
top: 5.5
left: -0.5
right: 6.5
bottom: 5.5
top: -0.5
left: 6.5
right: -0.5
bottom: -0.5
top: 5.5
left: 6.5
right: -0.5
bottom: 5.5
top: -0.5
equals
extent=None
[0, 0]
[0, N']
[M', 0]
[M', N']
top
bottom
left
right

Explicit extent and axes limits

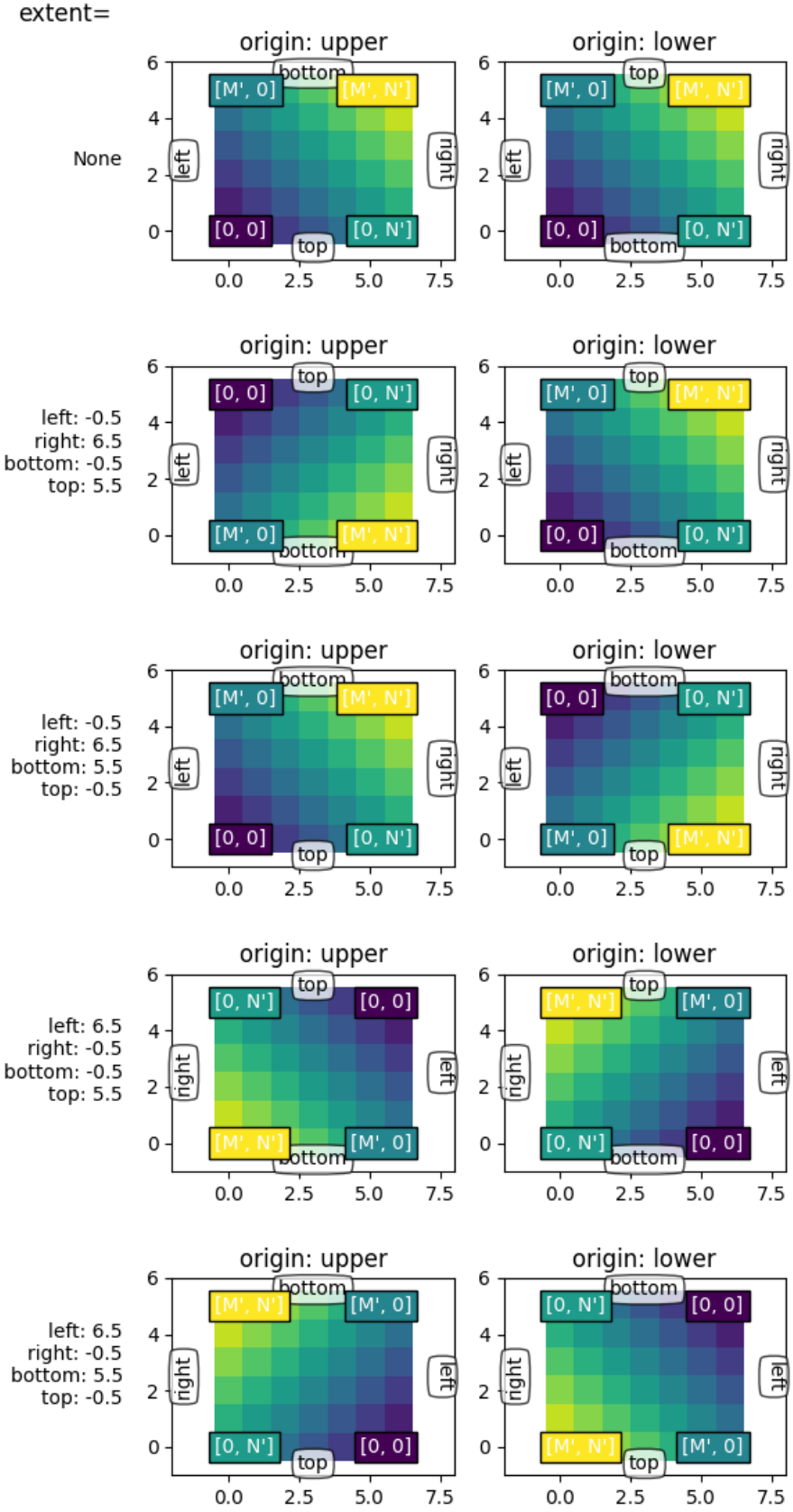
If we fix the axes limits by explicitly setting *set_xlim* / *set_ylim*, we force a certain size and orientation of the axes. This can decouple the 'left-right' and 'top-bottom' sense of the image from the orientation on the screen.

In the example below we have chosen the limits slightly larger than the extent (note the white areas within the Axes).

While we keep the extents as in the examples before, the coordinate (0, 0) is now explicitly put at the bottom left and values increase to up and to the right (from the viewer point of view). We can see that:

- The coordinate `(left, bottom)` anchors the image which then fills the box going towards the `(right, top)` point in data space.
- The first column is always closest to the 'left'.
- *origin* controls if the first row is closest to 'top' or 'bottom'.
- The image may be inverted along either direction.
- The 'left-right' and 'top-bottom' sense of the image may be uncoupled from the orientation on the screen.

```
generate_imshow_demo_grid(extents=[None] + extents,
                          xlim=(-2, 8), ylim=(-1, 6))

plt.show()
```


extent=
origin: upper
origin: lower
None
left: -0.5
right: 6.5
bottom: -0.5
top: 5.5
left: -0.5
right: 6.5
bottom: 5.5
top: -0.5
left: 6.5
right: -0.5
bottom: -0.5
top: 5.5
left: 6.5
right: -0.5
bottom: 5.5
top: -0.5
[M', 0]
[M', N']
[0, 0]
[0, N']
top
bottom
left
right
0.0
2.5
5.0
7.5
0
2
4
6

Total running time of the script: (0 minutes 2.456 seconds)

2.3 Advanced

These tutorials cover advanced topics for experienced Matplotlib users and developers.

2.3.1 Blitting tutorial

'Blitting' is a standard technique in raster graphics that, in the context of Matplotlib, can be used to (drastically) improve performance of interactive figures. For example, the *animation* and *widgets* modules use blitting internally. Here, we demonstrate how to implement your own blitting, outside of these classes.

The source of the performance gains is simply not re-doing work we do not have to. If the limits of an Axes have not changed, then there is no need to re-draw all of the ticks and tick-labels (particularly because text is one of the more expensive things to render).

The procedure to save our work is roughly:

- draw the figure, but exclude any artists marked as 'animated'
- save a copy of the RBGA buffer

In the future, to update the 'animated' artists we

- restore our copy of the RGBA buffer
- redraw only the animated artists
- show the resulting image on the screen

thus saving us from having to re-draw everything which is _not_ animated. One consequence of this procedure is that your animated artists are always drawn at a higher z-order than the static artists.

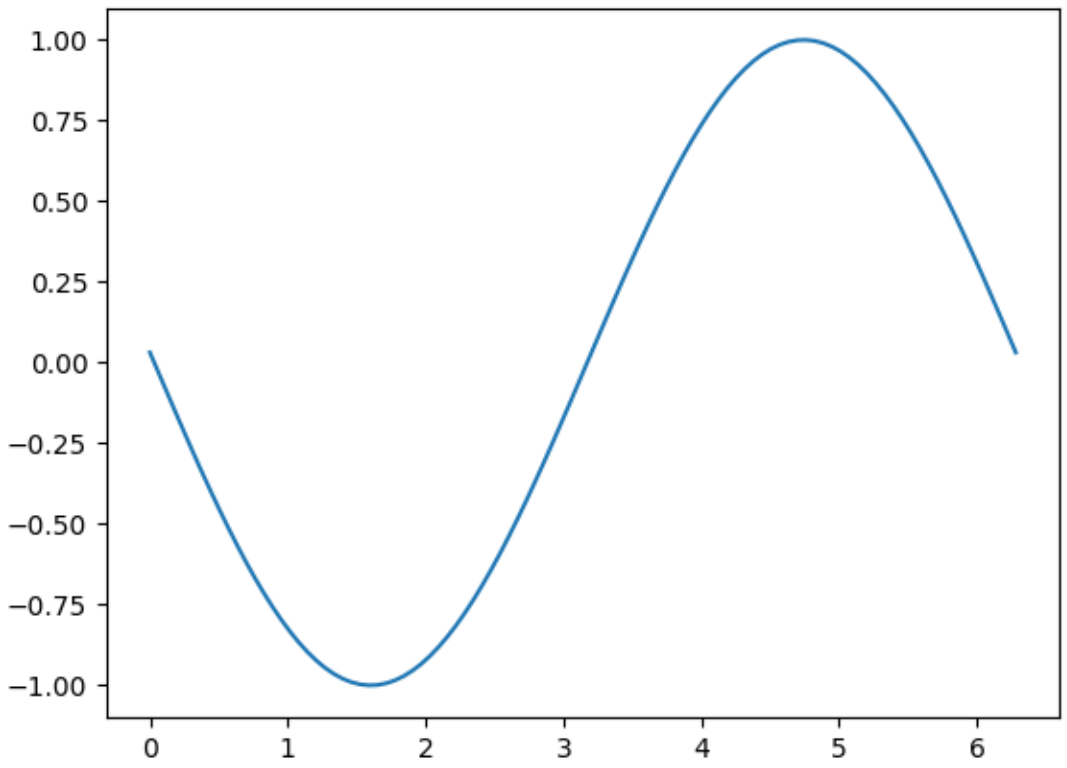
Not all backends support blitting. You can check if a given canvas does via the *FigureCanvasBase.supports_blit* property.

Warning: This code does not work with the OSX backend (but does work with other GUI backends on mac).

Minimal example

We can use the *FigureCanvasAgg* methods *copy_from_bbox* and *restore_region* in conjunction with setting `animated=True` on our artist to implement a minimal example that uses blitting to accelerate rendering

```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 2 * np.pi, 100)

fig, ax = plt.subplots()

# animated=True tells matplotlib to only draw the artist when we
# explicitly request it
(ln,) = ax.plot(x, np.sin(x), animated=True)

# make sure the window is raised, but the script keeps going
plt.show(block=False)

# stop to admire our empty window axes and ensure it is rendered at
# least once.
#
# We need to fully draw the figure at its final size on the screen
# before we continue on so that :
#  a) we have the correctly sized and drawn background to grab
#  b) we have a cached renderer so that ``ax.draw_artist`` works
# so we spin the event loop to let the backend process any pending operations
plt.pause(0.1)

# get copy of entire figure (everything inside fig.bbox) sans animated artist
bg = fig.canvas.copy_from_bbox(fig.bbox)
# draw the animated artist, this uses a cached renderer
ax.draw_artist(ln)
# show the result to the screen, this pushes the updated RGBA buffer from the
# renderer to the GUI framework so you can see it
fig.canvas.blit(fig.bbox)

for j in range(100):
    # reset the background back in the canvas state, screen unchanged
    fig.canvas.restore_region(bg)
    # update the artist, neither the canvas state nor the screen have changed
    ln.set_ydata(np.sin(x + (j / 100) * np.pi))
    # re-render the artist, updating the canvas state, but not the screen
    ax.draw_artist(ln)
    # copy the image to the GUI state, but screen might not changed yet
    fig.canvas.blit(fig.bbox)
    # flush any pending GUI events, re-painting the screen if needed
    fig.canvas.flush_events()
    # you can put a pause in if you want to slow things down
    # plt.pause(.1)
```



This example works and shows a simple animation, however because we are only grabbing the background once, if the size of the figure in pixels changes (due to either the size or dpi of the figure changing) , the background will be invalid and result in incorrect (but sometimes cool looking!) images. There is also a global variable and a fair amount of boiler plate which suggests we should wrap this in a class.

Class-based example

We can use a class to encapsulate the boilerplate logic and state of restoring the background, drawing the artists, and then blitting the result to the screen. Additionally, we can use the `'draw_event'` callback to capture a new background whenever a full re-draw happens to handle resizes correctly.

```
class BlitManager:
    def __init__(self, canvas, animated_artists=()):
        """
        Parameters
        ----------
        canvas : FigureCanvasAgg
            The canvas to work with, this only works for sub-classes of the Agg
            canvas which have the `~FigureCanvasAgg.copy_from_bbox` and
```

(continues on next page)

(continued from previous page)

```
            `~FigureCanvasAgg.restore_region` methods.

        animated_artists : Iterable[Artist]
            List of the artists to manage
        """
        self.canvas = canvas
        self._bg = None
        self._artists = []

        for a in animated_artists:
            self.add_artist(a)
        # grab the background on every draw
        self.cid = canvas.mpl_connect("draw_event", self.on_draw)

    def on_draw(self, event):
        """Callback to register with 'draw_event'."""
        cv = self.canvas
        if event is not None:
            if event.canvas != cv:
                raise RuntimeError
        self._bg = cv.copy_from_bbox(cv.figure.bbox)
        self._draw_animated()

    def add_artist(self, art):
        """
        Add an artist to be managed.

        Parameters
        ----------
        art : Artist

            The artist to be added.  Will be set to 'animated' (just
            to be safe).  *art* must be in the figure associated with
            the canvas this class is managing.

        """
        if art.figure != self.canvas.figure:
            raise RuntimeError
        art.set_animated(True)
        self._artists.append(art)

    def _draw_animated(self):
        """Draw all of the animated artists."""
        fig = self.canvas.figure
        for a in self._artists:
            fig.draw_artist(a)

    def update(self):
        """Update the screen with animated artists."""
        cv = self.canvas
        fig = cv.figure
```

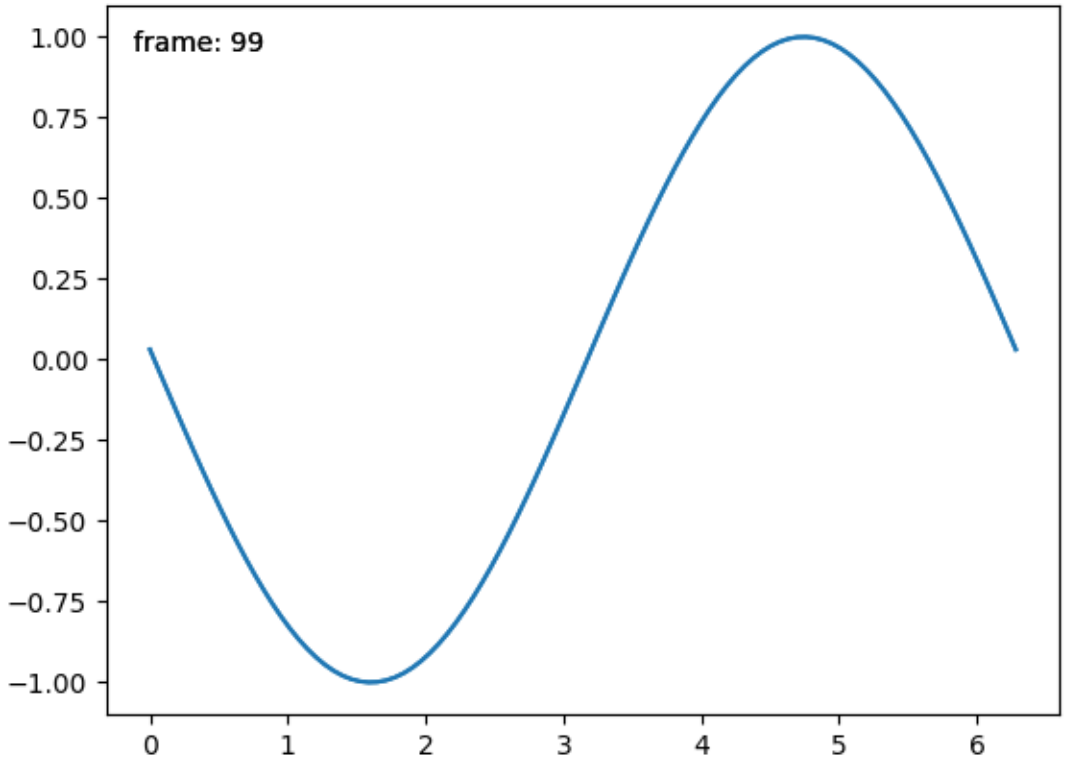
(continues on next page)

(continued from previous page)

```
        # paranoia in case we missed the draw event,
        if self._bg is None:
            self.on_draw(None)
        else:
            # restore the background
            cv.restore_region(self._bg)
            # draw all of the animated artists
            self._draw_animated()
            # update the GUI state
            cv.blit(fig.bbox)
        # let the GUI event loop process anything it has to do
        cv.flush_events()
```

Here is how we would use our class. This is a slightly more complicated example than the first case as we add a text frame counter as well.

```
# make a new figure
fig, ax = plt.subplots()
# add a line
(ln,) = ax.plot(x, np.sin(x), animated=True)
# add a frame number
fr_number = ax.annotate(
    "0",
    (0, 1),
    xycoords="axes fraction",
    xytext=(10, -10),
    textcoords="offset points",
    ha="left",
    va="top",
    animated=True,
)
bm = BlitManager(fig.canvas, [ln, fr_number])
# make sure our window is on the screen and drawn
plt.show(block=False)
plt.pause(.1)

for j in range(100):
    # update the artists
    ln.set_ydata(np.sin(x + (j / 100) * np.pi))
    fr_number.set_text("frame: {j}".format(j=j))
    # tell the blitting manager to do it's thing
    bm.update()
```



This class does not depend on *pyplot* and is suitable to embed into larger GUI application.

Total running time of the script: (0 minutes 1.143 seconds)

2.3.2 Path Tutorial

Defining paths in your Matplotlib visualization.

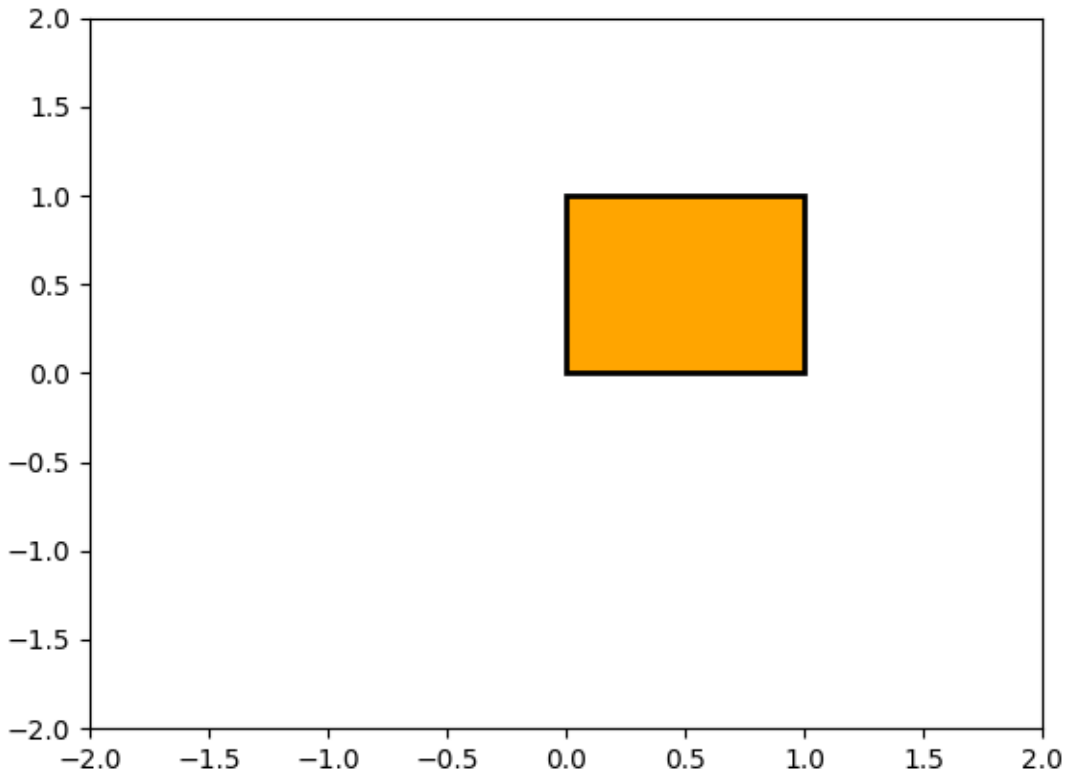
The object underlying all of the *matplotlib.patches* objects is the *Path*, which supports the standard set of moveto, lineto, curveto commands to draw simple and compound outlines consisting of line segments and splines. The `Path` is instantiated with a (N, 2) array of (x, y) vertices, and a N-length array of path codes. For example to draw the unit rectangle from (0, 0) to (1, 1), we could use this code:

```
import matplotlib.pyplot as plt
from matplotlib.path import Path
import matplotlib.patches as patches

verts = [
   (0., 0.),  # left, bottom
   (0., 1.),  # left, top
```

(continues on next page)

(continued from previous page)

```
    (1., 1.),  # right, top
    (1., 0.),  # right, bottom
    (0., 0.),  # ignored
]

codes = [
    Path.MOVETO,
    Path.LINETO,
    Path.LINETO,
    Path.LINETO,
    Path.CLOSEPOLY,
]

path = Path(verts, codes)

fig, ax = plt.subplots()
patch = patches.PathPatch(path, facecolor='orange', lw=2)
ax.add_patch(patch)
ax.set_xlim(-2, 2)
ax.set_ylim(-2, 2)
plt.show()
```



The following path codes are recognized

Code	Vertices	Description
STOP	1 (ignored)	A marker for the end of the entire path (currently not required and ignored).
MOVETO	1	Pick up the pen and move to the given vertex.
LINETO	1	Draw a line from the current position to the given vertex.
CURVE3	2: 1 control point, 1 end point	Draw a quadratic Bézier curve from the current position, with the given control point, to the given end point.
CURVE4	3: 2 control points, 1 end point	Draw a cubic Bézier curve from the current position, with the given control points, to the given end point.
CLOSEPOLY	1 (the point is ignored)	Draw a line segment to the start point of the current polyline.

Bézier example

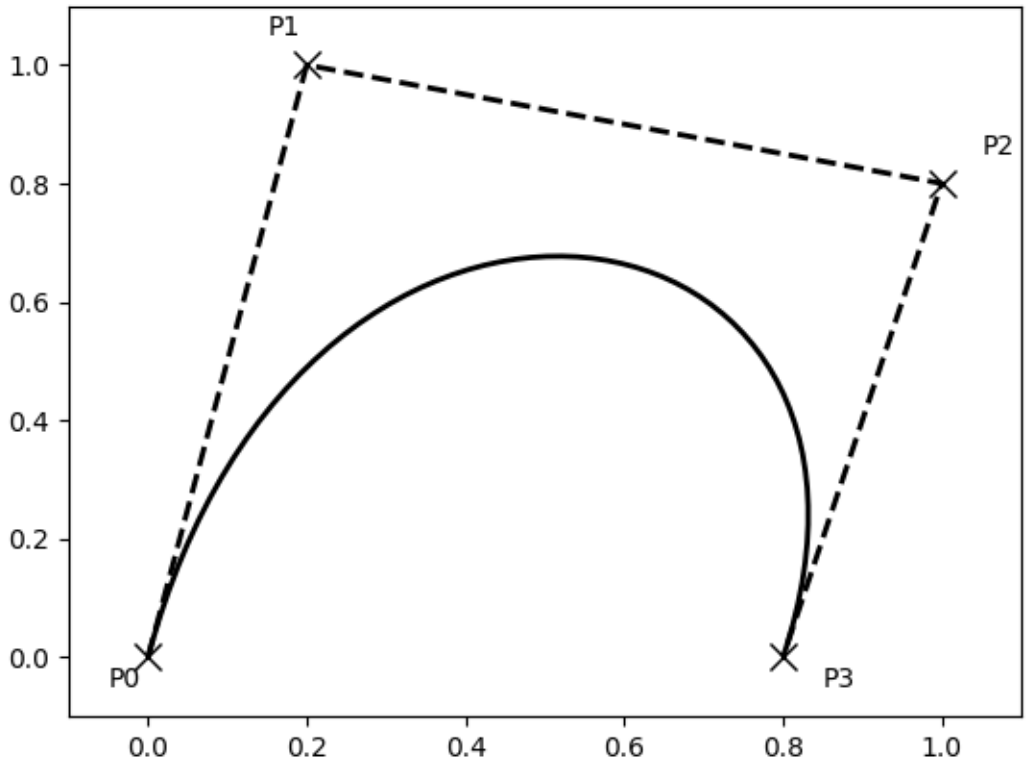
Some of the path components require multiple vertices to specify them: for example CURVE 3 is a bézier curve with one control point and one end point, and CURVE4 has three vertices for the two control points and the end point. The example below shows a CURVE4 Bézier spline -- the bézier curve will be contained in the convex hull of the start point, the two control points, and the end point

```
verts = [
   (0., 0.),   # P0
   (0.2, 1.),  # P1
   (1., 0.8),  # P2
   (0.8, 0.),  # P3
]

codes = [
    Path.MOVETO,
    Path.CURVE4,
    Path.CURVE4,
    Path.CURVE4,
]

path = Path(verts, codes)

fig, ax = plt.subplots()
patch = patches.PathPatch(path, facecolor='none', lw=2)
ax.add_patch(patch)

xs, ys = zip(*verts)
ax.plot(xs, ys, 'x--', lw=2, color='black', ms=10)

ax.text(-0.05, -0.05, 'P0')
ax.text(0.15, 1.05, 'P1')
ax.text(1.05, 0.85, 'P2')
```

(continues on next page)

(continued from previous page)

```
ax.text(0.85, -0.05, 'P3')

ax.set_xlim(-0.1, 1.1)
ax.set_ylim(-0.1, 1.1)
plt.show()
```



Compound paths

All of the simple patch primitives in matplotlib, Rectangle, Circle, Polygon, etc, are implemented with simple path. Plotting functions like *hist()* and *bar()*, which create a number of primitives, e.g., a bunch of Rectangles, can usually be implemented more efficiently using a compound path. The reason `bar` creates a list of rectangles and not a compound path is largely historical: the *Path* code is comparatively new and `bar` predates it. While we could change it now, it would break old code, so here we will cover how to create compound paths, replacing the functionality in bar, in case you need to do so in your own code for efficiency reasons, e.g., you are creating an animated bar plot.

We will make the histogram chart by creating a series of rectangles for each histogram bar: the rectangle width is the bin width and the rectangle height is the number of

datapoints in that bin. First we'll create some random normally distributed data and compute the histogram. Because numpy returns the bin edges and not centers, the length of `bins` is 1 greater than the length of `n` in the example below:

```
# histogram our data with numpy
data = np.random.randn(1000)
n, bins = np.histogram(data, 100)
```

We'll now extract the corners of the rectangles. Each of the `left`, `bottom`, etc, arrays below is `len(n)`, where `n` is the array of counts for each histogram bar:

```
# get the corners of the rectangles for the histogram
left = np.array(bins[:-1])
right = np.array(bins[1:])
bottom = np.zeros(len(left))
top = bottom + n
```

Now we have to construct our compound path, which will consist of a series of `MOVETO`, `LINETO` and `CLOSEPOLY` for each rectangle. For each rectangle, we need 5 vertices: 1 for the `MOVETO`, 3 for the `LINETO`, and 1 for the `CLOSEPOLY`. As indicated in the table above, the vertex for the closepoly is ignored but we still need it to keep the codes aligned with the vertices:

```
nverts = nrects*(1+3+1)
verts = np.zeros((nverts, 2))
codes = np.ones(nverts, int) * path.Path.LINETO
codes[0::5] = path.Path.MOVETO
codes[4::5] = path.Path.CLOSEPOLY
verts[0::5, 0] = left
verts[0::5, 1] = bottom
verts[1::5, 0] = left
verts[1::5, 1] = top
verts[2::5, 0] = right
verts[2::5, 1] = top
verts[3::5, 0] = right
verts[3::5, 1] = bottom
```

All that remains is to create the path, attach it to a *PathPatch*, and add it to our axes:

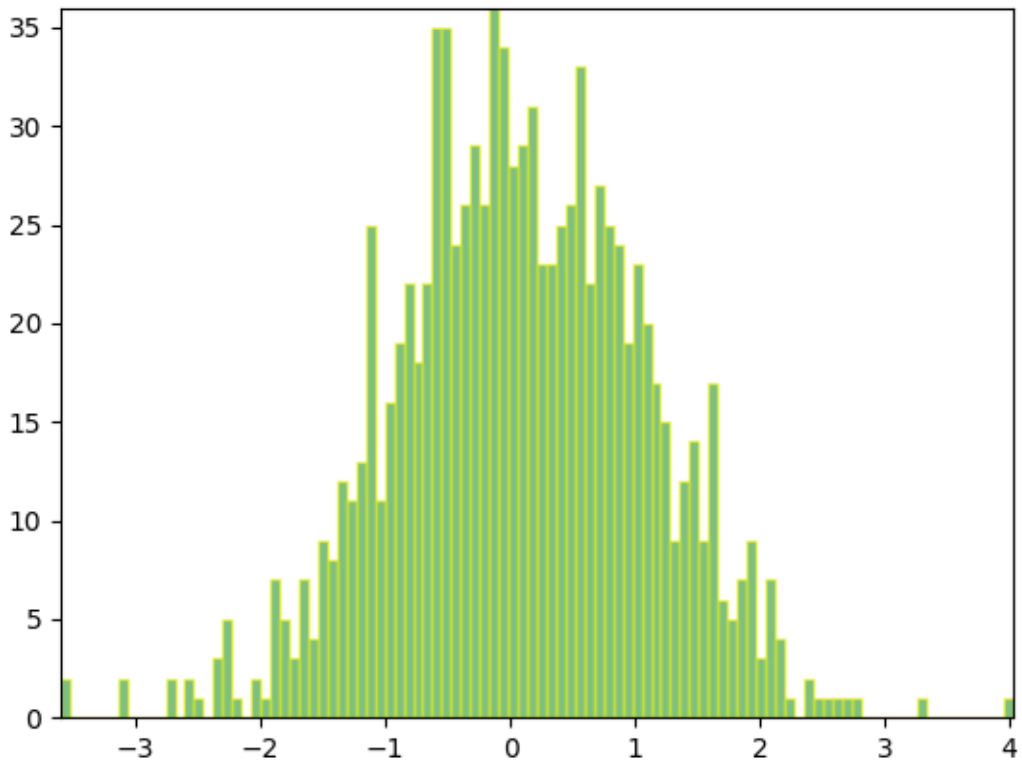
```
barpath = path.Path(verts, codes)
patch = patches.PathPatch(barpath, facecolor='green',
  edgecolor='yellow', alpha=0.5)
ax.add_patch(patch)
```

```
import numpy as np
import matplotlib.patches as patches
import matplotlib.path as path

fig, ax = plt.subplots()
# Fixing random state for reproducibility
np.random.seed(19680801)
```

(continues on next page)

(continued from previous page)

```
# histogram our data with numpy
data = np.random.randn(1000)
n, bins = np.histogram(data, 100)

# get the corners of the rectangles for the histogram
left = np.array(bins[:-1])
right = np.array(bins[1:])
bottom = np.zeros(len(left))
top = bottom + n
nrects = len(left)

nverts = nrects*(1+3+1)
verts = np.zeros((nverts, 2))
codes = np.ones(nverts, int) * path.Path.LINETO
codes[0::5] = path.Path.MOVETO
codes[4::5] = path.Path.CLOSEPOLY
verts[0::5, 0] = left
verts[0::5, 1] = bottom
verts[1::5, 0] = left
verts[1::5, 1] = top
verts[2::5, 0] = right
verts[2::5, 1] = top
verts[3::5, 0] = right
verts[3::5, 1] = bottom

barpath = path.Path(verts, codes)
patch = patches.PathPatch(barpath, facecolor='green',
                          edgecolor='yellow', alpha=0.5)
ax.add_patch(patch)

ax.set_xlim(left[0], right[-1])
ax.set_ylim(bottom.min(), top.max())

plt.show()
```



2.3.3 Path effects guide

Defining paths that objects follow on a canvas.

Matplotlib's *patheffects* module provides functionality to apply a multiple draw stage to any Artist which can be rendered via a *path.Path*.

Artists which can have a path effect applied to them include *patches.Patch*, *lines.Line2D*, *collections.Collection* and even *text.Text*. Each artist's path effects can be controlled via the *Artist.set_path_effects* method, which takes an iterable of *AbstractPathEffect* instances.

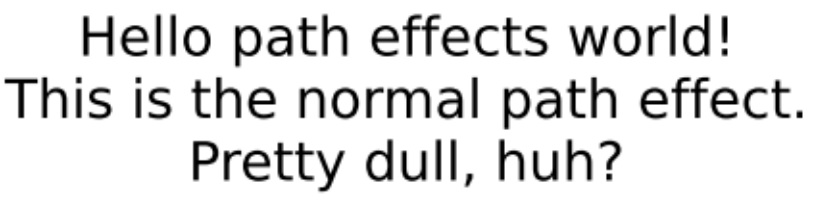
The simplest path effect is the *Normal* effect, which simply draws the artist without any effect:

```
import matplotlib.pyplot as plt
import matplotlib.patheffects as path_effects

fig = plt.figure(figsize=(5, 1.5))
text = fig.text(0.5, 0.5, 'Hello path effects world!\nThis is the normal '
                          'path effect.\nPretty dull, huh?',
                ha='center', va='center', size=20)
```

(continues on next page)

(continued from previous page)

```
text.set_path_effects([path_effects.Normal()])
plt.show()
```

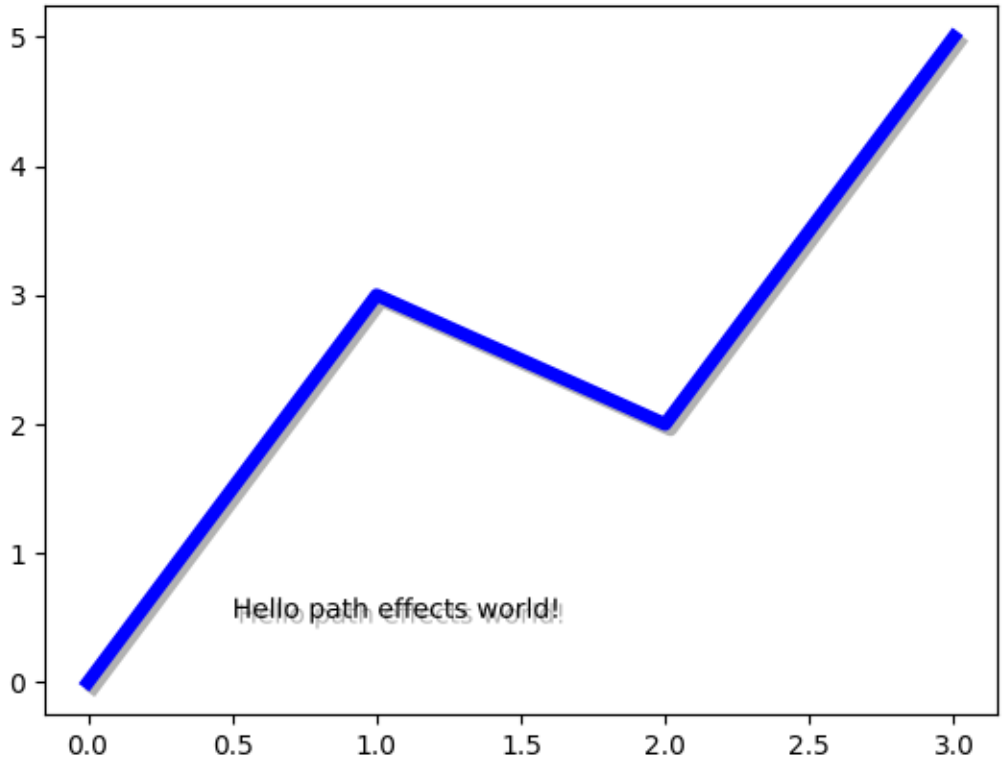


Whilst the plot doesn't look any different to what you would expect without any path effects, the drawing of the text has now been changed to use the path effects framework, opening up the possibilities for more interesting examples.

Adding a shadow

A far more interesting path effect than *Normal* is the drop-shadow, which we can apply to any of our path based artists. The classes *SimplePatchShadow* and *SimpleLineShadow* do precisely this by drawing either a filled patch or a line patch below the original artist:

```
import matplotlib.patheffects as path_effects

text = plt.text(0.5, 0.5, 'Hello path effects world!',
                path_effects=[path_effects.withSimplePatchShadow()])

plt.plot([0, 3, 2, 5], linewidth=5, color='blue',
         path_effects=[path_effects.SimpleLineShadow(),
                       path_effects.Normal()])
plt.show()
```



Notice the two approaches to setting the path effects in this example. The first uses the `with*` classes to include the desired functionality automatically followed with the "normal" effect, whereas the latter explicitly defines the two path effects to draw.

Making an artist stand out

One nice way of making artists visually stand out is to draw an outline in a bold color below the actual artist. The *Stroke* path effect makes this a relatively simple task:

```
fig = plt.figure(figsize=(7, 1))
text = fig.text(0.5, 0.5, 'This text stands out because of\n'
                          'its black border.', color='white',
                          ha='center', va='center', size=30)
text.set_path_effects([path_effects.Stroke(linewidth=3, foreground='black'),
                       path_effects.Normal()])
plt.show()
```

It is important to note that this effect only works because we have drawn the text path twice; once with a thick black line, and then once with the original text path on top.

You may have noticed that the keywords to *Stroke* and *SimplePatchShadow* and *SimpleLineShadow* are not the usual Artist keywords (*facecolor* *edgecolor*, etc.). This is because with these path effects we are operating at lower level of Matplotlib. In fact, the keywords which are accepted are those for a *matplotlib.backend_bases.GraphicsContextBase* instance, which have been designed for making it easy to create new backends - and not for its user interface.

Greater control of the path effect artist

As already mentioned, some of the path effects operate at a lower level than most users will be used to, meaning that setting keywords such as *facecolor* and *edgecolor* raise an AttributeError. Luckily there is a generic *PathPatchEffect* path effect which creates a *patches.PathPatch* class with the original path. The keywords to this effect are identical to those of *patches.PathPatch*:

```
fig = plt.figure(figsize=(8, 1))
t = fig.text(0.02, 0.5, 'Hatch shadow', fontsize=75, weight=1000, va='center')
t.set_path_effects([
    path_effects.PathPatchEffect(
        offset=(4, -4), hatch='xxxx', facecolor='gray'),
    path_effects.PathPatchEffect(
        edgecolor='white', linewidth=1.1, facecolor='black')])
plt.show()
```

2.3.4 Transformations Tutorial

Like any graphics packages, Matplotlib is built on top of a transformation framework to easily move between coordinate systems, the userland *data* coordinate system, the *axes* coordinate system, the *figure* coordinate system, and the *display* coordinate system. In 95% of your plotting, you won't need to think about this, as it happens under the hood, but as you push the limits of custom figure generation, it helps to have an understanding of these objects so you can reuse the existing transformations Matplotlib makes available to you, or create your own (see *matplotlib.transforms*). The table below summarizes the some useful coordinate systems, the transformation object you should use to work in that coordinate system, and the description of that system. In the `Transformation Object` column, `ax` is a *Axes* instance, and `fig` is a *Figure* instance.

Co-ordi-nates	Transformation object	Description
"data"	`ax.transData`	The coordinate system for the data, controlled by xlim and ylim.
"axes"	`ax.transAxes`	The coordinate system of the *Axes*; (0, 0) is bottom left of the axes, and (1, 1) is top right of the axes.
"fig-ure"	`fig.transFigure`	The coordinate system of the *Figure*; (0, 0) is bottom left of the figure, and (1, 1) is top right of the figure.
"figure-inches"	`fig.dpi_scale_trans`	The coordinate system of the *Figure* in inches; (0, 0) is bottom left of the figure, and (width, height) is the top right of the figure in inches.
"dis-play"	`None`, or `IdentityTransform()`	The pixel coordinate system of the display window; (0, 0) is bottom left of the window, and (width, height) is top right of the display window in pixels.
"xaxis", "yaxis"	`ax.get_xaxis_transform()`, `ax.get_yaxis_transform()`	Blended coordinate systems; use data coordinates on one of the axis and axes coordinates on the other.

All of the transformation objects in the table above take inputs in their coordinate system, and transform the input to the *display* coordinate system. That is why the *display* coordinate system has `None` for the `Transformation Object` column -- it already is in *display* coordinates. The transformations also know how to invert themselves, to go from *display* back to the native coordinate system. This is particularly useful when processing events from the user interface, which typically occur in display space, and you want to know where the mouse click or key-press occurred in your *data* coordinate system.

Note that specifying objects in *display* coordinates will change their location if the `dpi` of the figure changes. This can cause confusion when printing or changing screen resolution, because the object can change location and size. Therefore it is most common for artists placed in an axes or figure to have their transform set to something *other* than the *IdentityTransform()*; the default when an artist is placed on an axes using *add_artist* is for the transform to be `ax.transData`.

Data coordinates

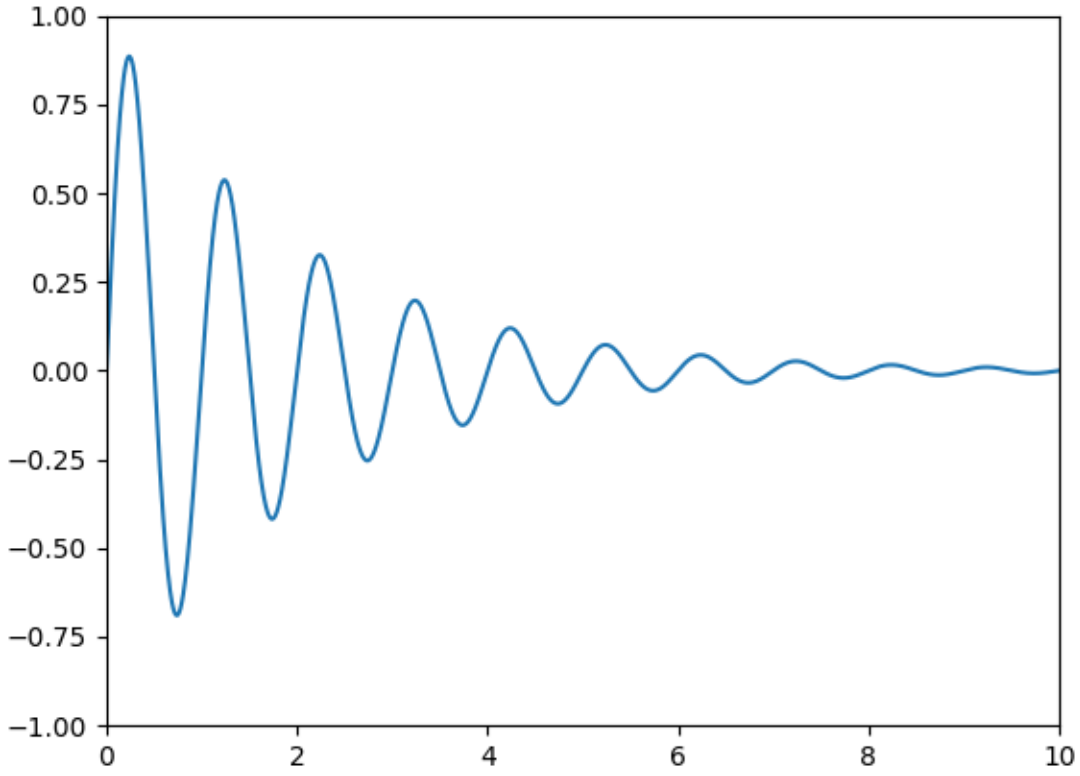
Let's start with the most commonly used coordinate, the *data* coordinate system. Whenever you add data to the axes, Matplotlib updates the datalimits, most commonly updated with the *set_xlim()* and *set_ylim()* methods. For example, in the figure below, the data limits stretch from 0 to 10 on the x-axis, and -1 to 1 on the y-axis.

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
```

(continues on next page)

(continued from previous page)

```
x = np.arange(0, 10, 0.005)
y = np.exp(-x/2.) * np.sin(2*np.pi*x)

fig, ax = plt.subplots()
ax.plot(x, y)
ax.set_xlim(0, 10)
ax.set_ylim(-1, 1)

plt.show()
```



You can use the `ax.transData` instance to transform from your *data* to your *display* coordinate system, either a single point or a sequence of points as shown below:

```
In [14]: type(ax.transData)
Out[14]: <class 'matplotlib.transforms.CompositeGenericTransform'>

In [15]: ax.transData.transform((5, 0))
Out[15]: array([ 335.175,  247.   ])

In [16]: ax.transData.transform([(5, 0), (1, 2)])
Out[16]:
```

(continues on next page)

(continued from previous page)

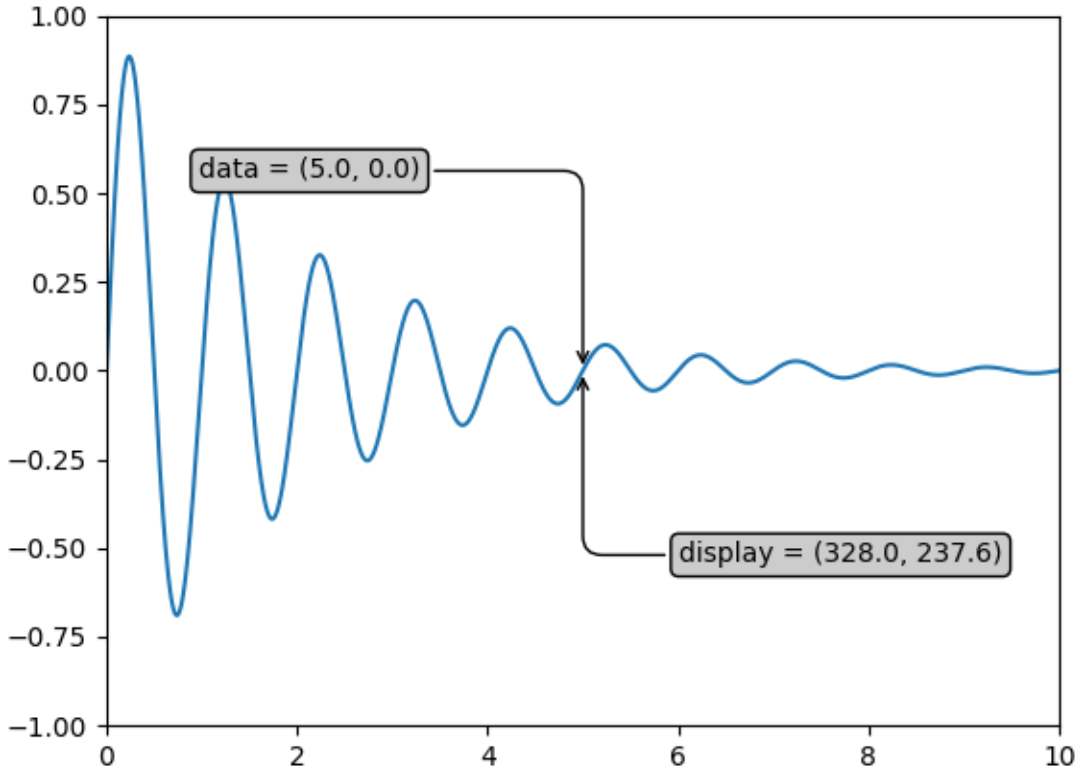
```
array([[ 335.175,  247.   ],
       [ 132.435,  642.2  ]])
```

You can use the *inverted()* method to create a transform which will take you from *display* to *data* coordinates:

```
In [41]: inv = ax.transData.inverted()

In [42]: type(inv)
Out[42]: <class 'matplotlib.transforms.CompositeGenericTransform'>

In [43]: inv.transform((335.175,  247.))
Out[43]: array([ 5.,  0.])
```

If your are typing along with this tutorial, the exact values of the *display* coordinates may differ if you have a different window size or dpi setting. Likewise, in the figure below, the display labeled points are probably not the same as in the ipython session because the documentation figure size defaults are different.

```
x = np.arange(0, 10, 0.005)
y = np.exp(-x/2.) * np.sin(2*np.pi*x)

fig, ax = plt.subplots()
ax.plot(x, y)
ax.set_xlim(0, 10)
ax.set_ylim(-1, 1)

xdata, ydata = 5, 0
xdisplay, ydisplay = ax.transData.transform((xdata, ydata))

bbox = dict(boxstyle="round", fc="0.8")
arrowprops = dict(
    arrowstyle="->",
    connectionstyle="angle,angleA=0,angleB=90,rad=10")

offset = 72
ax.annotate('data = (%.1f, %.1f)' % (xdata, ydata),
            (xdata, ydata), xytext=(-2*offset, offset), textcoords='offset points',
            bbox=bbox, arrowprops=arrowprops)

disp = ax.annotate('display = (%.1f, %.1f)' % (xdisplay, ydisplay),
                   (xdisplay, ydisplay), xytext=(0.5*offset, -offset),
                   xycoords='figure pixels',
                   textcoords='offset points',
                   bbox=bbox, arrowprops=arrowprops)

plt.show()
```



Note: If you run the source code in the example above in a GUI backend, you may also find that the two arrows for the *data* and *display* annotations do not point to exactly the same point. This is because the display point was computed before the figure was displayed, and the GUI backend may slightly resize the figure when it is created. The effect is more pronounced if you resize the figure yourself. This is one good reason why you rarely want to work in *display* space, but you can connect to the `'on_draw'` *Event* to update *figure* coordinates on figure draws; see *Event handling and picking*.

When you change the x or y limits of your axes, the data limits are updated so the transformation yields a new display point. Note that when we just change the ylim, only the y-display coordinate is altered, and when we change the xlim too, both are altered. More on this later when we talk about the *Bbox*.

```
In [54]: ax.transData.transform((5, 0))
Out[54]: array([ 335.175,  247.   ])

In [55]: ax.set_ylim(-1, 2)
Out[55]: (-1, 2)
```

(continues on next page)

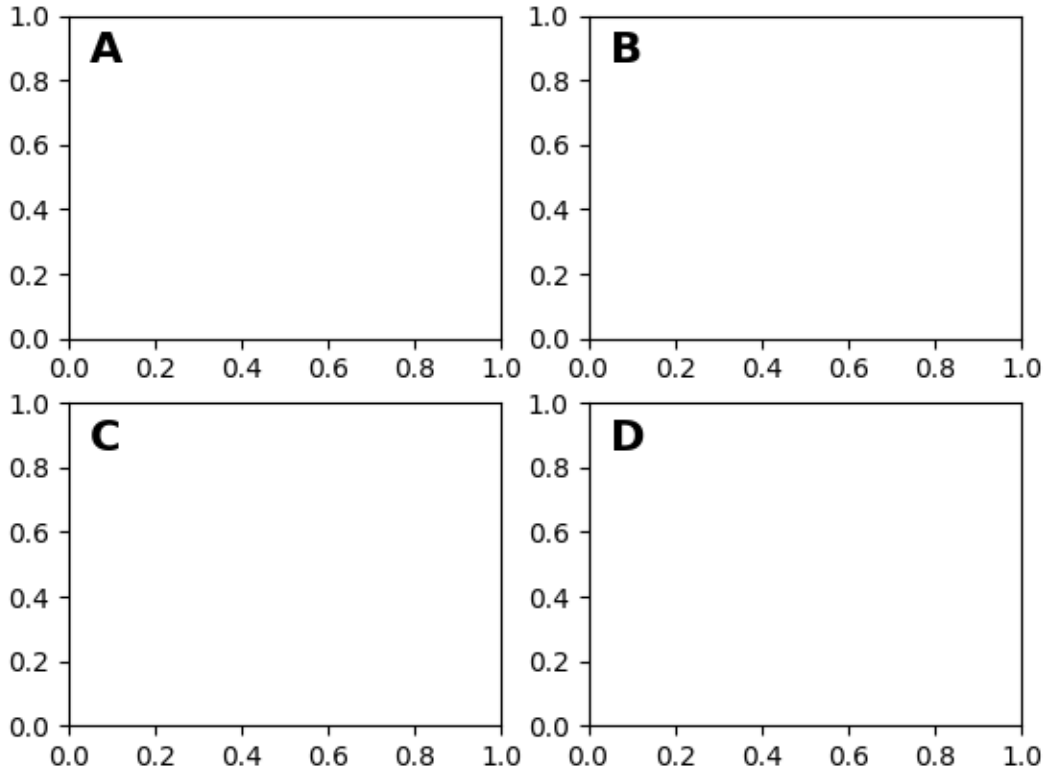
(continued from previous page)

```
In [56]: ax.transData.transform((5, 0))
Out[56]: array([ 335.175     ,  181.13333333])

In [57]: ax.set_xlim(10, 20)
Out[57]: (10, 20)

In [58]: ax.transData.transform((5, 0))
Out[58]: array([-171.675     ,  181.13333333])
```

Axes coordinates

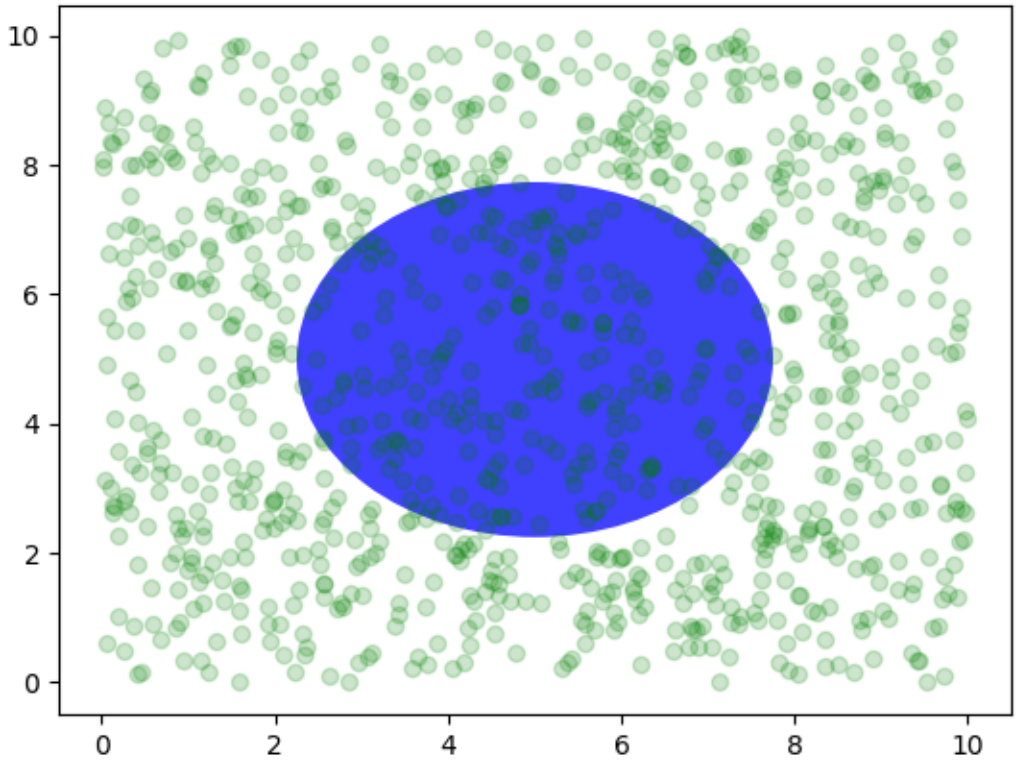
After the *data* coordinate system, *axes* is probably the second most useful coordinate system. Here the point (0, 0) is the bottom left of your axes or subplot, (0.5, 0.5) is the center, and (1.0, 1.0) is the top right. You can also refer to points outside the range, so (-0.1, 1.1) is to the left and above your axes. This coordinate system is extremely useful when placing text in your axes, because you often want a text bubble in a fixed, location, e.g., the upper left of the axes pane, and have that location remain fixed when you pan or zoom. Here is a simple example that creates four panels and labels them 'A', 'B', 'C', 'D' as you often see in journals.

```
fig = plt.figure()
for i, label in enumerate(('A', 'B', 'C', 'D')):
    ax = fig.add_subplot(2, 2, i+1)
    ax.text(0.05, 0.95, label, transform=ax.transAxes,
            fontsize=16, fontweight='bold', va='top')

plt.show()
```



You can also make lines or patches in the *axes* coordinate system, but this is less useful in my experience than using `ax.transAxes` for placing text. Nonetheless, here is a silly example which plots some random dots in data space, and overlays a semi-transparent *Circle* centered in the middle of the axes with a radius one quarter of the axes -- if your axes does not preserve aspect ratio (see *set_aspect()*), this will look like an ellipse. Use the pan/zoom tool to move around, or manually change the data xlim and ylim, and you will see the data move, but the circle will remain fixed because it is not in *data* coordinates and will always remain at the center of the axes.

```
fig, ax = plt.subplots()
x, y = 10*np.random.rand(2, 1000)
ax.plot(x, y, 'go', alpha=0.2)  # plot some data in data coordinates

circ = mpatches.Circle((0.5, 0.5), 0.25, transform=ax.transAxes,
                       facecolor='blue', alpha=0.75)
ax.add_patch(circ)
plt.show()
```



Blended transformations

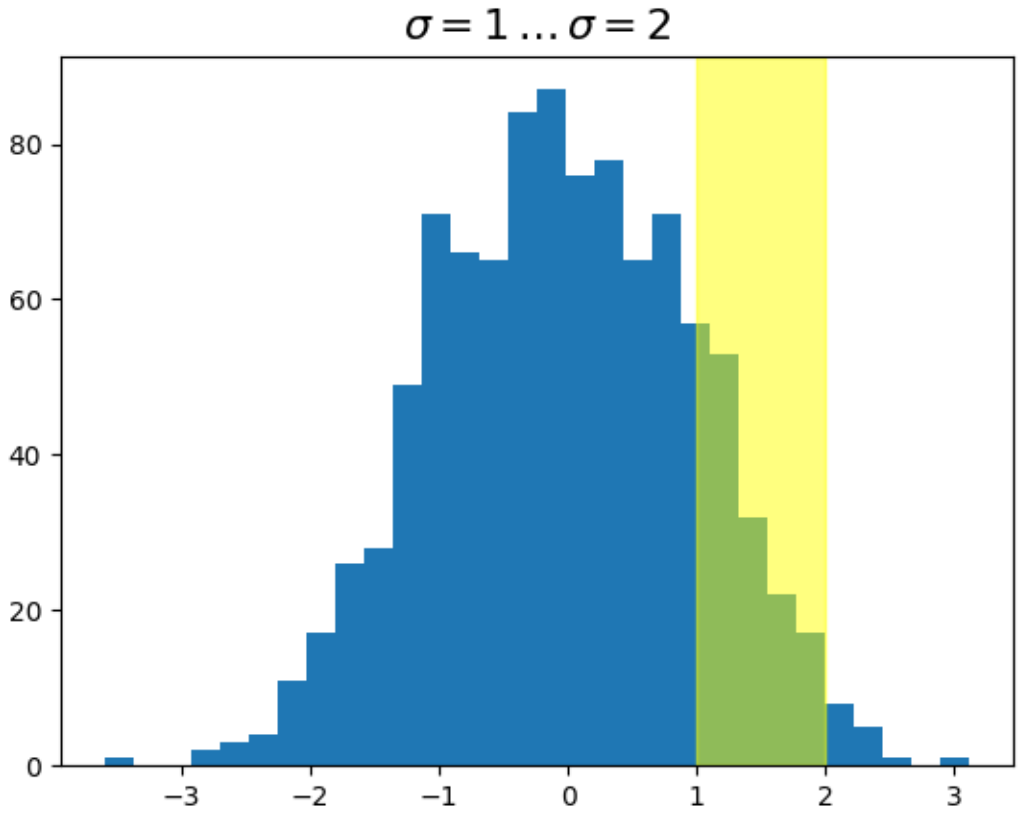
Drawing in *blended* coordinate spaces which mix *axes* with *data* coordinates is extremely useful, for example to create a horizontal span which highlights some region of the y-data but spans across the x-axis regardless of the data limits, pan or zoom level, etc. In fact these blended lines and spans are so useful, we have built in functions to make them easy to plot (see *axhline()*, *axvline()*, *axhspan()*, *axvspan()*) but for didactic purposes we will implement the horizontal span here using a blended transformation. This trick only works for separable transformations, like you see in normal Cartesian coordinate systems, but not on inseparable transformations like the *PolarTransform*.

```
import matplotlib.transforms as transforms

fig, ax = plt.subplots()
x = np.random.randn(1000)

ax.hist(x, 30)
ax.set_title(r'$\sigma=1 \/ \dots \/ \sigma=2$', fontsize=16)
```

(continues on next page)

(continued from previous page)

```
# the x coords of this transformation are data, and the y coord are axes
trans = transforms.blended_transform_factory(
    ax.transData, ax.transAxes)
# highlight the 1..2 stddev region with a span.
# We want x to be in data coordinates and y to span from 0..1 in axes coords.
rect = mpatches.Rectangle((1, 0), width=1, height=1, transform=trans,
                          color='yellow', alpha=0.5)
ax.add_patch(rect)

plt.show()
```



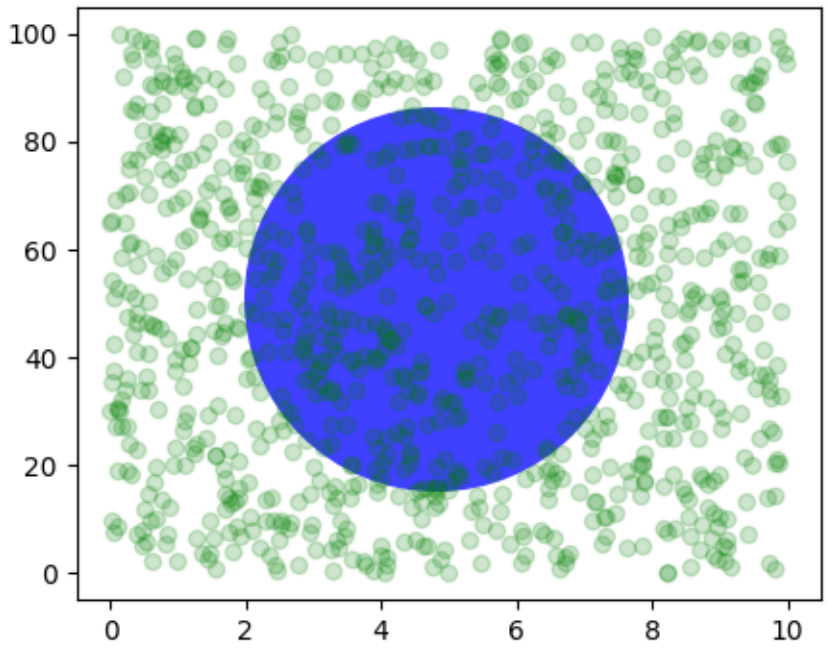
Note: The blended transformations where x is in *data* coords and y in *axes* coordinates is so useful that we have helper methods to return the versions Matplotlib uses internally for drawing ticks, ticklabels, etc. The methods are *matplotlib.axes.Axes.get_xaxis_transform()* and *matplotlib.axes.Axes.get_yaxis_transform()*. So in the example above, the call to *blended_transform_factory()* can be replaced by `get_xaxis_transform`:

```
trans = ax.get_xaxis_transform()
```

Plotting in physical coordinates

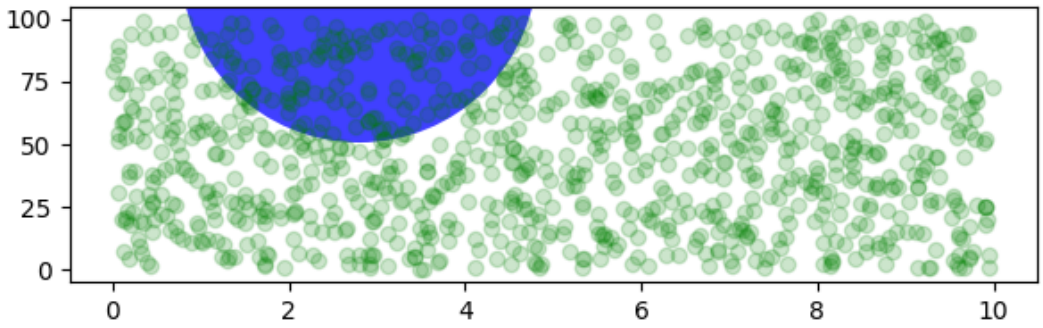
Sometimes we want an object to be a certain physical size on the plot. Here we draw the same circle as above, but in physical coordinates. If done interactively, you can see that changing the size of the figure does not change the offset of the circle from the lower-left corner, does not change its size, and the circle remains a circle regardless of the aspect ratio of the axes.

```
fig, ax = plt.subplots(figsize=(5, 4))
x, y = 10*np.random.rand(2, 1000)
ax.plot(x, y*10., 'go', alpha=0.2)  # plot some data in data coordinates
# add a circle in fixed-coordinates
circ = mpatches.Circle((2.5, 2), 1.0, transform=fig.dpi_scale_trans,
                       facecolor='blue', alpha=0.75)
ax.add_patch(circ)
plt.show()
```



If we change the figure size, the circle does not change its absolute position and is cropped.

```
fig, ax = plt.subplots(figsize=(7, 2))
x, y = 10*np.random.rand(2, 1000)
ax.plot(x, y*10., 'go', alpha=0.2)  # plot some data in data coordinates
# add a circle in fixed-coordinates
circ = mpatches.Circle((2.5, 2), 1.0, transform=fig.dpi_scale_trans,
                       facecolor='blue', alpha=0.75)
ax.add_patch(circ)
plt.show()
```



Another use is putting a patch with a set physical dimension around a data point on the axes. Here we add together two transforms. The first sets the scaling of how large the ellipse should be and the second sets its position. The ellipse is then placed at the origin, and then we use the helper transform *ScaledTranslation* to move it to the right place in the `ax.transData` coordinate system. This helper is instantiated with:

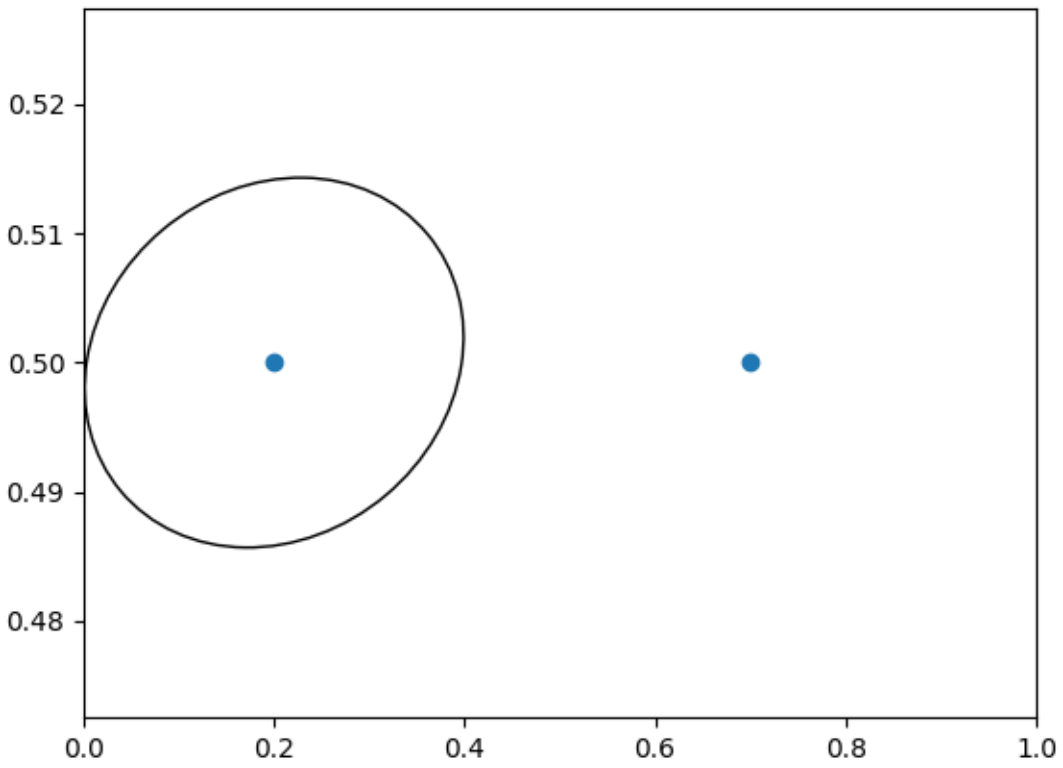
```
trans = ScaledTranslation(xt, yt, scale_trans)
```

where *xt* and *yt* are the translation offsets, and *scale_trans* is a transformation which scales *xt* and *yt* at transformation time before applying the offsets.

Note the use of the plus operator on the transforms below. This code says: first apply the scale transformation `fig.dpi_scale_trans` to make the ellipse the proper size, but still centered at (0, 0), and then translate the data to `xdata[0]` and `ydata[0]` in data space.

In interactive use, the ellipse stays the same size even if the axes limits are changed via zoom.

```
fig, ax = plt.subplots()
xdata, ydata = (0.2, 0.7), (0.5, 0.5)
ax.plot(xdata, ydata, "o")
ax.set_xlim((0, 1))

trans = (fig.dpi_scale_trans +
         transforms.ScaledTranslation(xdata[0], ydata[0], ax.transData))

# plot an ellipse around the point that is 150 x 130 points in diameter...
circle = mpatches.Ellipse((0, 0), 150/72, 130/72, angle=40,
                          fill=None, transform=trans)
ax.add_patch(circle)
plt.show()
```



Note: The order of transformation matters. Here the ellipse is given the right dimensions in display space *first* and then moved in data space to the correct spot. If we had done the `ScaledTranslation` first, then `xdata[0]` and `ydata[0]` would first be transformed to *display* coordinates (`[ 358.4 475.2]` on a 200-dpi monitor) and then those coordinates would be scaled by `fig.dpi_scale_trans` pushing the center of the ellipse well off the screen (i.e. `[ 71680. 95040.]`).

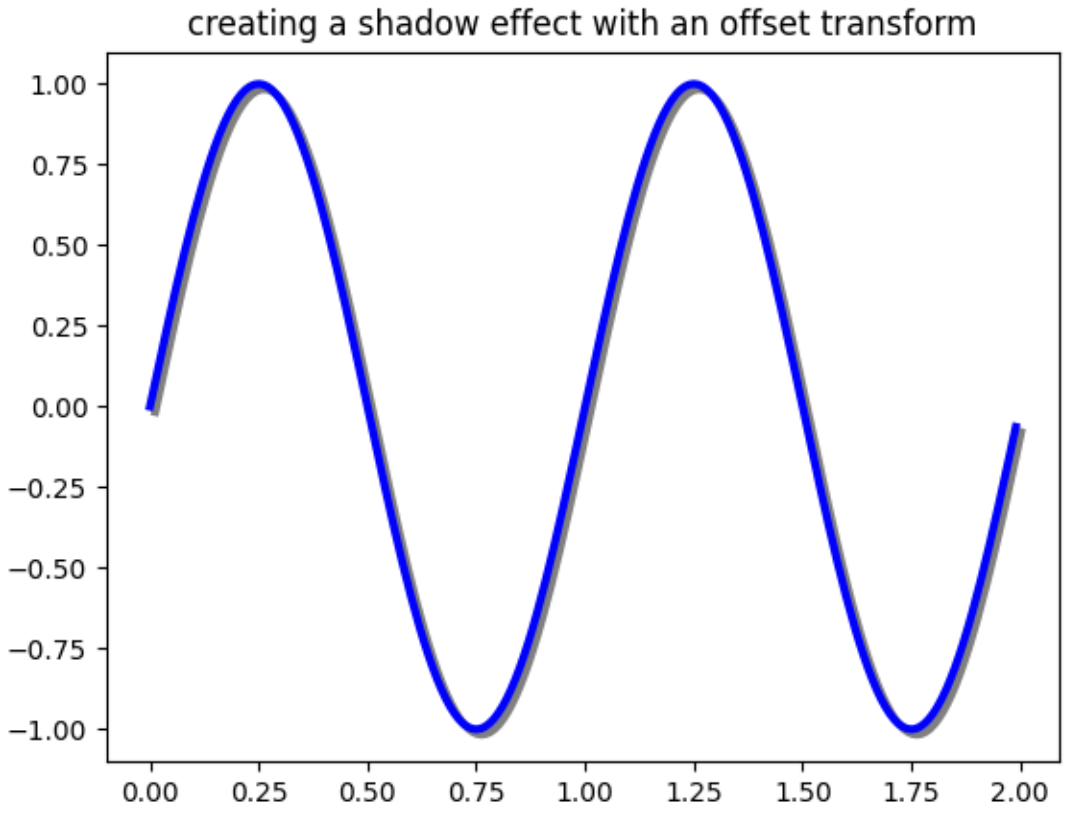
Using offset transforms to create a shadow effect

Another use of *`ScaledTranslation`* is to create a new transformation that is offset from another transformation, e.g., to place one object shifted a bit relative to another object. Typically you want the shift to be in some physical dimension, like points or inches rather than in *data* coordinates, so that the shift effect is constant at different zoom levels and dpi settings.

One use for an offset is to create a shadow effect, where you draw one object identical to the first just to the right of it, and just below it, adjusting the zorder to make sure the shadow is drawn first and then the object it is shadowing above it.

Here we apply the transforms in the *opposite* order to the use of *ScaledTranslation* above. The plot is first made in data coordinates (`ax.transData`) and then shifted by `dx` and `dy` points using `fig.dpi_scale_trans`. (In typography, a point is 1/72 inches, and by specifying your offsets in points, your figure will look the same regardless of the dpi resolution it is saved in.)

```
fig, ax = plt.subplots()

# make a simple sine wave
x = np.arange(0., 2., 0.01)
y = np.sin(2*np.pi*x)
line, = ax.plot(x, y, lw=3, color='blue')

# shift the object over 2 points, and down 2 points
dx, dy = 2/72., -2/72.
offset = transforms.ScaledTranslation(dx, dy, fig.dpi_scale_trans)
shadow_transform = ax.transData + offset

# now plot the same data with our offset transform;
# use the zorder to make sure we are below the line
ax.plot(x, y, lw=3, color='gray',
        transform=shadow_transform,
        zorder=0.5*line.get_zorder())

ax.set_title('creating a shadow effect with an offset transform')
plt.show()
```



Note: The dpi and inches offset is a common-enough use case that we have a special helper function to create it in *matplotlib.transforms.offset_copy()*, which returns a new transform with an added offset. So above we could have done:

```
shadow_transform = transforms.offset_copy(ax.transData,
        fig=fig, dx, dy, units='inches')
```

The transformation pipeline

The `ax.transData` transform we have been working with in this tutorial is a composite of three different transformations that comprise the transformation pipeline from *data -> display* coordinates. Michael Droettboom implemented the transformations framework, taking care to provide a clean API that segregated the nonlinear projections and scales that happen in polar and logarithmic plots, from the linear affine transformations that happen when you pan and zoom. There is an efficiency here, because you can pan and zoom in your axes which affects the affine transformation, but you may not need to compute the potentially expensive nonlinear scales or projections on simple navigation events. It is also possible to multiply affine transformation

matrices together, and then apply them to coordinates in one step. This is not true of all possible transformations.

Here is how the `ax.transData` instance is defined in the basic separable axis *Axes* class:

```
self.transData = self.transScale + (self.transLimits + self.transAxes)
```

We've been introduced to the `transAxes` instance above in *Axes coordinates*, which maps the (0, 0), (1, 1) corners of the axes or subplot bounding box to *display* space, so let's look at these other two pieces.

`self.transLimits` is the transformation that takes you from *data* to *axes* coordinates; i.e., it maps your view xlim and ylim to the unit space of the axes (and `transAxes` then takes that unit space to display space). We can see this in action here

```
In [80]: ax = subplot(111)

In [81]: ax.set_xlim(0, 10)
Out[81]: (0, 10)

In [82]: ax.set_ylim(-1, 1)
Out[82]: (-1, 1)

In [84]: ax.transLimits.transform((0, -1))
Out[84]: array([ 0.,  0.])

In [85]: ax.transLimits.transform((10, -1))
Out[85]: array([ 1.,  0.])

In [86]: ax.transLimits.transform((10, 1))
Out[86]: array([ 1.,  1.])

In [87]: ax.transLimits.transform((5, 0))
Out[87]: array([ 0.5,  0.5])
```

and we can use this same inverted transformation to go from the unit *axes* coordinates back to *data* coordinates.

```
In [90]: inv.transform((0.25, 0.25))
Out[90]: array([ 2.5, -0.5])
```

The final piece is the `self.transScale` attribute, which is responsible for the optional non-linear scaling of the data, e.g., for logarithmic axes. When an Axes is initially setup, this is just set to the identity transform, since the basic Matplotlib axes has linear scale, but when you call a logarithmic scaling function like *semilogx()* or explicitly set the scale to logarithmic with *set_xscale()*, then the `ax.transScale` attribute is set to handle the nonlinear projection. The scales transforms are properties of the respective `xaxis` and `yaxis` *Axis* instances. For example, when you call `ax.set_xscale('log')`, the xaxis updates its scale to a *matplotlib.scale.LogScale* instance.

For non-separable axes the PolarAxes, there is one more piece to consider, the projection transformation. The `transData` *matplotlib.projections.polar.PolarAxes* is

similar to that for the typical separable matplotlib Axes, with one additional piece `transProjection`:

```
self.transData = self.transScale + self.transProjection + \
    (self.transProjectionAffine + self.transAxes)
```

`transProjection` handles the projection from the space, e.g., latitude and longitude for map data, or radius and theta for polar data, to a separable Cartesian coordinate system. There are several projection examples in the *matplotlib.projections* package, and the best way to learn more is to open the source for those packages and see how to make your own, since Matplotlib supports extensible axes and projections. Michael Droettboom has provided a nice tutorial example of creating a Hammer projection axes; see /gallery/misc/custom_projection.

Total running time of the script: (0 minutes 2.835 seconds)

2.4 Colors

Matplotlib has support for visualizing information with a wide array of colors and colormaps. These tutorials cover the basics of how these colormaps look, how you can create your own, and how you can customize colormaps for your use case.

For even more information see the examples page.

2.4.1 Specifying Colors

Matplotlib recognizes the following formats to specify a color:

- an RGB or RGBA (red, green, blue, alpha) tuple of float values in closed interval `[0, 1]` (e.g., `(0.1, 0.2, 0.5)` or `(0.1, 0.2, 0.5, 0.3)`);
- a hex RGB or RGBA string (e.g., `'#0f0f0f'` or `'#0f0f0f80'`; case-insensitive);
- a shorthand hex RGB or RGBA string, equivalent to the hex RGB or RGBA string obtained by duplicating each character, (e.g., `'#abc'`, equivalent to `'#aabbcc'`, or `'#abcd'`, equivalent to `'#aabbccdd'`; case-insensitive);
- a string representation of a float value in `[0, 1]` inclusive for gray level (e.g., `'0.5'`);
- one of the characters `{'b', 'g', 'r', 'c', 'm', 'y', 'k', 'w'}`, which are short-hand notations for shades of blue, green, red, cyan, magenta, yellow, black, and white. Note that the colors `'g'`, `'c'`, `'m'`, `'y'` do not coincide with the X11/CSS4 colors. Their particular shades were chosen for better visibility of colored lines against typical backgrounds.
- a X11/CSS4 color name (case-insensitive);
- a name from the xkcd color survey, prefixed with `'xkcd:'` (e.g., `'xkcd:sky blue'`; case insensitive);

- one of the Tableau Colors from the 'T10' categorical palette (the default color cycle): `{'tab:blue', 'tab:orange', 'tab:green', 'tab:red', 'tab:purple', 'tab:brown', 'tab:pink', 'tab:gray', 'tab:olive', 'tab:cyan'}` (case-insensitive);
- a "CN" color spec, i.e. `'C'` followed by a number, which is an index into the default property cycle (`rcParams["axes.prop_cycle"]` (default: `cycler('color', ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728', '#9467bd', '#8c564b', '#e377c2', '#7f7f7f', '#bcbd22', '#17becf'])`); the indexing is intended to occur at rendering time, and defaults to black if the cycle does not include color.

"Red", "Green", and "Blue" are the intensities of those colors, the combination of which span the colorspace.

How "Alpha" behaves depends on the `zorder` of the Artist. Higher `zorder` Artists are drawn on top of lower Artists, and "Alpha" determines whether the lower artist is covered by the higher. If the old RGB of a pixel is `RGBold` and the RGB of the pixel of the Artist being added is `RGBnew` with Alpha `alpha`, then the RGB of the pixel is updated to: `RGB = RGBOld * (1 - Alpha) + RGBnew * Alpha`. Alpha of 1 means the old color is completely covered by the new Artist, Alpha of 0 means that pixel of the Artist is transparent.

For more information on colors in matplotlib see

- the /gallery/color/color_demo example;
- the *matplotlib.colors* API;
- the /gallery/color/named_colors example.

"CN" color selection

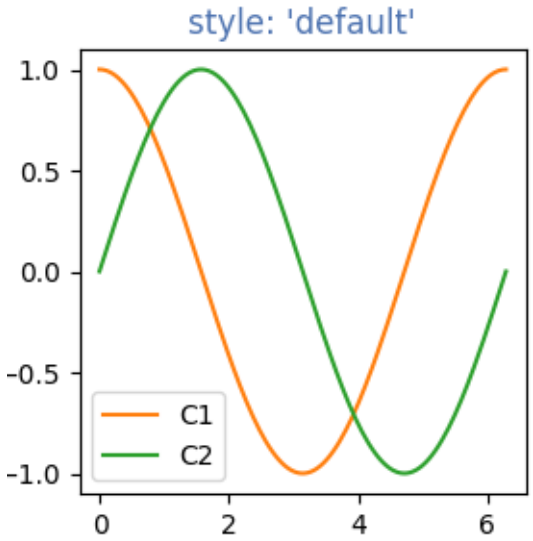
"CN" colors are converted to RGBA as soon as the artist is created. For example,

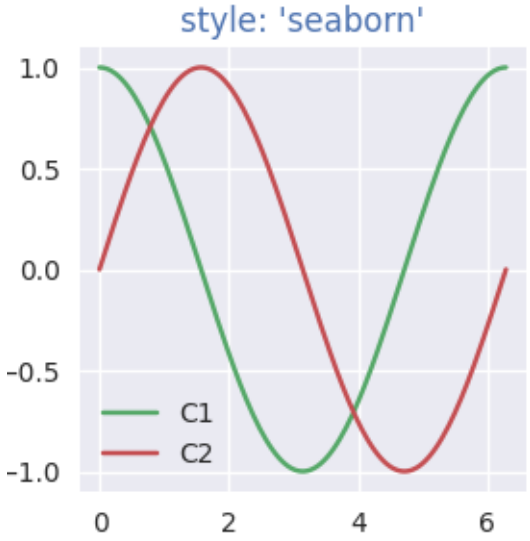
```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl

th = np.linspace(0, 2*np.pi, 128)


def demo(sty):
    mpl.style.use(sty)
    fig, ax = plt.subplots(figsize=(3, 3))

    ax.set_title('style: {!r}'.format(sty), color='C0')

    ax.plot(th, np.cos(th), 'C1', label='C1')
    ax.plot(th, np.sin(th), 'C2', label='C2')
    ax.legend()
```

(continues on next page)

(continued from previous page)

```
demo('default')
demo('seaborn')
```





will use the first color for the title and then plot using the second and third colors of each style's `rcParams["axes.prop_cycle"]` (default: `cycler('color', ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728', '#9467bd', '#8c564b', '#e377c2', '#7f7f7f', '#bcbd22', '#17becf'])`).

xkcd v X11/CSS4

The xkcd colors are derived from a user survey conducted by the webcomic xkcd. Details of the survey are available on the xkcd blog.

Out of 148 colors in the CSS color list, there are 95 name collisions between the X11/CSS4 names and the xkcd names, all but 3 of which have different hex values. For example `'blue'` maps to `'#0000FF'` where as `'xkcd:blue'` maps to `'#0343DF'`. Due to these name collisions all of the xkcd colors have `'xkcd:'` prefixed. As noted in the blog post, while it might be interesting to re-define the X11/CSS4 names based on such a survey, we do not do so unilaterally.

The name collisions are shown in the table below; the color names where the hex values agree are shown in bold.

```
import matplotlib._color_data as mcd
import matplotlib.patches as mpatch

overlap = {name for name in mcd.CSS4_COLORS
           if "xkcd:" + name in mcd.XKCD_COLORS}

fig = plt.figure(figsize=[4.8, 16])
ax = fig.add_axes([0, 0, 1, 1])

for j, n in enumerate(sorted(overlap, reverse=True)):
    weight = None
    cn = mcd.CSS4_COLORS[n]
    xkcd = mcd.XKCD_COLORS["xkcd:" + n].upper()
    if cn == xkcd:
        weight = 'bold'

    r1 = mpatch.Rectangle((0, j), 1, 1, color=cn)
    r2 = mpatch.Rectangle((1, j), 1, 1, color=xkcd)
    txt = ax.text(2, j+.5, '  ' + n, va='center', fontsize=10,
                  weight=weight)
    ax.add_patch(r1)
    ax.add_patch(r2)
    ax.axhline(j, color='k')

ax.text(.5, j + 1.5, 'X11', ha='center', va='center')
ax.text(1.5, j + 1.5, 'xkcd', ha='center', va='center')
ax.set_xlim(0, 3)
ax.set_ylim(0, j + 2)
ax.axis('off')
```


X11
xkcd
aqua
aquamarine
azure
beige
black
blue
brown
chartreuse
chocolate
coral
crimson
cyan
darkblue
darkgreen
fuchsia
gold
goldenrod
green
grey
indigo
ivory
khaki
lavender
lightblue
lightgreen
lime
magenta
maroon
navy
olive
orange
orangered
orchid
pink
plum
purple
red
salmon
sienna
silver
tan
teal
tomato
turquoise
violet
wheat
white
yellow
yellowgreen

Out:

```
(0.0, 3.0, 0.0, 50.0)
```

Total running time of the script: (0 minutes 1.068 seconds)

2.4.2 Customized Colorbars Tutorial

This tutorial shows how to build and customize standalone colorbars, i.e. without an attached plot.

Customized Colorbars

A *colorbar* needs a "mappable" (*matplotlib.cm.ScalarMappable*) object (typically, an image) which indicates the colormap and the norm to be used. In order to create a colorbar without an attached image, one can instead use a *ScalarMappable* with no associated data.

Basic continuous colorbar

Here we create a basic continuous colorbar with ticks and labels.

The arguments to the *colorbar* call are the *ScalarMappable* (constructed using the *norm* and *cmap* arguments), the axes where the colorbar should be drawn, and the colorbar's orientation.

For more information see the *colorbar* API.

```
import matplotlib.pyplot as plt
import matplotlib as mpl

fig, ax = plt.subplots(figsize=(6, 1))
fig.subplots_adjust(bottom=0.5)

cmap = mpl.cm.cool
norm = mpl.colors.Normalize(vmin=5, vmax=10)

fig.colorbar(mpl.cm.ScalarMappable(norm=norm, cmap=cmap),
             cax=ax, orientation='horizontal', label='Some Units')
```

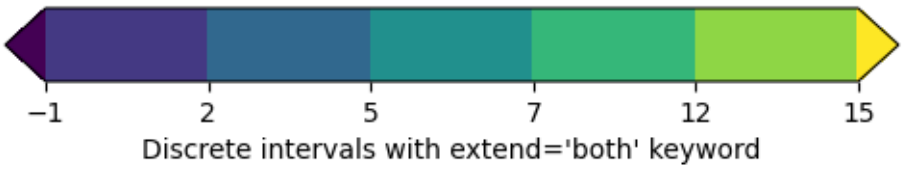
Out:

```
<matplotlib.colorbar.Colorbar object at 0x7f53ff7ba220>
```

Extended colorbar with continuous colorscale

The second example shows how to make a discrete colorbar based on a continuous cmap. With the "extend" keyword argument the appropriate colors are chosen to fill the colorspace, including the extensions:

```
fig, ax = plt.subplots(figsize=(6, 1))
fig.subplots_adjust(bottom=0.5)

cmap = mpl.cm.viridis
bounds = [-1, 2, 5, 7, 12, 15]
norm = mpl.colors.BoundaryNorm(bounds, cmap.N, extend='both')
cb2 = mpl.colorbar.ColorbarBase(ax, cmap=cmap,
                                norm=norm,
                                orientation='horizontal')
cb2.set_label("Discrete intervals with extend='both' keyword")
fig.show()
```



Discrete intervals colorbar

The third example illustrates the use of a *ListedColormap* which generates a colormap from a set of listed colors, *colors.BoundaryNorm* which generates a colormap index based on discrete intervals and extended ends to show the "over" and "under" value colors. Over and under are used to display data outside of the normalized [0, 1] range. Here we pass colors as gray shades as a string encoding a float in the 0-1 range.

If a *ListedColormap* is used, the length of the bounds array must be one greater than the length of the color list. The bounds must be monotonically increasing.

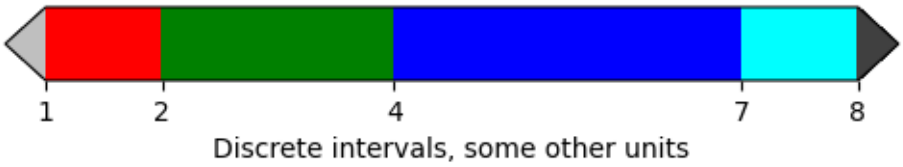
This time we pass additional arguments to *colorbar*. For the out-of-range values to display on the colorbar without using the *extend* keyword with *colors.BoundaryNorm*, we have to use the *extend* keyword argument directly in the colorbar call, and supply an additional boundary on each end of the range. Here we also use the spacing argument to make the length of each colorbar segment proportional to its corresponding interval.

```
fig, ax = plt.subplots(figsize=(6, 1))
fig.subplots_adjust(bottom=0.5)
```

(continues on next page)

(continued from previous page)

```
cmap = mpl.colors.ListedColormap(['red', 'green', 'blue', 'cyan'])
cmap.set_over('0.25')
cmap.set_under('0.75')

bounds = [1, 2, 4, 7, 8]
norm = mpl.colors.BoundaryNorm(bounds, cmap.N)
fig.colorbar(
    mpl.cm.ScalarMappable(cmap=cmap, norm=norm),
    cax=ax,
    boundaries=[0] + bounds + [13],  # Adding values for extensions.
    extend='both',
    ticks=bounds,
    spacing='proportional',
    orientation='horizontal',
    label='Discrete intervals, some other units',
)
```



Out:

```
<matplotlib.colorbar.Colorbar object at 0x7f54005f6e20>
```

Colorbar with custom extension lengths

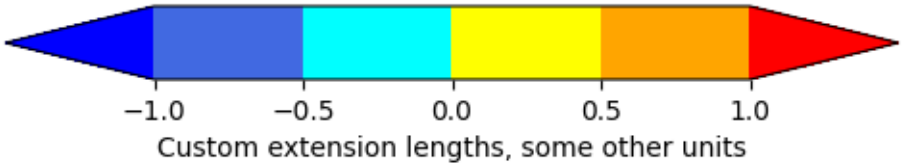
Here we illustrate the use of custom length colorbar extensions, on a colorbar with discrete intervals. To make the length of each extension the same as the length of the interior colors, use `extendfrac='auto'`.

```
fig, ax = plt.subplots(figsize=(6, 1))
fig.subplots_adjust(bottom=0.5)

cmap = mpl.colors.ListedColormap(['royalblue', 'cyan',
                                  'yellow', 'orange'])
cmap.set_over('red')
cmap.set_under('blue')

bounds = [-1.0, -0.5, 0.0, 0.5, 1.0]
norm = mpl.colors.BoundaryNorm(bounds, cmap.N)
fig.colorbar(
    mpl.cm.ScalarMappable(cmap=cmap, norm=norm),
    cax=ax,
    boundaries=[-10] + bounds + [10],
```

(continues on next page)

(continued from previous page)

```
    extend='both',
    extendfrac='auto',
    ticks=bounds,
    spacing='uniform',
    orientation='horizontal',
    label='Custom extension lengths, some other units',
)

plt.show()
```



2.4.3 Creating Colormaps in Matplotlib

Matplotlib has a number of built-in colormaps accessible via *matplotlib.cm.get_cmap*. There are also external libraries like palettable that have many extra colormaps.

However, we often want to create or manipulate colormaps in Matplotlib. This can be done using the class *ListedColormap* or *LinearSegmentedColormap*. Seen from the outside, both colormap classes map values between 0 and 1 to a bunch of colors. There are, however, slight differences, some of which are shown in the following.

Before manually creating or manipulating colormaps, let us first see how we can obtain colormaps and their colors from existing colormap classes.

Getting colormaps and accessing their values

First, getting a named colormap, most of which are listed in *Choosing Colormaps in Matplotlib*, may be done using *matplotlib.cm.get_cmap*, which returns a colormap object. The second argument gives the size of the list of colors used to define the colormap, and below we use a modest value of 8 so there are not a lot of values to look at.

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm
from matplotlib.colors import ListedColormap, LinearSegmentedColormap

viridis = cm.get_cmap('viridis', 8)
```

The object `viridis` is a callable, that when passed a float between 0 and 1 returns an RGBA value from the colormap:

```
print(viridis(0.56))
```

Out:

```
(0.122312, 0.633153, 0.530398, 1.0)
```

ListedColormap

ListedColormap s store their color values in a `.colors` attribute. The list of colors that comprise the colormap can be directly accessed using the `colors` property, or it can be accessed indirectly by calling `viridis` with an array of values matching the length of the colormap. Note that the returned list is in the form of an RGBA Nx4 array, where N is the length of the colormap.

```
print('viridis.colors', viridis.colors)
print('viridis(range(8))', viridis(range(8)))
print('viridis(np.linspace(0, 1, 8))', viridis(np.linspace(0, 1, 8)))
```

Out:

```
viridis.colors [[0.267004 0.004874 0.329415 1.      ]
 [0.275191 0.194905 0.496005 1.      ]
 [0.212395 0.359683 0.55171  1.      ]
 [0.153364 0.497    0.557724 1.      ]
 [0.122312 0.633153 0.530398 1.      ]
 [0.288921 0.758394 0.428426 1.      ]
 [0.626579 0.854645 0.223353 1.      ]
 [0.993248 0.906157 0.143936 1.      ]]
viridis(range(8)) [[0.267004 0.004874 0.329415 1.      ]
 [0.275191 0.194905 0.496005 1.      ]
 [0.212395 0.359683 0.55171  1.      ]
 [0.153364 0.497    0.557724 1.      ]
 [0.122312 0.633153 0.530398 1.      ]
 [0.288921 0.758394 0.428426 1.      ]
 [0.626579 0.854645 0.223353 1.      ]
 [0.993248 0.906157 0.143936 1.      ]]
viridis(np.linspace(0, 1, 8)) [[0.267004 0.004874 0.329415 1.      ]
 [0.275191 0.194905 0.496005 1.      ]
 [0.212395 0.359683 0.55171  1.      ]
 [0.153364 0.497    0.557724 1.      ]
 [0.122312 0.633153 0.530398 1.      ]
 [0.288921 0.758394 0.428426 1.      ]
 [0.626579 0.854645 0.223353 1.      ]
 [0.993248 0.906157 0.143936 1.      ]]
```

The colormap is a lookup table, so "oversampling" the colormap returns nearest-neighbor interpolation (note the repeated colors in the list below)

```
print('viridis(np.linspace(0, 1, 12))', viridis(np.linspace(0, 1, 12)))
```

Out:

```
viridis(np.linspace(0, 1, 12)) [[0.267004 0.004874 0.329415 1.      ]
 [0.267004 0.004874 0.329415 1.      ]
 [0.275191 0.194905 0.496005 1.      ]
 [0.212395 0.359683 0.55171  1.      ]
 [0.212395 0.359683 0.55171  1.      ]
 [0.153364 0.497    0.557724 1.      ]
 [0.122312 0.633153 0.530398 1.      ]
 [0.288921 0.758394 0.428426 1.      ]
 [0.288921 0.758394 0.428426 1.      ]
 [0.626579 0.854645 0.223353 1.      ]
 [0.993248 0.906157 0.143936 1.      ]
 [0.993248 0.906157 0.143936 1.      ]]
```

LinearSegmentedColormap

LinearSegmentedColormap s do not have a `.colors` attribute. However, one may still call the colormap with an integer array, or with a float array between 0 and 1.

```
copper = cm.get_cmap('copper', 8)

print('copper(range(8))', copper(range(8)))
print('copper(np.linspace(0, 1, 8))', copper(np.linspace(0, 1, 8)))
```

Out:

```
copper(range(8)) [[0.         0.         0.         1.        ]
 [0.17647055 0.1116     0.07107143 1.        ]
 [0.35294109 0.2232     0.14214286 1.        ]
 [0.52941164 0.3348     0.21321429 1.        ]
 [0.70588219 0.4464     0.28428571 1.        ]
 [0.88235273 0.558      0.35535714 1.        ]
 [1.         0.6696     0.42642857 1.        ]
 [1.         0.7812     0.4975     1.        ]]
copper(np.linspace(0, 1, 8)) [[0.         0.         0.         1.        ]
 [0.17647055 0.1116     0.07107143 1.        ]
 [0.35294109 0.2232     0.14214286 1.        ]
 [0.52941164 0.3348     0.21321429 1.        ]
 [0.70588219 0.4464     0.28428571 1.        ]
 [0.88235273 0.558      0.35535714 1.        ]
 [1.         0.6696     0.42642857 1.        ]
 [1.         0.7812     0.4975     1.        ]]
```

Creating listed colormaps

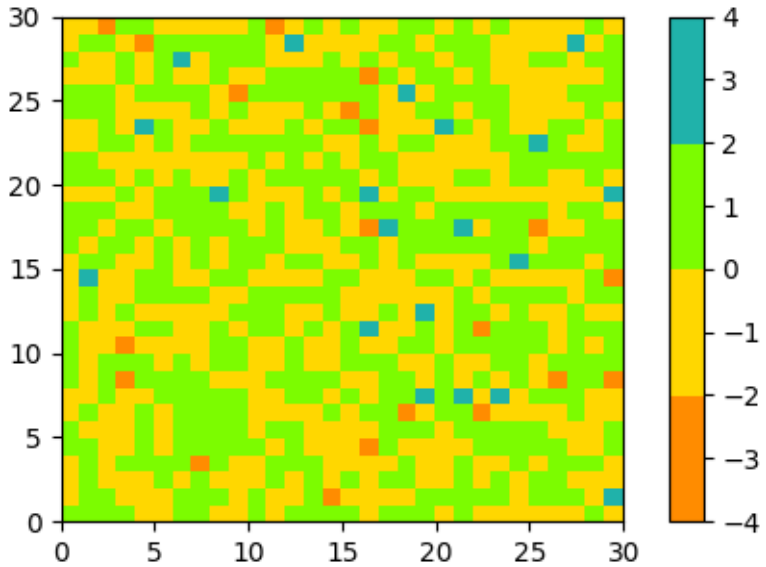
Creating a colormap is essentially the inverse operation of the above where we supply a list or array of color specifications to *ListedColormap* to make a new colormap.

Before continuing with the tutorial, let us define a helper function that takes one of more colormaps as input, creates some random data and applies the colormap(s) to an image plot of that dataset.

```
def plot_examples(colormaps):
    """
    Helper function to plot data with associated colormap.
    """
    np.random.seed(19680801)
    data = np.random.randn(30, 30)
    n = len(colormaps)
    fig, axs = plt.subplots(1, n, figsize=(n * 2 + 2, 3),
                            constrained_layout=True, squeeze=False)
    for [ax, cmap] in zip(axs.flat, colormaps):
        psm = ax.pcolormesh(data, cmap=cmap, rasterized=True, vmin=-4, vmax=4)
        fig.colorbar(psm, ax=ax)
    plt.show()
```

In the simplest case we might type in a list of color names to create a colormap from those.

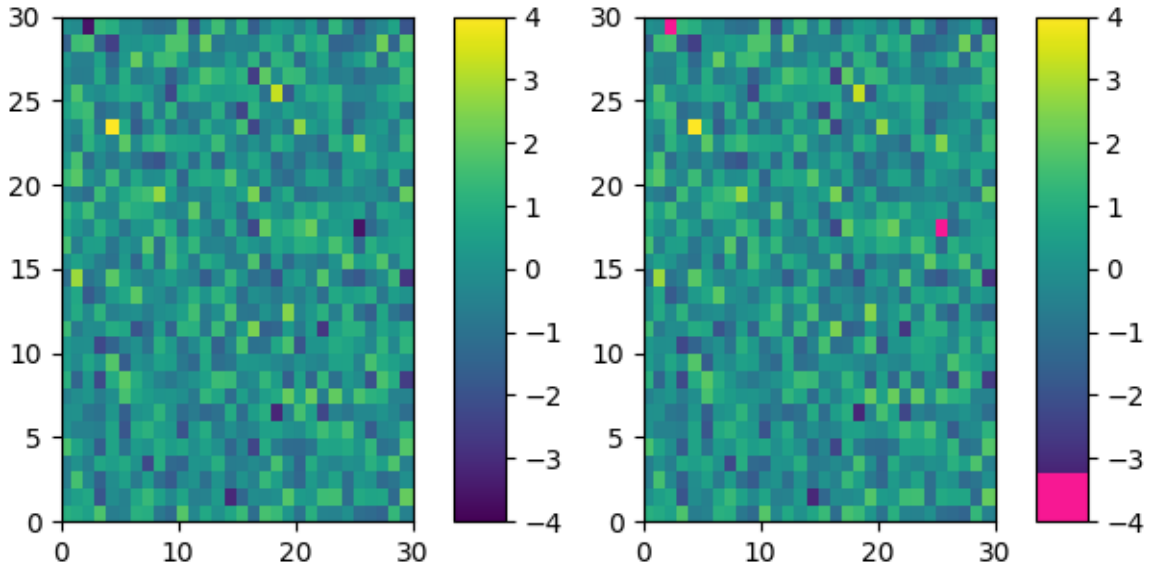
```
cmap = ListedColormap(["darkorange", "gold", "lawngreen", "lightseagreen"])
plot_examples([cmap])
```



In fact, that list may contain any valid *matplotlib color specification*. Particularly useful for creating custom colormaps are Nx4 numpy arrays. Because with the variety of numpy operations that we can do on a such an array, carpentry of new colormaps from existing colormaps become quite straight forward.

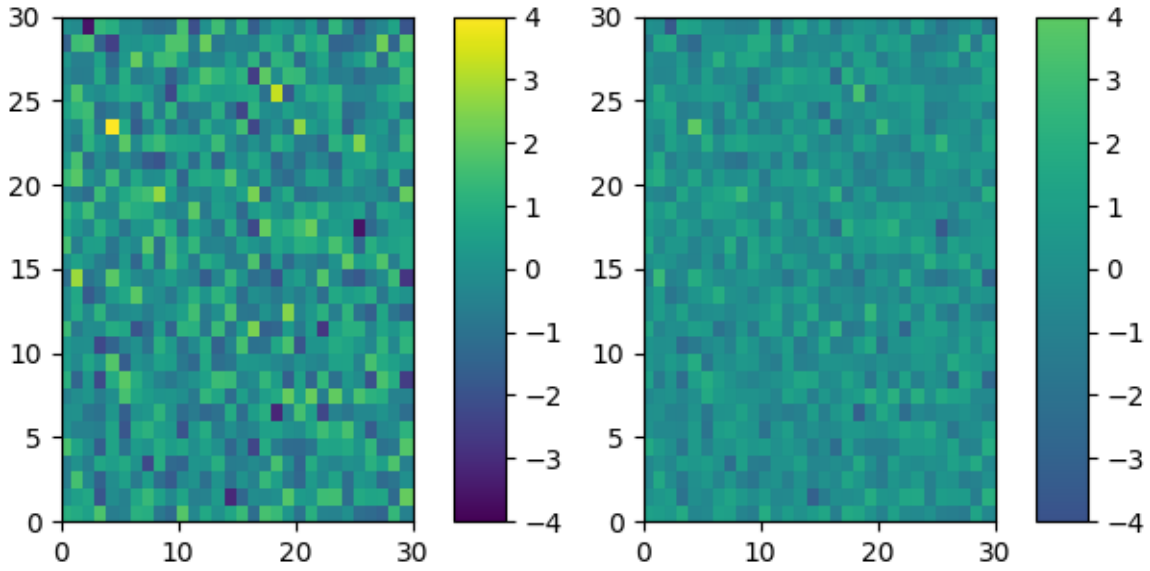
For example, suppose we want to make the first 25 entries of a 256-length "viridis" colormap pink for some reason:

```
viridis = cm.get_cmap('viridis', 256)
newcolors = viridis(np.linspace(0, 1, 256))
pink = np.array([248/256, 24/256, 148/256, 1])
newcolors[:25, :] = pink
newcmp = ListedColormap(newcolors)

plot_examples([viridis, newcmp])
```



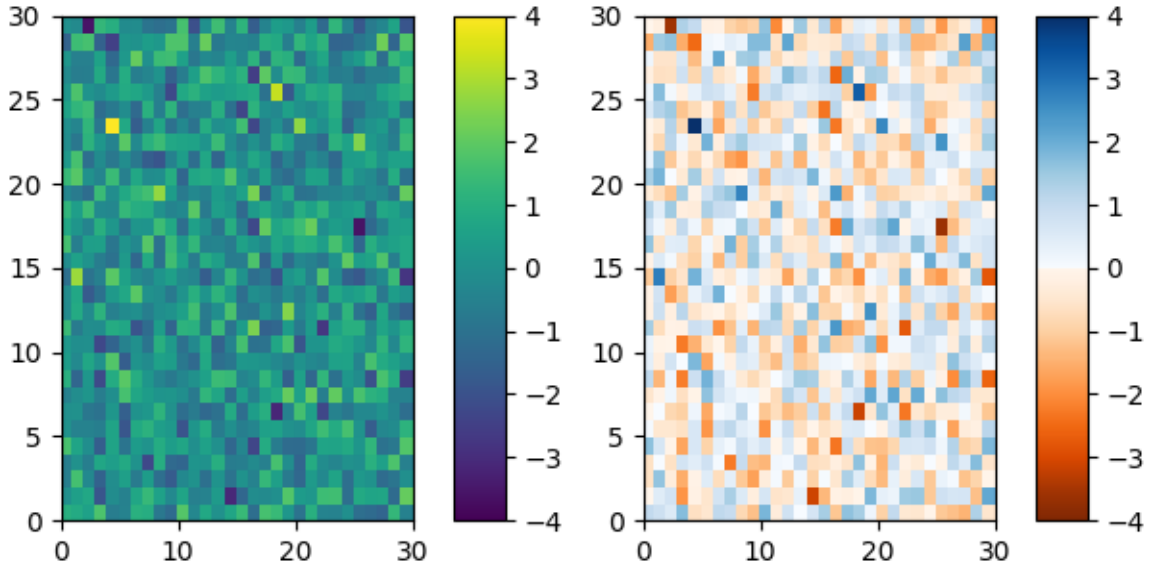
We can easily reduce the dynamic range of a colormap; here we choose the middle 0.5 of the colormap. However, we need to interpolate from a larger colormap, otherwise the new colormap will have repeated values.

```
viridis_big = cm.get_cmap('viridis', 512)
newcmp = ListedColormap(viridis_big(np.linspace(0.25, 0.75, 256)))
plot_examples([viridis, newcmp])
```



and we can easily concatenate two colormaps:

```
top = cm.get_cmap('Oranges_r', 128)
bottom = cm.get_cmap('Blues', 128)

newcolors = np.vstack((top(np.linspace(0, 1, 128)),
                       bottom(np.linspace(0, 1, 128))))
newcmp = ListedColormap(newcolors, name='OrangeBlue')
plot_examples([viridis, newcmp])
```



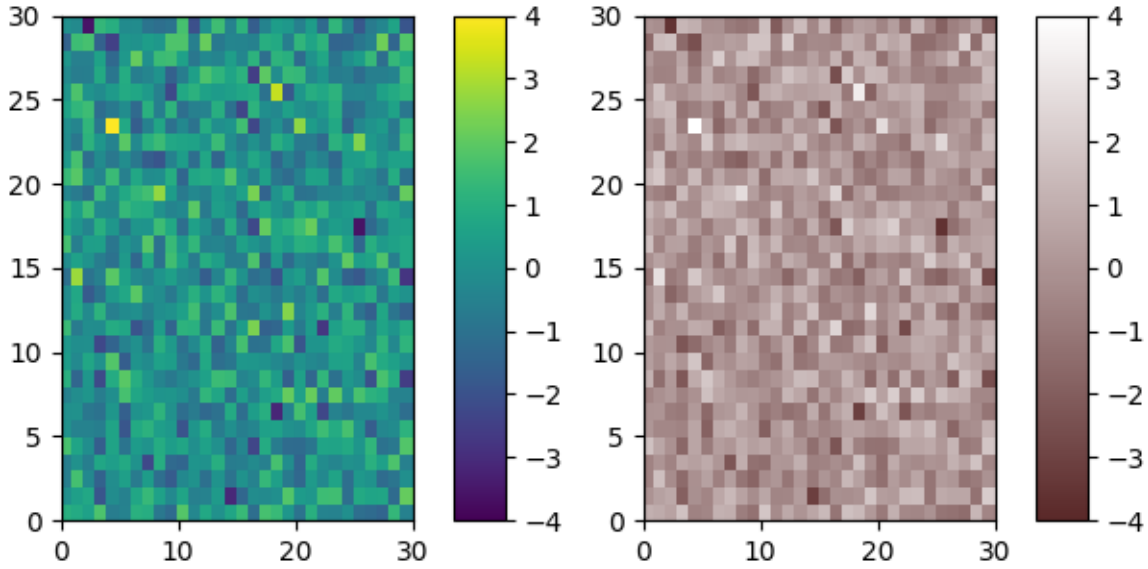
Of course we need not start from a named colormap, we just need to create the Nx4 array to pass to *ListedColormap*. Here we create a colormap that goes from brown (RGB: 90, 40, 40) to white (RGB: 255, 255, 255).

```
N = 256
```

(continues on next page)

(continued from previous page)

```
vals = np.ones((N, 4))
vals[:, 0] = np.linspace(90/256, 1, N)
vals[:, 1] = np.linspace(40/256, 1, N)
vals[:, 2] = np.linspace(40/256, 1, N)
newcmp = ListedColormap(vals)
plot_examples([viridis, newcmp])
```



Creating linear segmented colormaps

LinearSegmentedColormap class specifies colormaps using anchor points between which RGB(A) values are interpolated.

The format to specify these colormaps allows discontinuities at the anchor points. Each anchor point is specified as a row in a matrix of the form `[x[i] yleft[i] yright[i]]`, where `x[i]` is the anchor, and `yleft[i]` and `yright[i]` are the values of the color on either side of the anchor point.

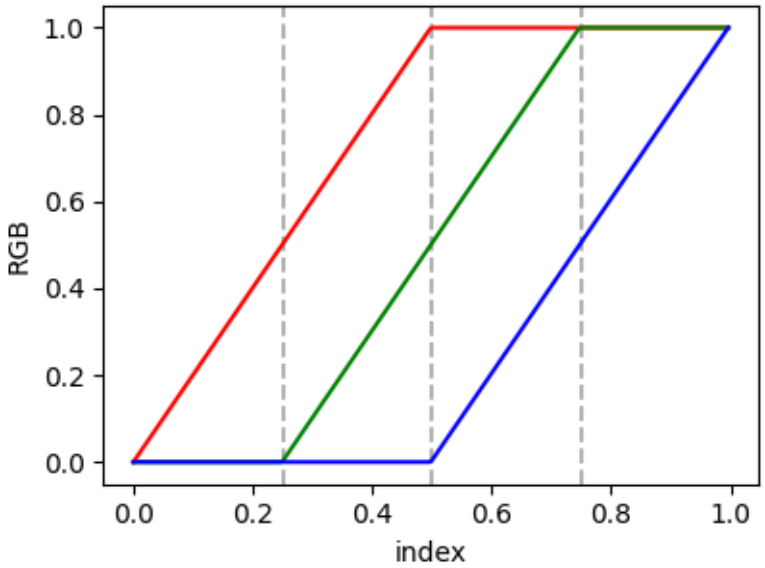
If there are no discontinuities, then `yleft[i]=yright[i]`:

```
cdict = {'red':   [[0.0,  0.0, 0.0],
                   [0.5,  1.0, 1.0],
                   [1.0,  1.0, 1.0]],
         'green': [[0.0,  0.0, 0.0],
                   [0.25, 0.0, 0.0],
                   [0.75, 1.0, 1.0],
                   [1.0,  1.0, 1.0]],
         'blue':  [[0.0,  0.0, 0.0],
                   [0.5,  0.0, 0.0],
                   [1.0,  1.0, 1.0]]}
```

(continues on next page)

(continued from previous page)

```
def plot_linearmap(cdict):
    newcmp = LinearSegmentedColormap('testCmap', segmentdata=cdict, N=256)
    rgba = newcmp(np.linspace(0, 1, 256))
    fig, ax = plt.subplots(figsize=(4, 3), constrained_layout=True)
    col = ['r', 'g', 'b']
    for xx in [0.25, 0.5, 0.75]:
        ax.axvline(xx, color='0.7', linestyle='--')
    for i in range(3):
        ax.plot(np.arange(256)/256, rgba[:, i], color=col[i])
    ax.set_xlabel('index')
    ax.set_ylabel('RGB')
    plt.show()

plot_linearmap(cdict)
```



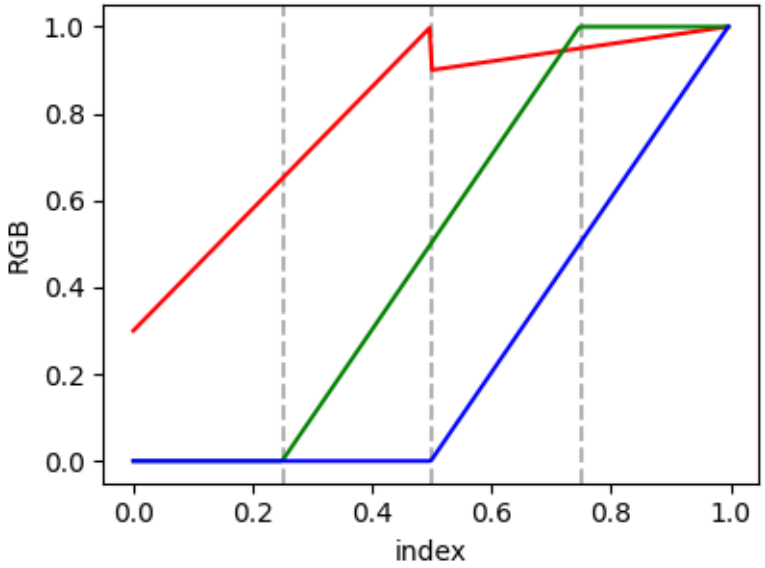
In order to make a discontinuity at an anchor point, the third column is different than the second. The matrix for each of "red", "green", "blue", and optionally "alpha" is set up as:

```
cdict['red'] = [...
                [x[i]      yleft[i]     yright[i]],
                [x[i+1]    yleft[i+1]   yright[i+1]],
               ...]
```

and for values passed to the colormap between `x[i]` and `x[i+1]`, the interpolation is between `yright[i]` and `yleft[i+1]`.

In the example below there is a discontinuity in red at 0.5. The interpolation between 0 and 0.5 goes from 0.3 to 1, and between 0.5 and 1 it goes from 0.9 to 1. Note that red[0, 1], and red[2, 2] are both superfluous to the interpolation because red[0, 1] is the value to the left of 0, and red[2, 2] is the value to the right of 1.0.

```
cdict['red'] = [[0.0,  0.0, 0.3],
                [0.5,  1.0, 0.9],
                [1.0,  1.0, 1.0]]
plot_linearmap(cdict)
```



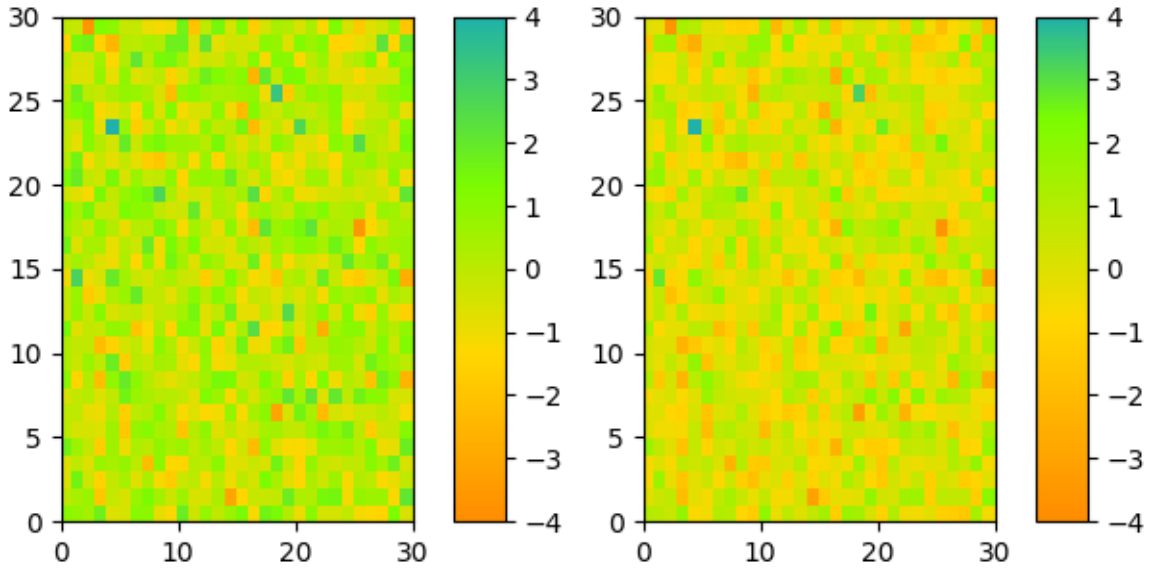
Directly creating a segmented colormap from a list

The above described is a very versatile approach, but admittedly a bit cumbersome to implement. For some basic cases, the use of *LinearSegmentedColormap.from_list* may be easier. This creates a segmented colormap with equal spacings from a supplied list of colors.

```
colors = ["darkorange", "gold", "lawngreen", "lightseagreen"]
cmap1 = LinearSegmentedColormap.from_list("mycmap", colors)
```

If desired, the nodes of the colormap can be given as numbers between 0 and 1. E.g. one could have the reddish part take more space in the colormap.

```
nodes = [0.0, 0.4, 0.8, 1.0]
cmap2 = LinearSegmentedColormap.from_list("mycmap", list(zip(nodes, colors)))

plot_examples([cmap1, cmap2])
```



References

The use of the following functions, methods, classes and modules is shown in this example:

```
import matplotlib
matplotlib.axes.Axes.pcolormesh
matplotlib.figure.Figure.colorbar
matplotlib.colors
matplotlib.colors.LinearSegmentedColormap
matplotlib.colors.ListedColormap
matplotlib.cm
matplotlib.cm.get_cmap
```

Out:

```
<function get_cmap at 0x7f5434bc43a0>
```

Total running time of the script: (0 minutes 2.681 seconds)

2.4.4 Colormap Normalization

Objects that use colormaps by default linearly map the colors in the colormap from data values *vmin* to *vmax*. For example:

```
pcm = ax.pcolormesh(x, y, Z, vmin=-1., vmax=1., cmap='RdBu_r')
```

will map the data in *Z* linearly from -1 to +1, so *Z=0* will give a color at the center of the colormap *RdBu_r* (white in this case).

Matplotlib does this mapping in two steps, with a normalization from the input data to [0, 1] occurring first, and then mapping onto the indices in the colormap. Normalizations are classes defined in the *matplotlib.colors()* module. The default, linear normalization is *matplotlib.colors.Normalize()*.

Artists that map data to color pass the arguments *vmin* and *vmax* to construct a *matplotlib.colors.Normalize()* instance, then call it:

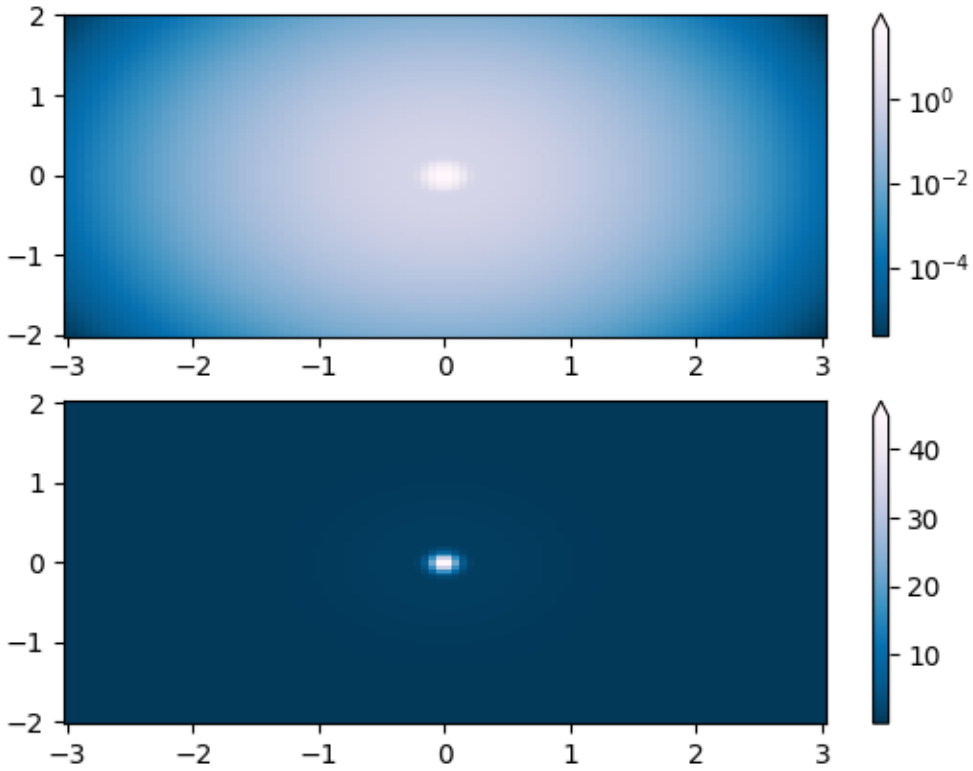
```
In [1]: import matplotlib as mpl

In [2]: norm = mpl.colors.Normalize(vmin=-1, vmax=1)

In [3]: norm(0)
Out[3]: 0.5
```

However, there are sometimes cases where it is useful to map data to colormaps in a non-linear fashion.

Logarithmic

One of the most common transformations is to plot data by taking its logarithm (to the base-10). This transformation is useful to display changes across disparate scales. Using *colors.LogNorm* normalizes the data via log_{10}. In the example below, there are two bumps, one much smaller than the other. Using *colors.LogNorm*, the shape and location of each bump can clearly be seen:

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.colors as colors
import matplotlib.cbook as cbook

N = 100
X, Y = np.mgrid[-3:3:complex(0, N), -2:2:complex(0, N)]

# A low hump with a spike coming out of the top right.  Needs to have
# z/colour axis on a log scale so we see both hump and spike.  linear
# scale only shows the spike.
Z1 = np.exp(-X**2 - Y**2)
Z2 = np.exp(-(X * 10)**2 - (Y * 10)**2)
Z = Z1 + 50 * Z2

fig, ax = plt.subplots(2, 1)

pcm = ax[0].pcolor(X, Y, Z,
                   norm=colors.LogNorm(vmin=Z.min(), vmax=Z.max()),
                   cmap='PuBu_r', shading='auto')
fig.colorbar(pcm, ax=ax[0], extend='max')

pcm = ax[1].pcolor(X, Y, Z, cmap='PuBu_r', shading='auto')
fig.colorbar(pcm, ax=ax[1], extend='max')
plt.show()
```



Symmetric logarithmic

Similarly, it sometimes happens that there is data that is positive and negative, but we would still like a logarithmic scaling applied to both. In this case, the negative numbers are also scaled logarithmically, and mapped to smaller numbers; e.g., if `vmin=-vmax`, then the negative numbers are mapped from 0 to 0.5 and the positive from 0.5 to 1.

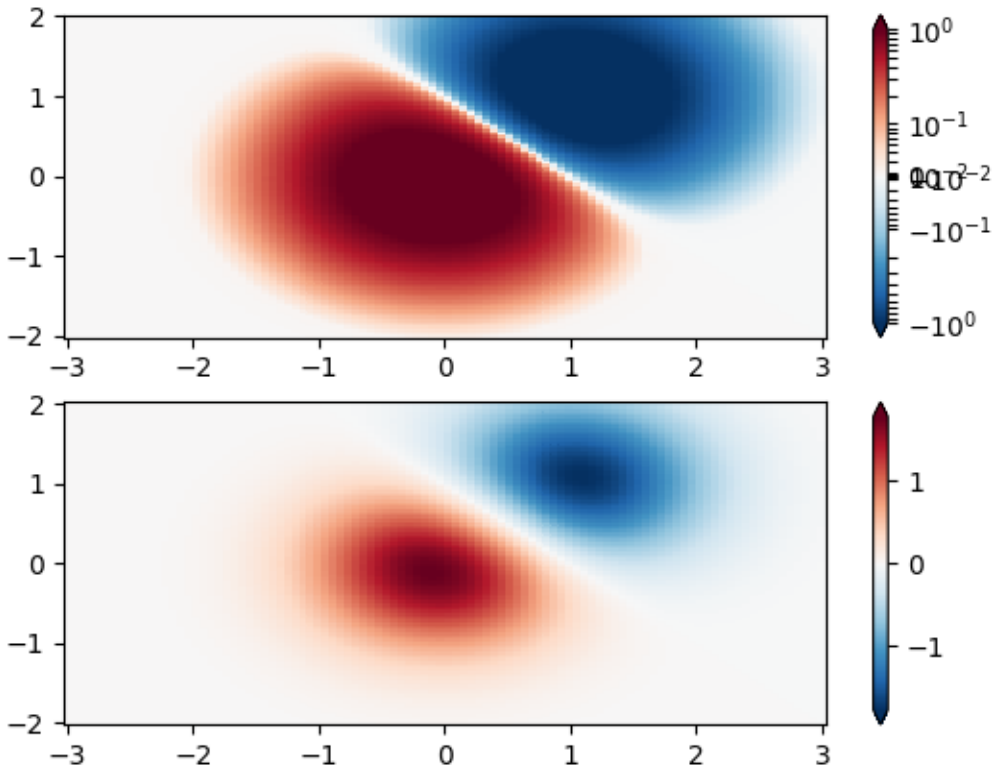
Since the logarithm of values close to zero tends toward infinity, a small range around zero needs to be mapped linearly. The parameter *linthresh* allows the user to specify the size of this range (*-linthresh*, *linthresh*). The size of this range in the colormap is set by *linscale*. When *linscale* == 1.0 (the default), the space used for the positive and negative halves of the linear range will be equal to one decade in the logarithmic range.

```
N = 100
X, Y = np.mgrid[-3:3:complex(0, N), -2:2:complex(0, N)]
Z1 = np.exp(-X**2 - Y**2)
Z2 = np.exp(-(X - 1)**2 - (Y - 1)**2)
Z = (Z1 - Z2) * 2
```

(continues on next page)

(continued from previous page)

```
fig, ax = plt.subplots(2, 1)

pcm = ax[0].pcolormesh(X, Y, Z,
                       norm=colors.SymLogNorm(linthresh=0.03, linscale=0.03,
                                              vmin=-1.0, vmax=1.0, base=10),
                       cmap='RdBu_r', shading='auto')
fig.colorbar(pcm, ax=ax[0], extend='both')

pcm = ax[1].pcolormesh(X, Y, Z, cmap='RdBu_r', vmin=-np.max(Z), shading='auto')
fig.colorbar(pcm, ax=ax[1], extend='both')
plt.show()
```

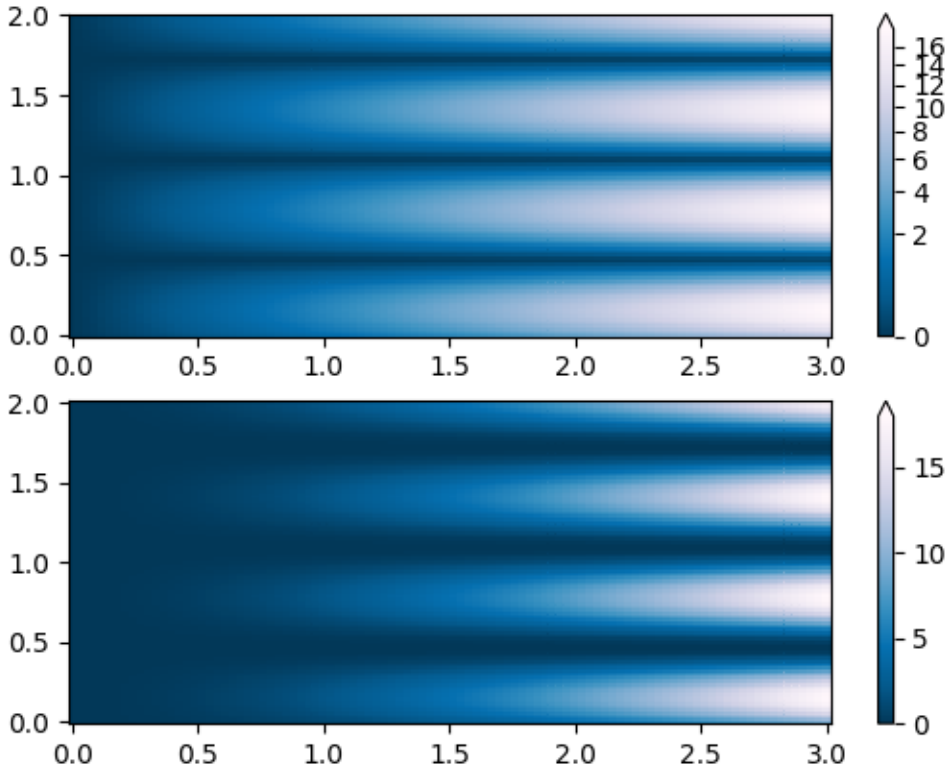


Power-law

Sometimes it is useful to remap the colors onto a power-law relationship (i.e. $y = x^{\gamma}$, where γ is the power). For this we use the *colors.PowerNorm*. It takes as an argument *gamma* (*gamma* == 1.0 will just yield the default linear normalization):

Note: There should probably be a good reason for plotting the data using this type of transformation. Technical viewers are used to linear and logarithmic axes and data transformations. Power laws are less common, and viewers should explicitly be made aware that they have been used.

```
N = 100
X, Y = np.mgrid[0:3:complex(0, N), 0:2:complex(0, N)]
Z1 = (1 + np.sin(Y * 10.)) * X**2

fig, ax = plt.subplots(2, 1)

pcm = ax[0].pcolormesh(X, Y, Z1, norm=colors.PowerNorm(gamma=0.5),
                       cmap='PuBu_r', shading='auto')
fig.colorbar(pcm, ax=ax[0], extend='max')

pcm = ax[1].pcolormesh(X, Y, Z1, cmap='PuBu_r', shading='auto')
fig.colorbar(pcm, ax=ax[1], extend='max')
plt.show()
```



Discrete bounds

Another normalization that comes with Matplotlib is *colors.BoundaryNorm*. In addition to *vmin* and *vmax*, this takes as arguments boundaries between which data is to be mapped. The colors are then linearly distributed between these "bounds". It can also take an *extend* argument to add upper and/or lower out-of-bounds values to the range over which the colors are distributed. For instance:

```
In [4]: import matplotlib.colors as colors

In [5]: bounds = np.array([-0.25, -0.125, 0, 0.5, 1])

In [6]: norm = colors.BoundaryNorm(boundaries=bounds, ncolors=4)

In [7]: print(norm([-0.2, -0.15, -0.02, 0.3, 0.8, 0.99]))
[0 0 1 2 3 3]
```

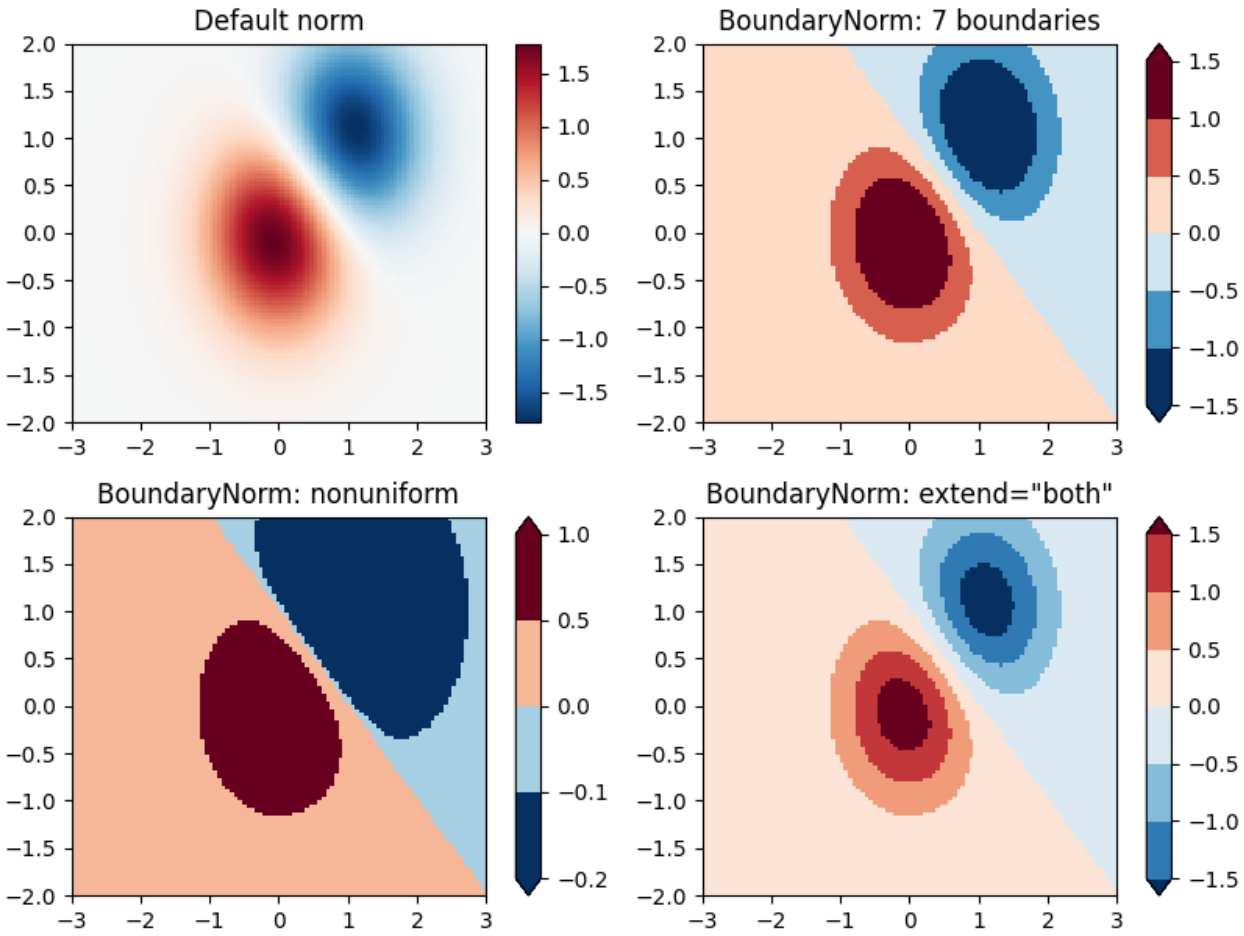
Note: Unlike the other norms, this norm returns values from 0 to *ncolors*-1.

```
N = 100
X, Y = np.meshgrid(np.linspace(-3, 3, N), np.linspace(-2, 2, N))
```

(continues on next page)

(continued from previous page)

```
Z1 = np.exp(-X**2 - Y**2)
Z2 = np.exp(-(X - 1)**2 - (Y - 1)**2)
Z = ((Z1 - Z2) * 2)[:-1, :-1]

fig, ax = plt.subplots(2, 2, figsize=(8, 6), constrained_layout=True)
ax = ax.flatten()

# Default norm:
pcm = ax[0].pcolormesh(X, Y, Z, cmap='RdBu_r')
fig.colorbar(pcm, ax=ax[0], orientation='vertical')
ax[0].set_title('Default norm')

# Even bounds give a contour-like effect:
bounds = np.linspace(-1.5, 1.5, 7)
norm = colors.BoundaryNorm(boundaries=bounds, ncolors=256)
pcm = ax[1].pcolormesh(X, Y, Z, norm=norm, cmap='RdBu_r')
fig.colorbar(pcm, ax=ax[1], extend='both', orientation='vertical')
ax[1].set_title('BoundaryNorm: 7 boundaries')

# Bounds may be unevenly spaced:
bounds = np.array([-0.2, -0.1, 0, 0.5, 1])
norm = colors.BoundaryNorm(boundaries=bounds, ncolors=256)
pcm = ax[2].pcolormesh(X, Y, Z, norm=norm, cmap='RdBu_r')
fig.colorbar(pcm, ax=ax[2], extend='both', orientation='vertical')
ax[2].set_title('BoundaryNorm: nonuniform')

# With out-of-bounds colors:
bounds = np.linspace(-1.5, 1.5, 7)
norm = colors.BoundaryNorm(boundaries=bounds, ncolors=256, extend='both')
pcm = ax[3].pcolormesh(X, Y, Z, norm=norm, cmap='RdBu_r')
# The colorbar inherits the "extend" argument from BoundaryNorm.
fig.colorbar(pcm, ax=ax[3], orientation='vertical')
ax[3].set_title('BoundaryNorm: extend="both"')
plt.show()
```



TwoSlopeNorm: Different mapping on either side of a center

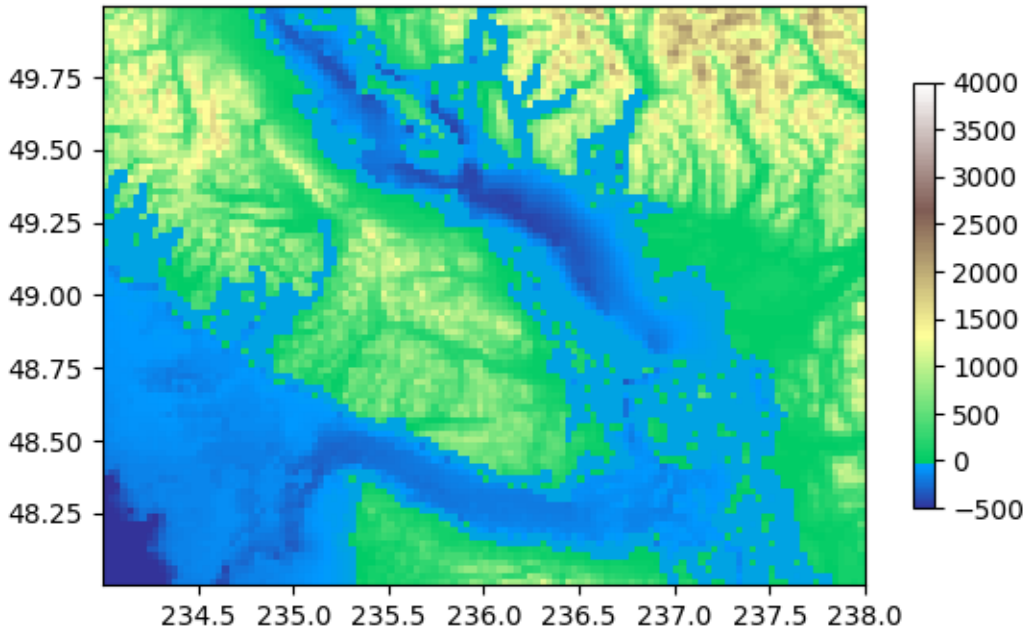
Sometimes we want to have a different colormap on either side of a conceptual center point, and we want those two colormaps to have different linear scales. An example is a topographic map where the land and ocean have a center at zero, but land typically has a greater elevation range than the water has depth range, and they are often represented by a different colormap.

```
dem = cbook.get_sample_data('topobathy.npz', np_load=True)
topo = dem['topo']
longitude = dem['longitude']
latitude = dem['latitude']

fig, ax = plt.subplots()
# make a colormap that has land and ocean clearly delineated and of the
# same length (256 + 256)
colors_undersea = plt.cm.terrain(np.linspace(0, 0.17, 256))
colors_land = plt.cm.terrain(np.linspace(0.25, 1, 256))
all_colors = np.vstack((colors_undersea, colors_land))
terrain_map = colors.LinearSegmentedColormap.from_list(
```

(continues on next page)

(continued from previous page)

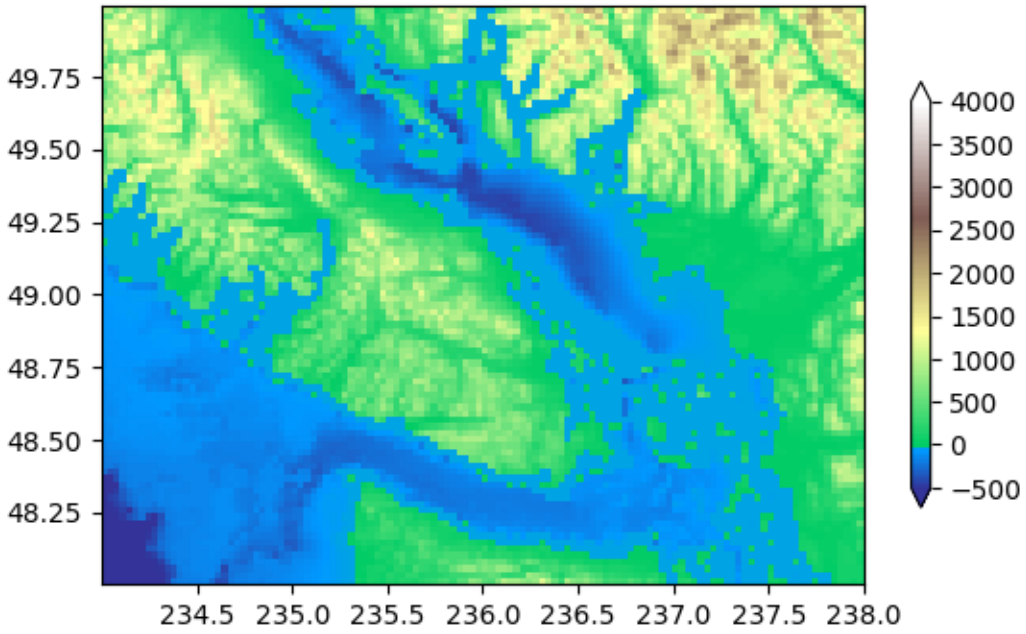
```
    'terrain_map', all_colors)

# make the norm:  Note the center is offset so that the land has more
# dynamic range:
divnorm = colors.TwoSlopeNorm(vmin=-500., vcenter=0, vmax=4000)

pcm = ax.pcolormesh(longitude, latitude, topo, rasterized=True, norm=divnorm,
                    cmap=terrain_map, shading='auto')
# Simple geographic plot, set aspect ratio beecause distance between lines of
# longitude depends on latitude.
ax.set_aspect(1 / np.cos(np.deg2rad(49)))
fig.colorbar(pcm, shrink=0.6)
plt.show()
```



Custom normalization: Manually implement two linear ranges

The *TwoSlopeNorm* described above makes a useful example for defining your own norm.

```
class MidpointNormalize(colors.Normalize):
    def __init__(self, vmin=None, vmax=None, vcenter=None, clip=False):
        self.vcenter = vcenter
        colors.Normalize.__init__(self, vmin, vmax, clip)

    def __call__(self, value, clip=None):
        # I'm ignoring masked values and all kinds of edge cases to make a
        # simple example...
        x, y = [self.vmin, self.vcenter, self.vmax], [0, 0.5, 1]
        return np.ma.masked_array(np.interp(value, x, y))


fig, ax = plt.subplots()
midnorm = MidpointNormalize(vmin=-500., vcenter=0, vmax=4000)

pcm = ax.pcolormesh(longitude, latitude, topo, rasterized=True, norm=midnorm,
                    cmap=terrain_map, shading='auto')
ax.set_aspect(1 / np.cos(np.deg2rad(49)))
fig.colorbar(pcm, shrink=0.6, extend='both')
plt.show()
```



Total running time of the script: (0 minutes 3.466 seconds)

2.4.5 Choosing Colormaps in Matplotlib

Matplotlib has a number of built-in colormaps accessible via *matplotlib.cm.get_cmap*. There are also external libraries like [palettable] and [colorcet] that have many extra colormaps. Here we briefly discuss how to choose between the many options. For help on creating your own colormaps, see *Creating Colormaps in Matplotlib*.

Overview

The idea behind choosing a good colormap is to find a good representation in 3D colorspace for your data set. The best colormap for any given data set depends on many things including:

- Whether representing form or metric data ([Ware])
- Your knowledge of the data set (*e.g.*, is there a critical value from which the other values deviate?)
- If there is an intuitive color scheme for the parameter you are plotting

- If there is a standard in the field the audience may be expecting

For many applications, a perceptually uniform colormap is the best choice; i.e. a colormap in which equal steps in data are perceived as equal steps in the color space. Researchers have found that the human brain perceives changes in the lightness parameter as changes in the data much better than, for example, changes in hue. Therefore, colormaps which have monotonically increasing lightness through the colormap will be better interpreted by the viewer. A wonderful example of perceptually uniform colormaps is [colorcet].

Color can be represented in 3D space in various ways. One way to represent color is using CIELAB. In CIELAB, color space is represented by lightness, L^*; red-green, a^*; and yellow-blue, b^*. The lightness parameter L^* can then be used to learn more about how the matplotlib colormaps will be perceived by viewers.

An excellent starting resource for learning about human perception of colormaps is from [IBM].

Classes of colormaps

Colormaps are often split into several categories based on their function (see, *e.g.*, [Moreland]):

1. Sequential: change in lightness and often saturation of color incrementally, often using a single hue; should be used for representing information that has ordering.
2. Diverging: change in lightness and possibly saturation of two different colors that meet in the middle at an unsaturated color; should be used when the information being plotted has a critical middle value, such as topography or when the data deviates around zero.
3. Cyclic: change in lightness of two different colors that meet in the middle and beginning/end at an unsaturated color; should be used for values that wrap around at the endpoints, such as phase angle, wind direction, or time of day.
4. Qualitative: often are miscellaneous colors; should be used to represent information which does not have ordering or relationships.

```
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
from matplotlib import cm
from colorspacious import cspace_converter
from collections import OrderedDict

cmaps = OrderedDict()
```

Sequential

For the Sequential plots, the lightness value increases monotonically through the colormaps. This is good. Some of the L^* values in the colormaps span from 0 to 100 (binary and the other grayscale), and others start around $L^* = 20$. Those that have a smaller range of L^* will accordingly have a smaller perceptual range. Note also that the L^* function varies amongst the colormaps: some are approximately linear in L^* and others are more curved.

```
cmaps['Perceptually Uniform Sequential'] = [
            'viridis', 'plasma', 'inferno', 'magma', 'cividis']

cmaps['Sequential'] = [
            'Greys', 'Purples', 'Blues', 'Greens', 'Oranges', 'Reds',
            'YlOrBr', 'YlOrRd', 'OrRd', 'PuRd', 'RdPu', 'BuPu',
            'GnBu', 'PuBu', 'YlGnBu', 'PuBuGn', 'BuGn', 'YlGn']
```

Sequential2

Many of the L^* values from the Sequential2 plots are monotonically increasing, but some (autumn, cool, spring, and winter) plateau or even go both up and down in L^* space. Others (afmhot, copper, gist_heat, and hot) have kinks in the L^* functions. Data that is being represented in a region of the colormap that is at a plateau or kink will lead to a perception of banding of the data in those values in the colormap (see [mycarta-banding] for an excellent example of this).

```
cmaps['Sequential (2)'] = [
            'binary', 'gist_yarg', 'gist_gray', 'gray', 'bone', 'pink',
            'spring', 'summer', 'autumn', 'winter', 'cool', 'Wistia',
            'hot', 'afmhot', 'gist_heat', 'copper']
```

Diverging

For the Diverging maps, we want to have monotonically increasing L^* values up to a maximum, which should be close to $L^* = 100$, followed by monotonically decreasing L^* values. We are looking for approximately equal minimum L^* values at opposite ends of the colormap. By these measures, BrBG and RdBu are good options. coolwarm is a good option, but it doesn't span a wide range of L^* values (see grayscale section below).

```
cmaps['Diverging'] = [
            'PiYG', 'PRGn', 'BrBG', 'PuOr', 'RdGy', 'RdBu',
            'RdYlBu', 'RdYlGn', 'Spectral', 'coolwarm', 'bwr', 'seismic']
```

Cyclic

For Cyclic maps, we want to start and end on the same color, and meet a symmetric center point in the middle. L^* should change monotonically from start to middle, and inversely from middle to end. It should be symmetric on the increasing and decreasing side, and only differ in hue. At the ends and middle, L^* will reverse direction, which should be smoothed in L^* space to reduce artifacts. See [kovesi-colormaps] for more information on the design of cyclic maps.

The often-used HSV colormap is included in this set of colormaps, although it is not symmetric to a center point. Additionally, the L^* values vary widely throughout the colormap, making it a poor choice for representing data for viewers to see perceptually. See an extension on this idea at [mycarta-jet].

```
cmaps['Cyclic'] = ['twilight', 'twilight_shifted', 'hsv']
```

Qualitative

Qualitative colormaps are not aimed at being perceptual maps, but looking at the lightness parameter can verify that for us. The L^* values move all over the place throughout the colormap, and are clearly not monotonically increasing. These would not be good options for use as perceptual colormaps.

```
cmaps['Qualitative'] = ['Pastel1', 'Pastel2', 'Paired', 'Accent',
                        'Dark2', 'Set1', 'Set2', 'Set3',
                        'tab10', 'tab20', 'tab20b', 'tab20c']
```

Miscellaneous

Some of the miscellaneous colormaps have particular uses for which they have been created. For example, gist_earth, ocean, and terrain all seem to be created for plotting topography (green/brown) and water depths (blue) together. We would expect to see a divergence in these colormaps, then, but multiple kinks may not be ideal, such as in gist_earth and terrain. CMRmap was created to convert well to grayscale, though it does appear to have some small kinks in L^*. cubehelix was created to vary smoothly in both lightness and hue, but appears to have a small hump in the green hue area. turbo was created to display depth and disparity data.

The often-used jet colormap is included in this set of colormaps. We can see that the L^* values vary widely throughout the colormap, making it a poor choice for representing data for viewers to see perceptually. See an extension on this idea at [mycarta-jet] and [turbo].

```
cmaps['Miscellaneous'] = [
            'flag', 'prism', 'ocean', 'gist_earth', 'terrain', 'gist_stern',
            'gnuplot', 'gnuplot2', 'CMRmap', 'cubehelix', 'brg',
```

(continues on next page)

(continued from previous page)

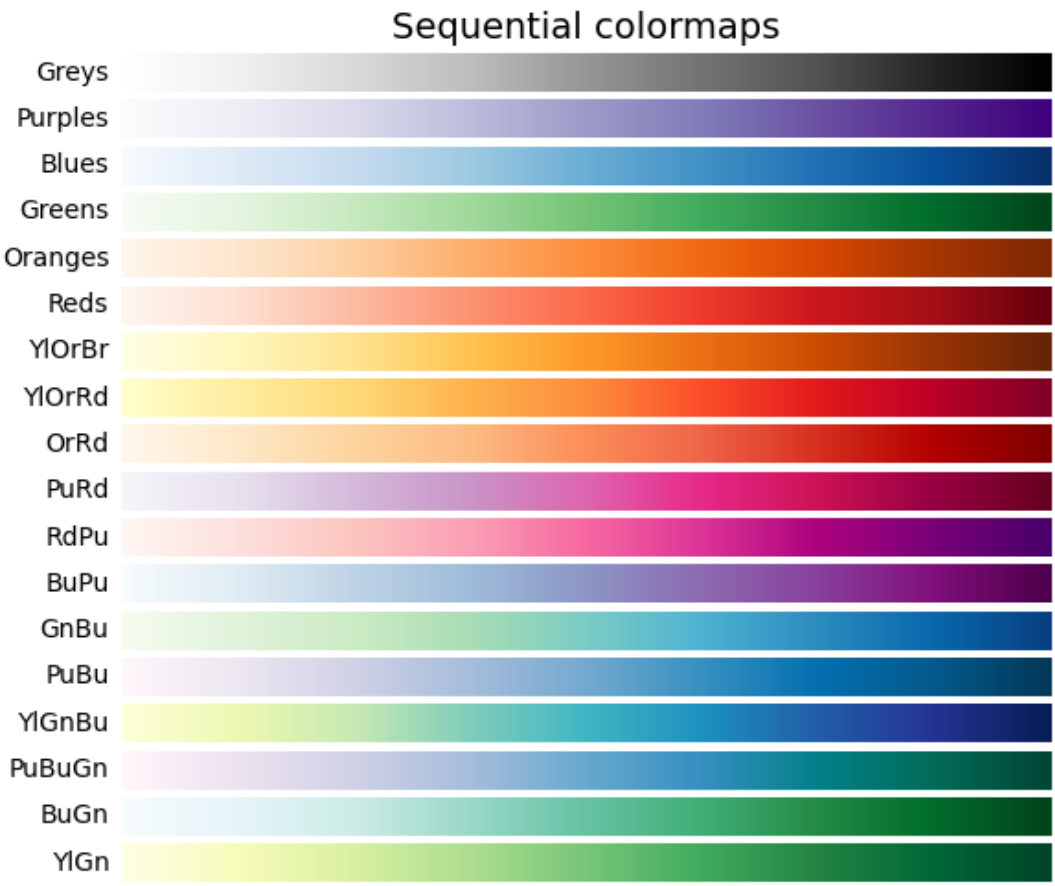
```
            'gist_rainbow', 'rainbow', 'jet', 'turbo', 'nipy_spectral',
            'gist_ncar']
```

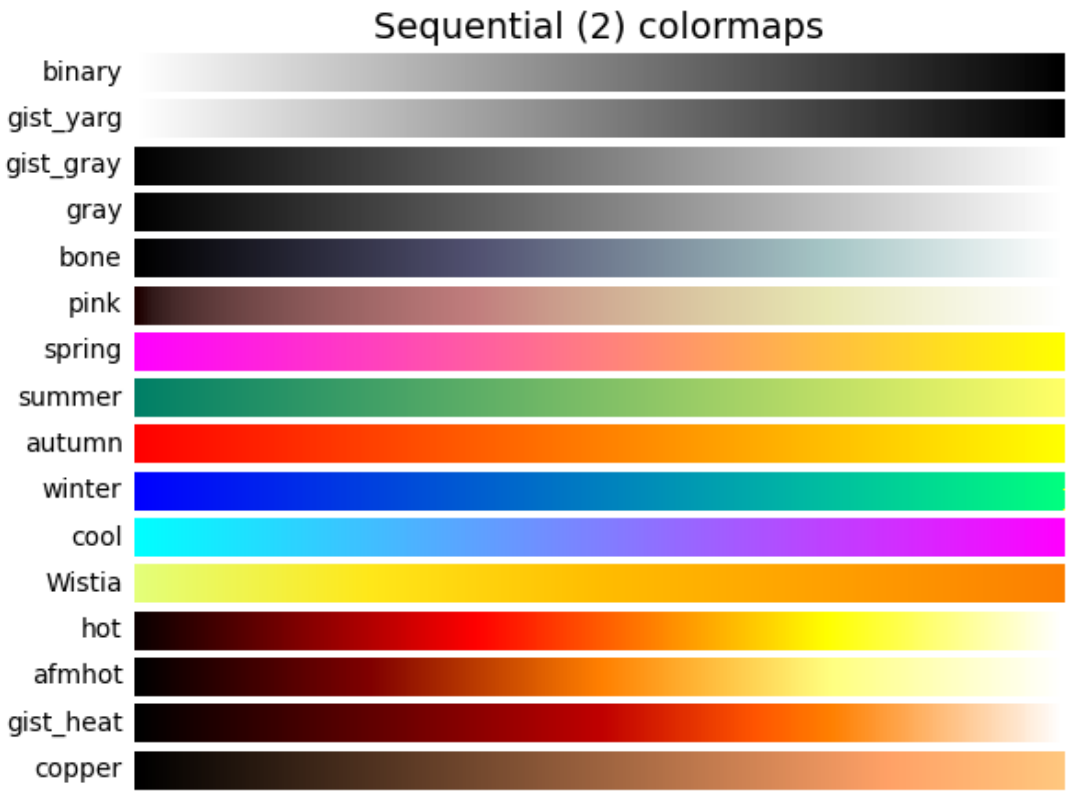
First, we'll show the range of each colormap. Note that some seem to change more "quickly" than others.

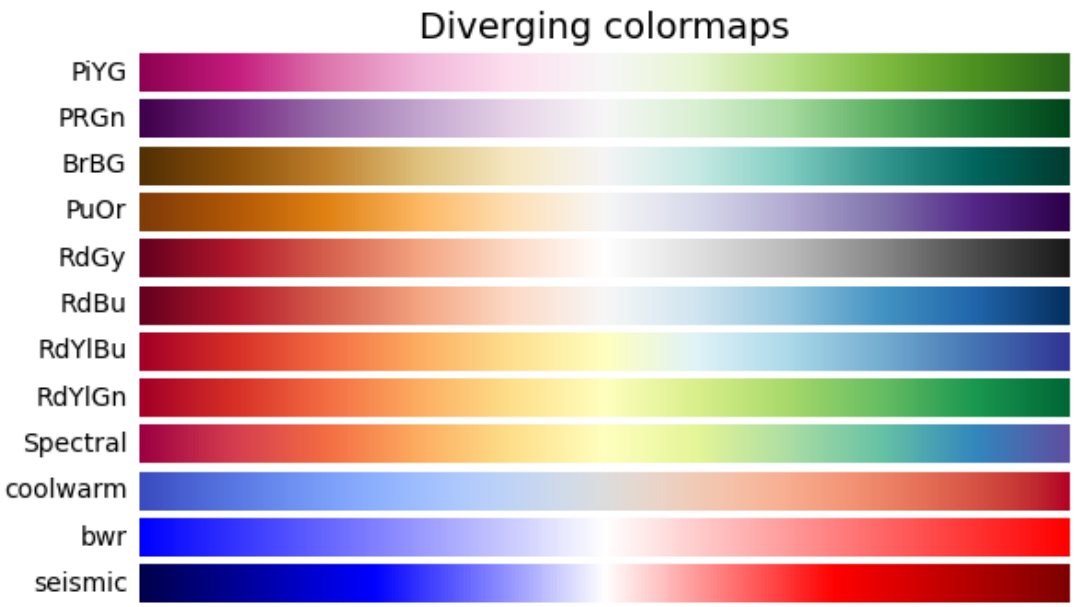
```
nrows = max(len(cmap_list) for cmap_category, cmap_list in cmaps.items())
gradient = np.linspace(0, 1, 256)
gradient = np.vstack((gradient, gradient))


def plot_color_gradients(cmap_category, cmap_list, nrows):
    fig, axes = plt.subplots(nrows=nrows)
    fig.subplots_adjust(top=0.95, bottom=0.01, left=0.2, right=0.99)
    axes[0].set_title(cmap_category + ' colormaps', fontsize=14)

    for ax, name in zip(axes, cmap_list):
        ax.imshow(gradient, aspect='auto', cmap=plt.get_cmap(name))
        pos = list(ax.get_position().bounds)
        x_text = pos[0] - 0.01
        y_text = pos[1] + pos[3]/2.
        fig.text(x_text, y_text, name, va='center', ha='right', fontsize=10)

    # Turn off *all* ticks & spines, not just the ones with colormaps.
    for ax in axes:
        ax.set_axis_off()


for cmap_category, cmap_list in cmaps.items():
    plot_color_gradients(cmap_category, cmap_list, nrows)

plt.show()
```


Perceptually Uniform Sequential colormaps
viridis
plasma
inferno
magma
cividis

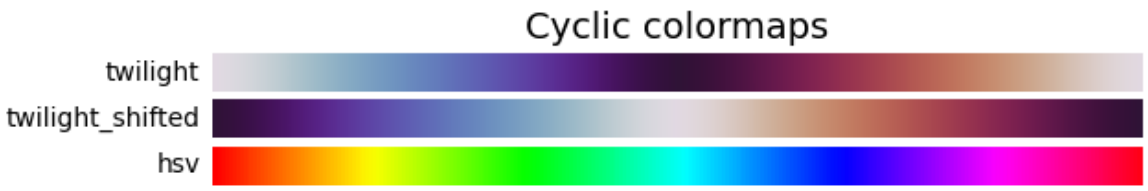
-


Sequential colormaps
Greys
Purples
Blues
Greens
Oranges
Reds
YlOrBr
YlOrRd
OrRd
PuRd
RdPu
BuPu
GnBu
PuBu
YlGnBu
PuBuGn
BuGn
YlGn

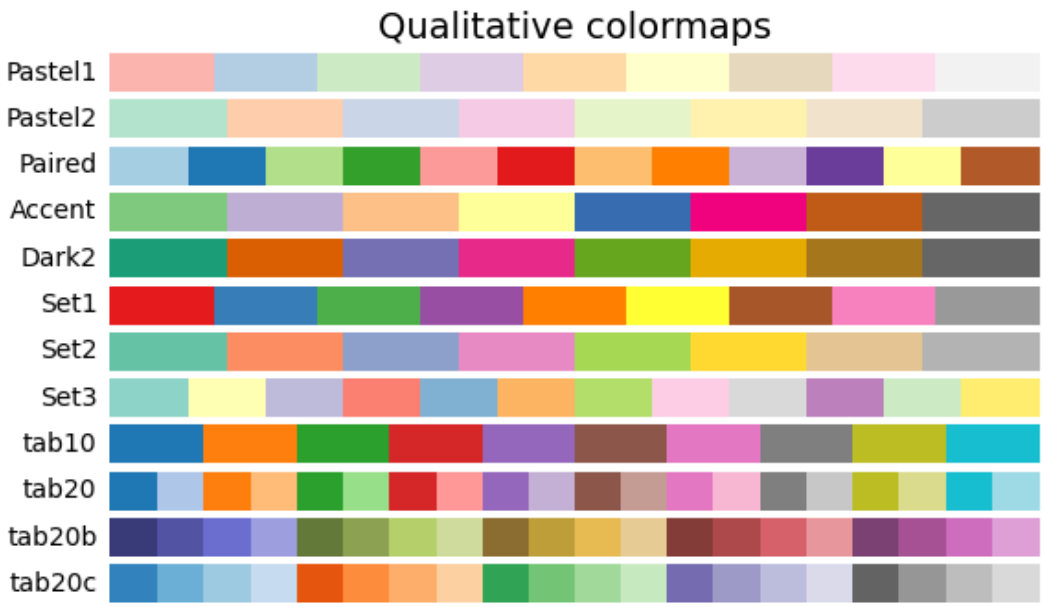

Sequential (2) colormaps
binary
gist_yarg
gist_gray
gray
bone
pink
spring
summer
autumn
winter
cool
Wistia
hot
afmhot
gist_heat
copper


Diverging colormaps
PiYG
PRGn
BrBG
PuOr
RdGy
RdBu
RdYlBu
RdYlGn
Spectral
coolwarm
bwr
seismic

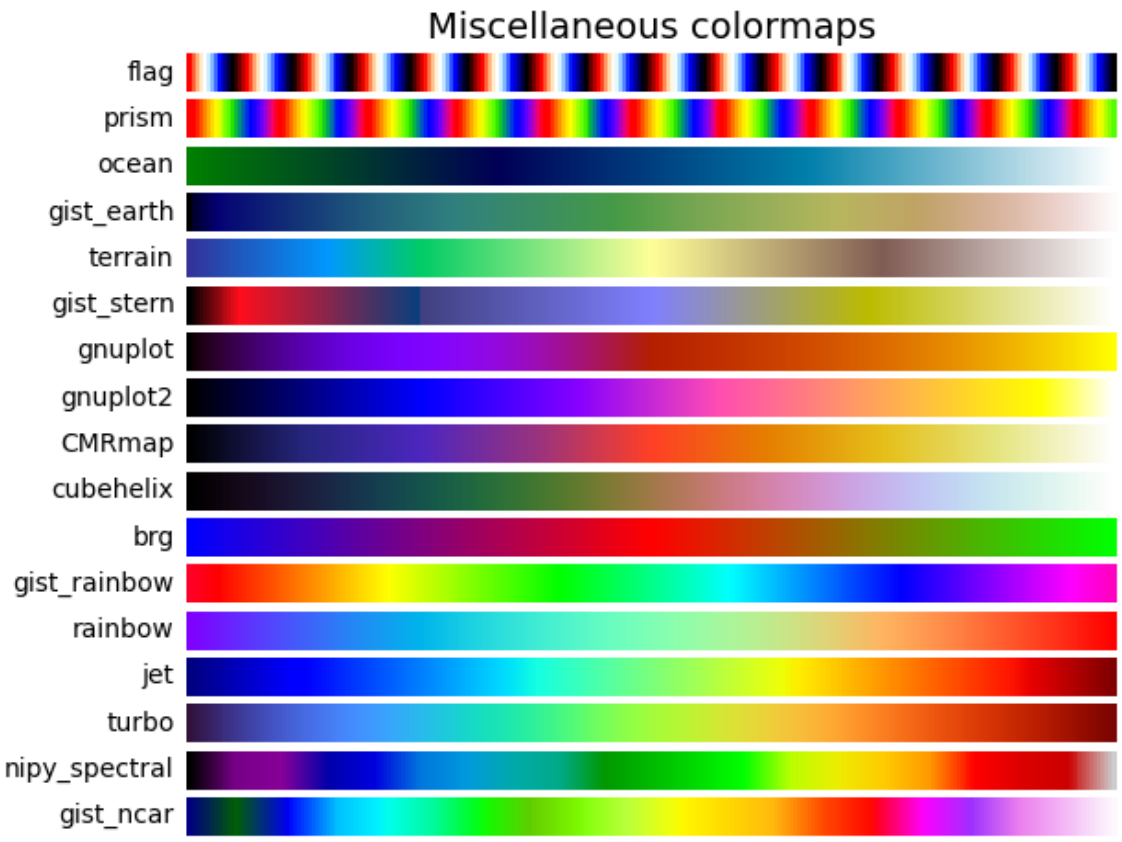
-


Cyclic colormaps
twilight
twilight_shifted
hsv

-


Qualitative colormaps
Pastel1
Pastel2
Paired
Accent
Dark2
Set1
Set2
Set3
tab10
tab20
tab20b
tab20c

-



-

Lightness of Matplotlib colormaps

Here we examine the lightness values of the matplotlib colormaps. Note that some documentation on the colormaps is available ([list-colormaps]).

```
mpl.rcParams.update({'font.size': 12})

# Number of colormap per subplot for particular cmap categories
_DSUBS = {'Perceptually Uniform Sequential': 5, 'Sequential': 6,
          'Sequential (2)': 6, 'Diverging': 6, 'Cyclic': 3,
          'Qualitative': 4, 'Miscellaneous': 6}

# Spacing between the colormaps of a subplot
_DC = {'Perceptually Uniform Sequential': 1.4, 'Sequential': 0.7,
       'Sequential (2)': 1.4, 'Diverging': 1.4, 'Cyclic': 1.4,
       'Qualitative': 1.4, 'Miscellaneous': 1.4}

# Indices to step through colormap
x = np.linspace(0.0, 1.0, 100)

# Do plot
for cmap_category, cmap_list in cmaps.items():
```

(continues on next page)

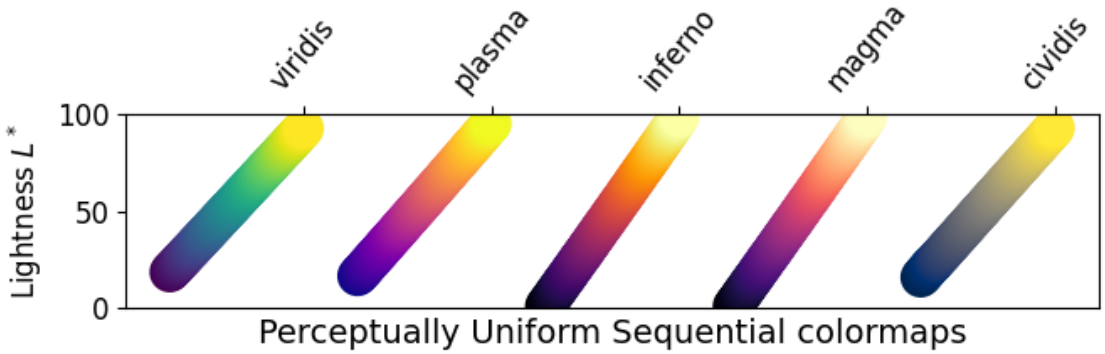
(continued from previous page)

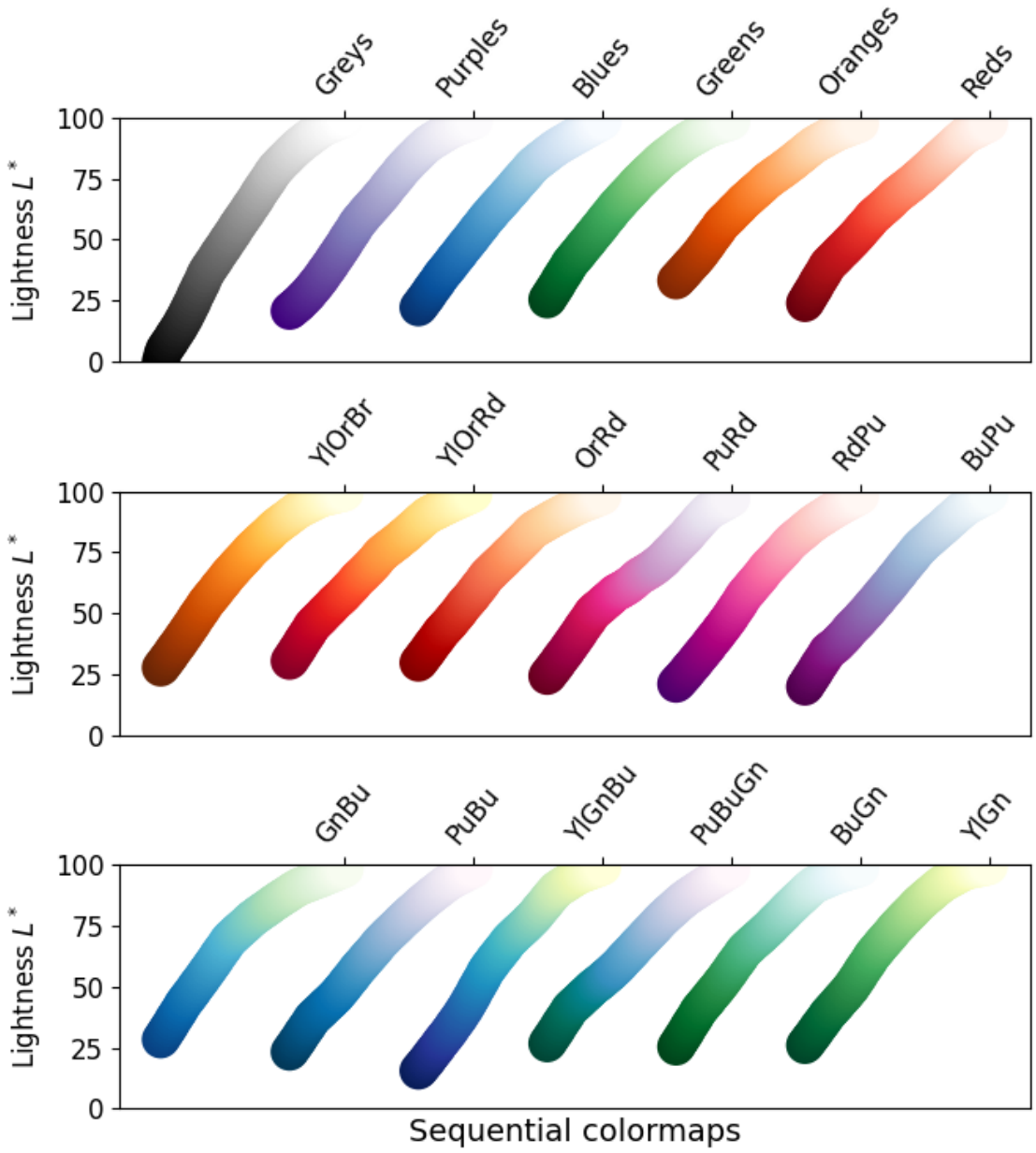
```
    # Do subplots so that colormaps have enough space.
    # Default is 6 colormaps per subplot.
    dsub = _DSUBS.get(cmap_category, 6)
    nsubplots = int(np.ceil(len(cmap_list) / dsub))

    # squeeze=False to handle similarly the case of a single subplot
    fig, axes = plt.subplots(nrows=nsubplots, squeeze=False,
                             figsize=(7, 2.6*nsubplots))

    for i, ax in enumerate(axes.flat):

        locs = []  # locations for text labels

        for j, cmap in enumerate(cmap_list[i*dsub:(i+1)*dsub]):

            # Get RGB values for colormap and convert the colormap in
            # CAM02-UCS colorspace.  lab[0, :, 0] is the lightness.
            rgb = cm.get_cmap(cmap)(x)[np.newaxis, :, :3]
            lab = cspace_converter("sRGB1", "CAM02-UCS")(rgb)

            # Plot colormap L values.  Do separately for each category
            # so each plot can be pretty.  To make scatter markers change
            # color along plot:
            # http://stackoverflow.com/questions/8202605/

            if cmap_category == 'Sequential':
                # These colormaps all start at high lightness but we want them
                # reversed to look nice in the plot, so reverse the order.
                y_ = lab[0, ::-1, 0]
                c_ = x[::-1]
            else:
                y_ = lab[0, :, 0]
                c_ = x

            dc = _DC.get(cmap_category, 1.4)  # cmaps horizontal spacing
            ax.scatter(x + j*dc, y_, c=c_, cmap=cmap, s=300, linewidths=0.0)

            # Store locations for colormap labels
            if cmap_category in ('Perceptually Uniform Sequential',
                                 'Sequential'):
                locs.append(x[-1] + j*dc)
            elif cmap_category in ('Diverging', 'Qualitative', 'Cyclic',
                                   'Miscellaneous', 'Sequential (2)'):
                locs.append(x[int(x.size/2.)] + j*dc)

        # Set up the axis limits:
        #   * the 1st subplot is used as a reference for the x-axis limits
        #   * lightness values goes from 0 to 100 (y-axis limits)
        ax.set_xlim(axes[0, 0].get_xlim())
        ax.set_ylim(0.0, 100.0)
```

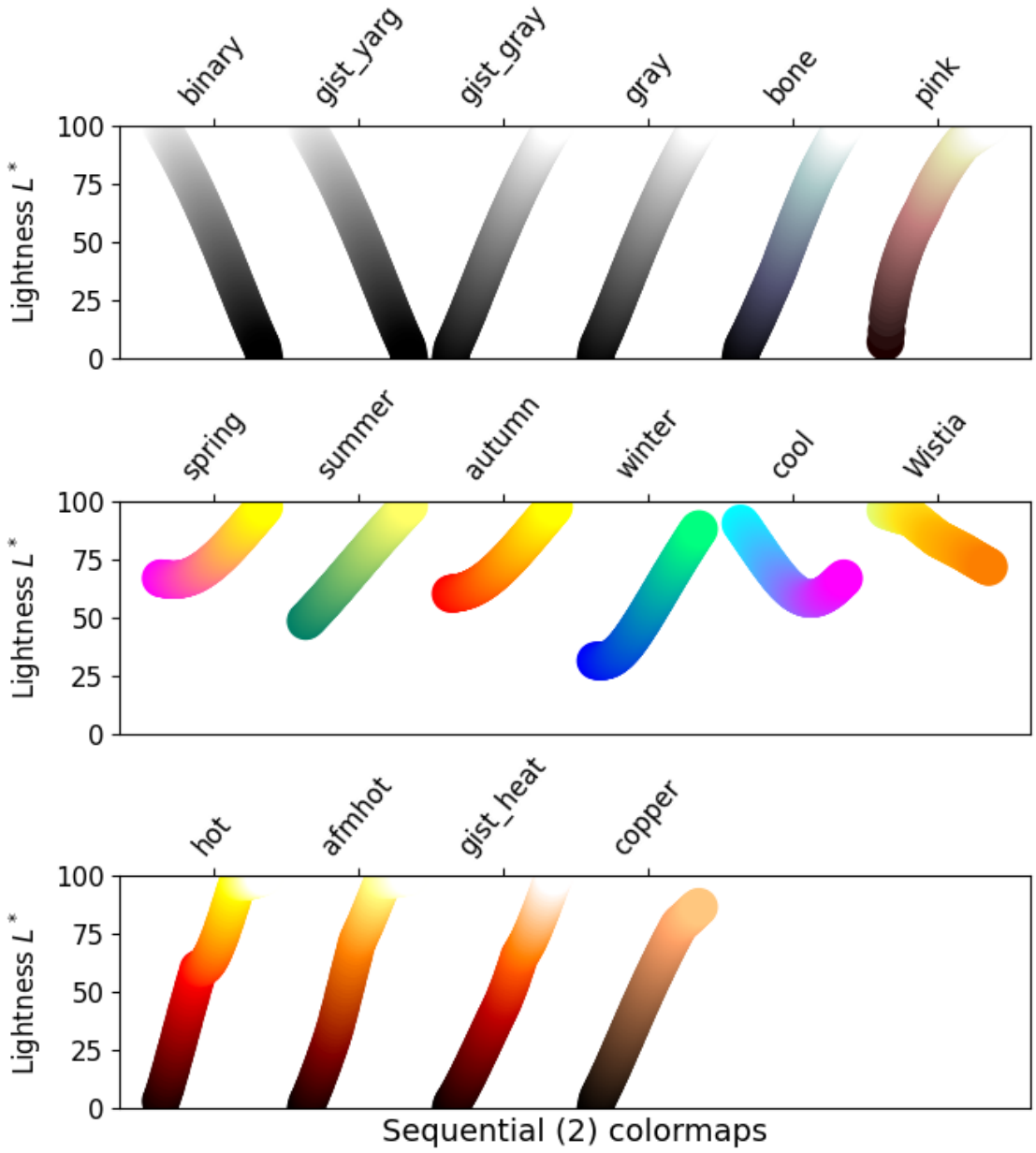
(continues on next page)

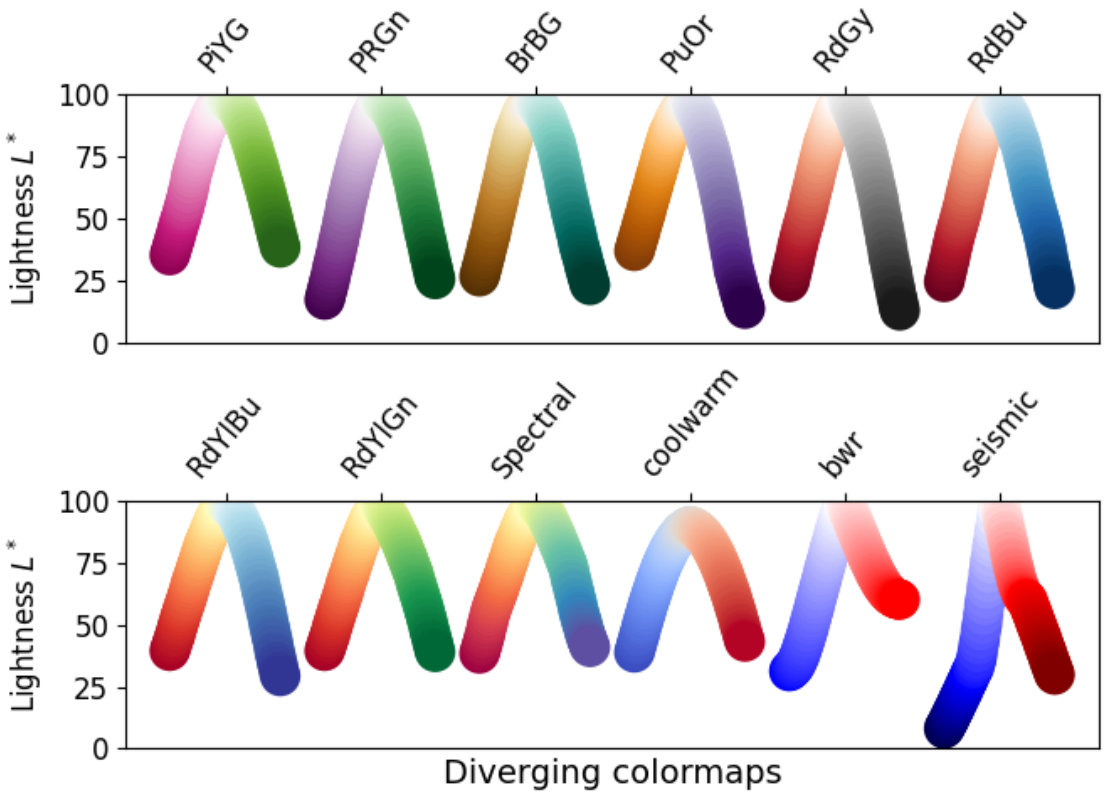
(continued from previous page)

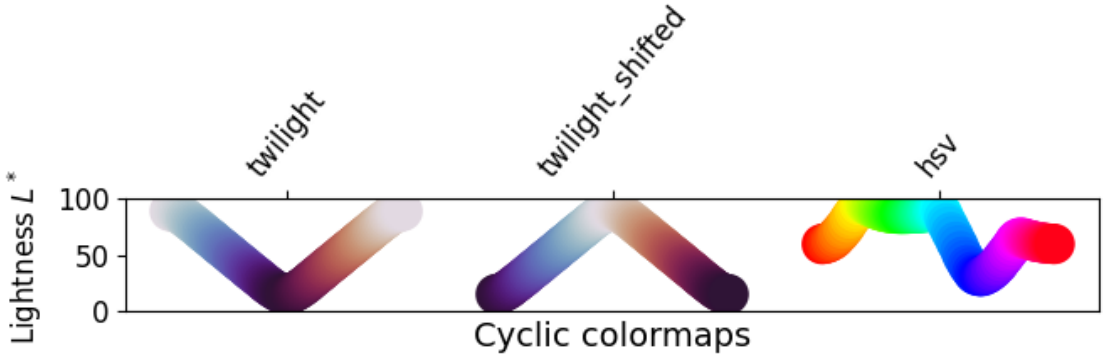
```
        # Set up labels for colormaps
        ax.xaxis.set_ticks_position('top')
        ticker = mpl.ticker.FixedLocator(locs)
        ax.xaxis.set_major_locator(ticker)
        formatter = mpl.ticker.FixedFormatter(cmap_list[i*dsub:(i+1)*dsub])
        ax.xaxis.set_major_formatter(formatter)
        ax.xaxis.set_tick_params(rotation=50)
        ax.set_ylabel('Lightness $L^*$', fontsize=12)

    ax.set_xlabel(cmap_category + ' colormaps', fontsize=14)

    fig.tight_layout(h_pad=0.0, pad=1.5)
    plt.show()
```

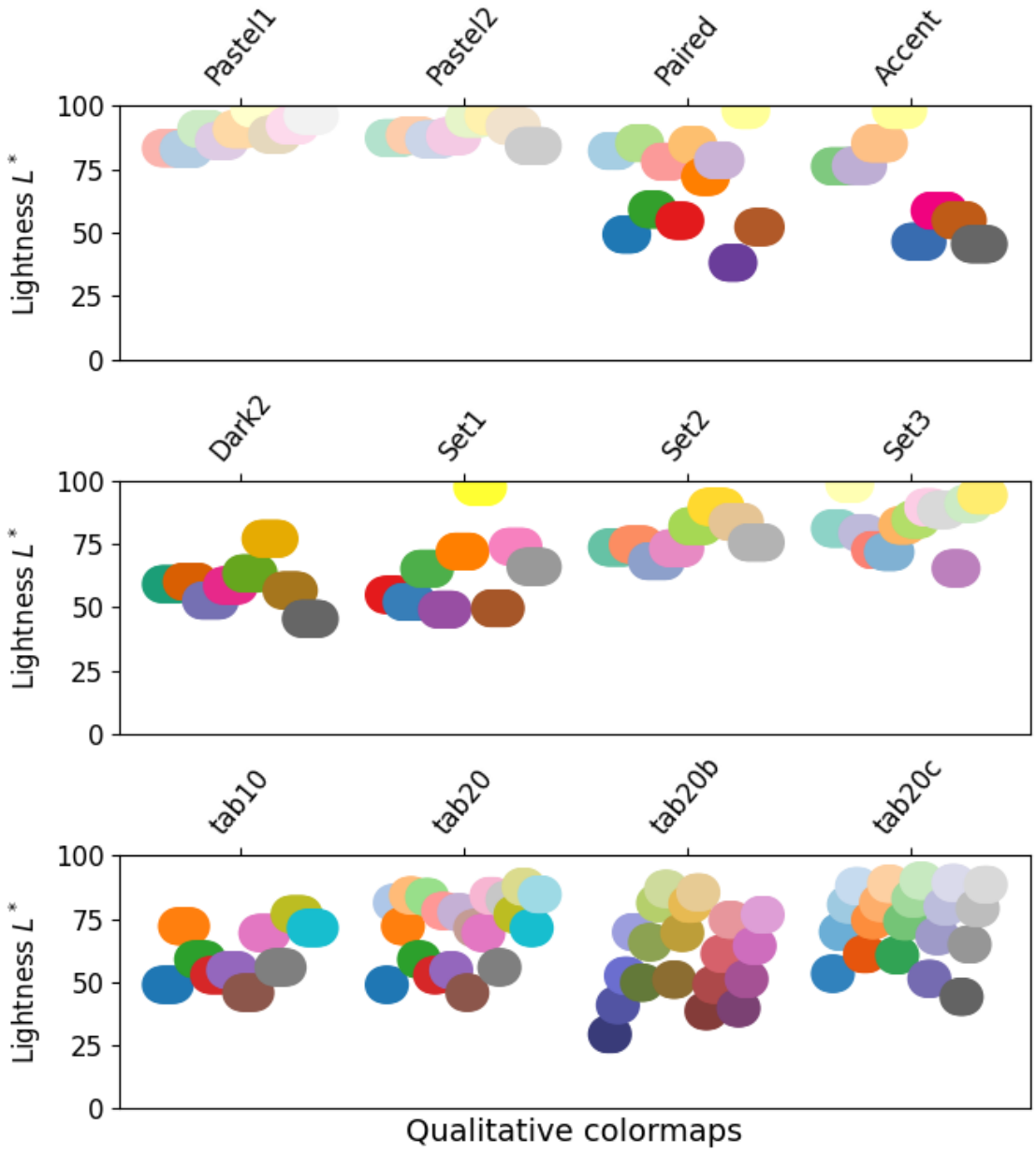


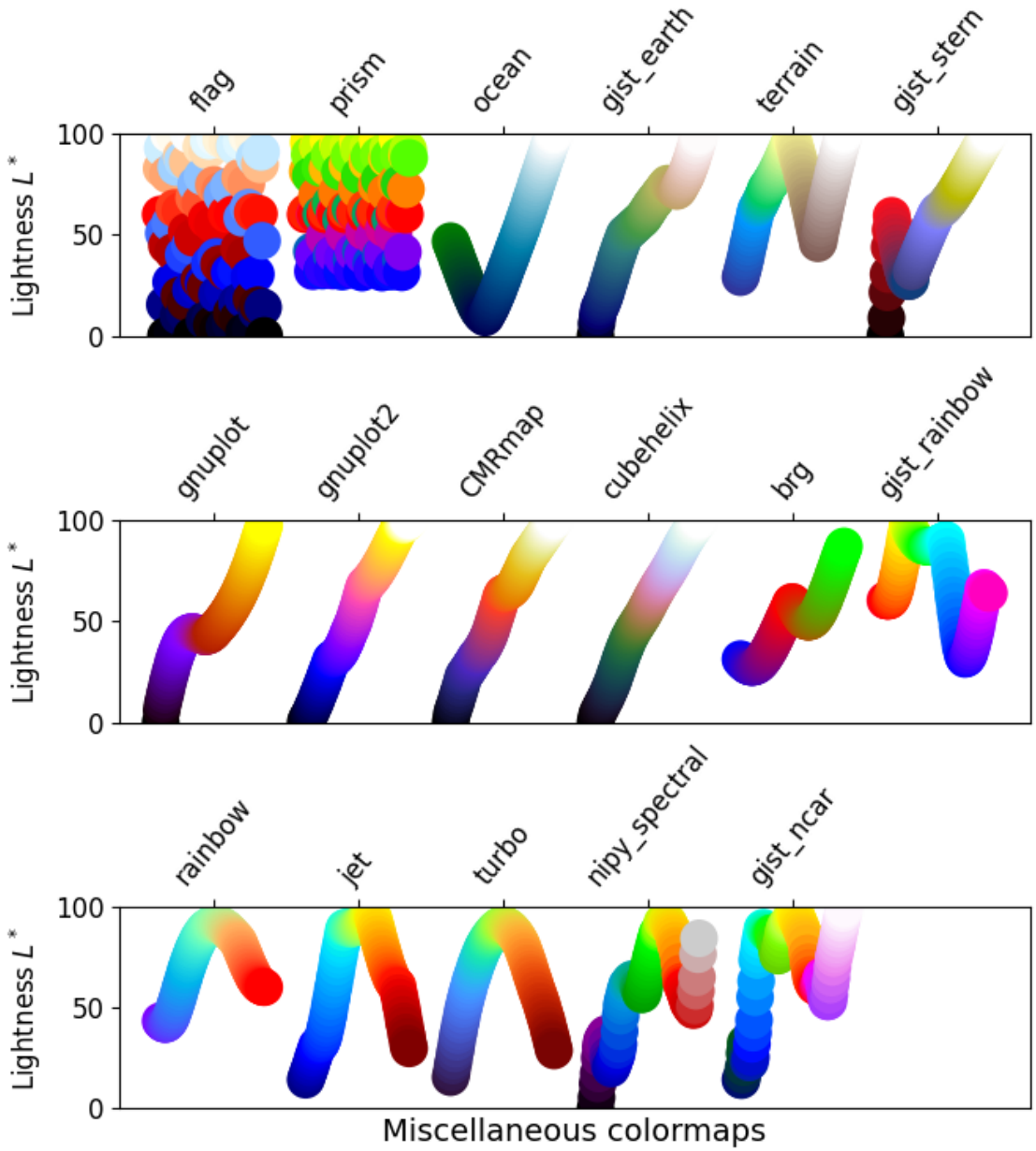

Greys
Purples
Blues
Greens
Oranges
Reds
YlOrBr
YlOrRd
OrRd
PuRd
RdPu
BuPu
GnBu
PuBu
YlGnBu
PuBuGn
BuGn
YlGn
Lightness L^*
100
75
50
25
0
Sequential colormaps


binary
gist_yarg
gist_gray
gray
bone
pink
spring
summer
autumn
winter
cool
Wistia
hot
afmhot
gist_heat
copper
Lightness L^*
100
75
50
25
0
Sequential (2) colormaps


PiYG
PRGn
BrBG
PuOr
RdGy
RdBu
RdYlBu
RdYlGn
Spectral
coolwarm
bwr
seismic
Lightness L^*
100
75
50
25
0
Diverging colormaps


twilight
twilight_shifted
hsv
Lightness L^*
100
50
0
Cyclic colormaps


Pastel1
Pastel2
Paired
Accent
Dark2
Set1
Set2
Set3
tab10
tab20
tab20b
tab20c
Lightness L^*
100
75
50
25
0
Qualitative colormaps



-

Grayscale conversion

It is important to pay attention to conversion to grayscale for color plots, since they may be printed on black and white printers. If not carefully considered, your readers may end up with indecipherable plots because the grayscale changes unpredictably through the colormap.

Conversion to grayscale is done in many different ways [bw]. Some of the better ones use a linear combination of the rgb values of a pixel, but weighted according to how we perceive color intensity. A nonlinear method of conversion to grayscale is to use

the L^* values of the pixels. In general, similar principles apply for this question as they do for presenting one's information perceptually; that is, if a colormap is chosen that is monotonically increasing in L^* values, it will print in a reasonable manner to grayscale.

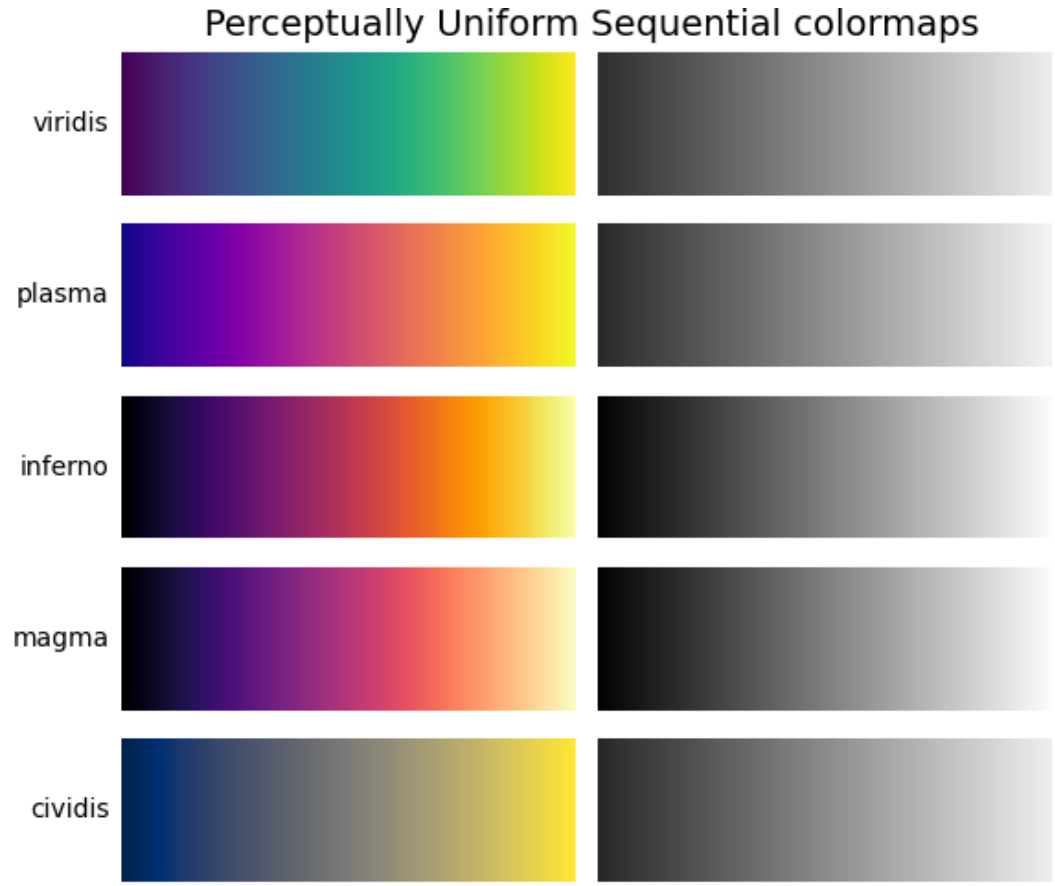
With this in mind, we see that the Sequential colormaps have reasonable representations in grayscale. Some of the Sequential2 colormaps have decent enough grayscale representations, though some (autumn, spring, summer, winter) have very little grayscale change. If a colormap like this was used in a plot and then the plot was printed to grayscale, a lot of the information may map to the same gray values. The Diverging colormaps mostly vary from darker gray on the outer edges to white in the middle. Some (PuOr and seismic) have noticeably darker gray on one side than the other and therefore are not very symmetric. coolwarm has little range of gray scale and would print to a more uniform plot, losing a lot of detail. Note that overlaid, labeled contours could help differentiate between one side of the colormap vs. the other since color cannot be used once a plot is printed to grayscale. Many of the Qualitative and Miscellaneous colormaps, such as Accent, hsv, jet and turbo, change from darker to lighter and back to darker grey throughout the colormap. This would make it impossible for a viewer to interpret the information in a plot once it is printed in grayscale.

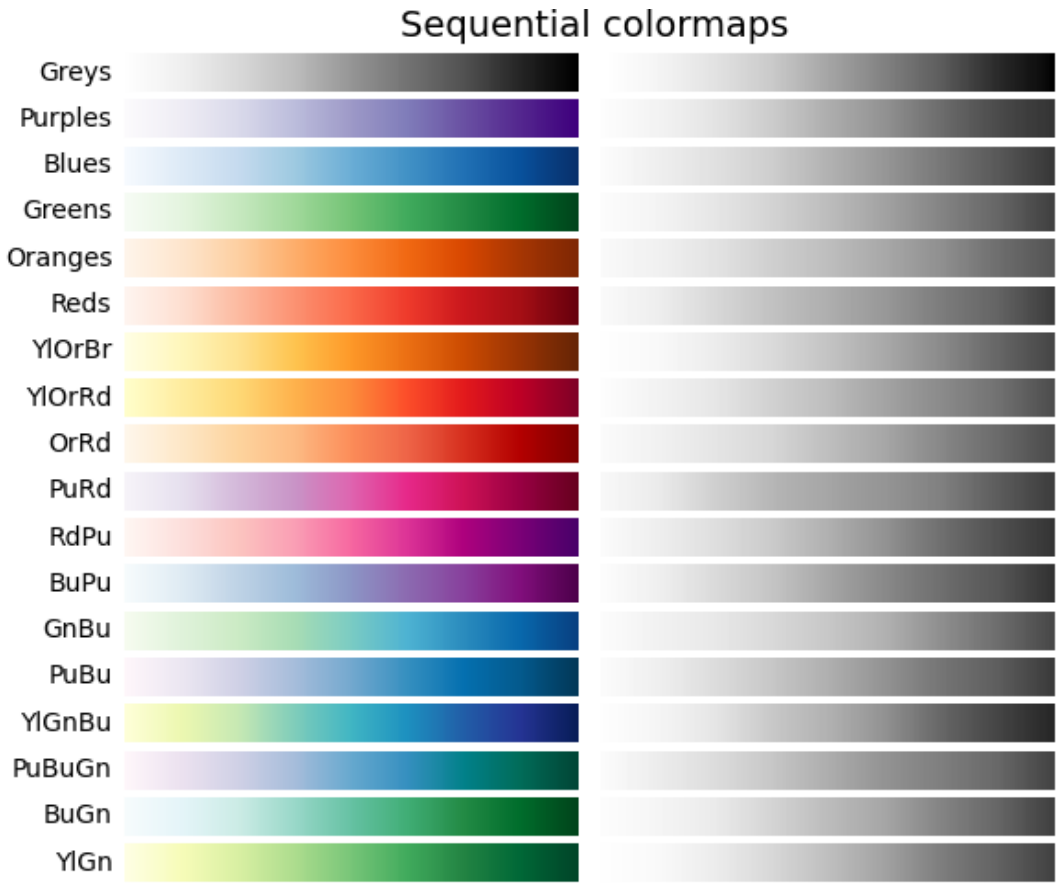
```
mpl.rcParams.update({'font.size': 14})

# Indices to step through colormap.
x = np.linspace(0.0, 1.0, 100)

gradient = np.linspace(0, 1, 256)
gradient = np.vstack((gradient, gradient))


def plot_color_gradients(cmap_category, cmap_list):
    fig, axes = plt.subplots(nrows=len(cmap_list), ncols=2)
    fig.subplots_adjust(top=0.95, bottom=0.01, left=0.2, right=0.99,
                        wspace=0.05)
    fig.suptitle(cmap_category + ' colormaps', fontsize=14, y=1.0, x=0.6)

    for ax, name in zip(axes, cmap_list):

        # Get RGB values for colormap.
        rgb = cm.get_cmap(plt.get_cmap(name))(x)[np.newaxis, :, :3]

        # Get colormap in CAM02-UCS colorspace. We want the lightness.
        lab = cspace_converter("sRGB1", "CAM02-UCS")(rgb)
        L = lab[0, :, 0]
        L = np.float32(np.vstack((L, L, L)))

        ax[0].imshow(gradient, aspect='auto', cmap=plt.get_cmap(name))
        ax[1].imshow(L, aspect='auto', cmap='binary_r', vmin=0., vmax=100.)
        pos = list(ax[0].get_position().bounds)
        x_text = pos[0] - 0.01
        y_text = pos[1] + pos[3]/2.
```

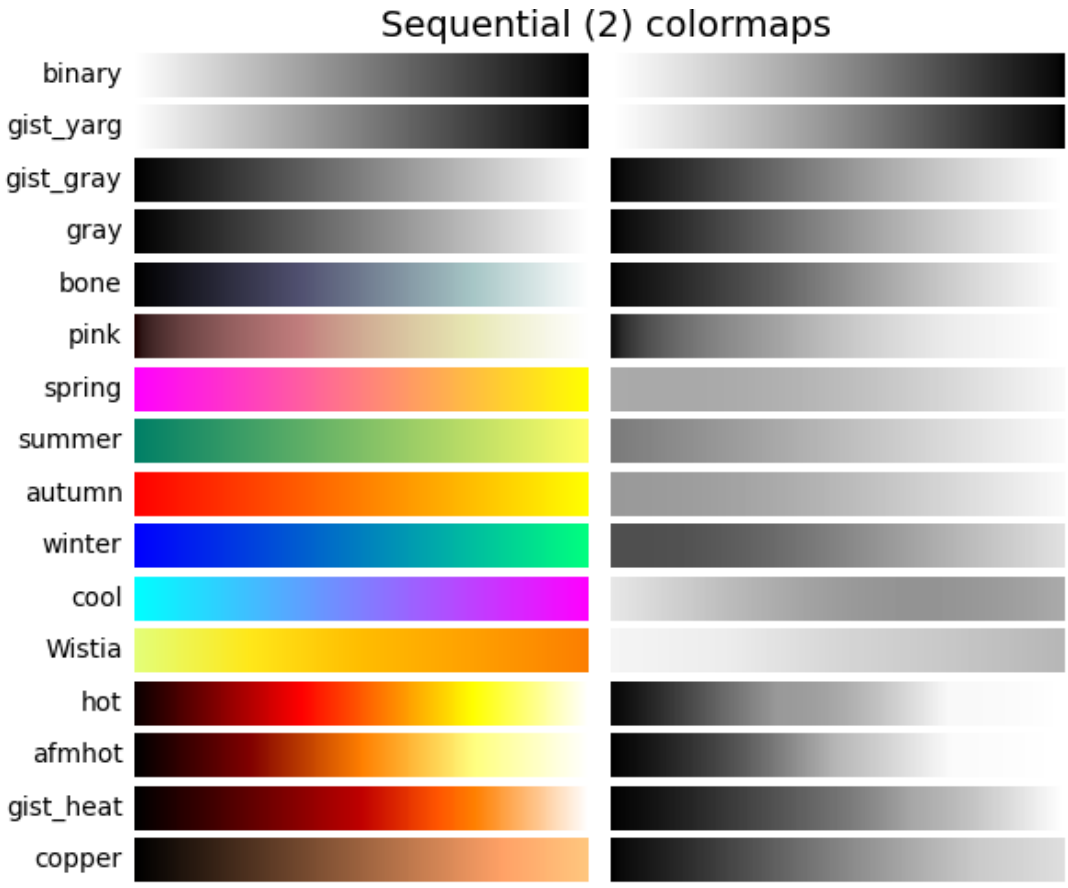
(continues on next page)

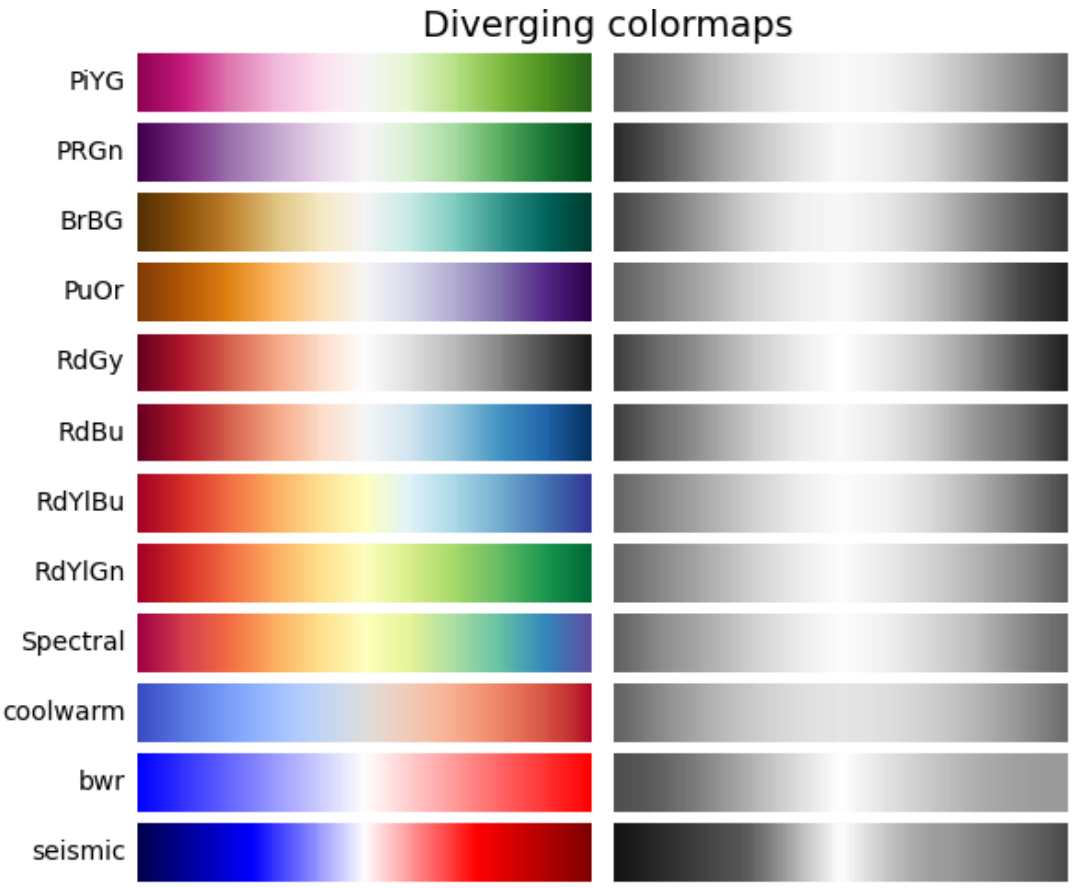
(continued from previous page)

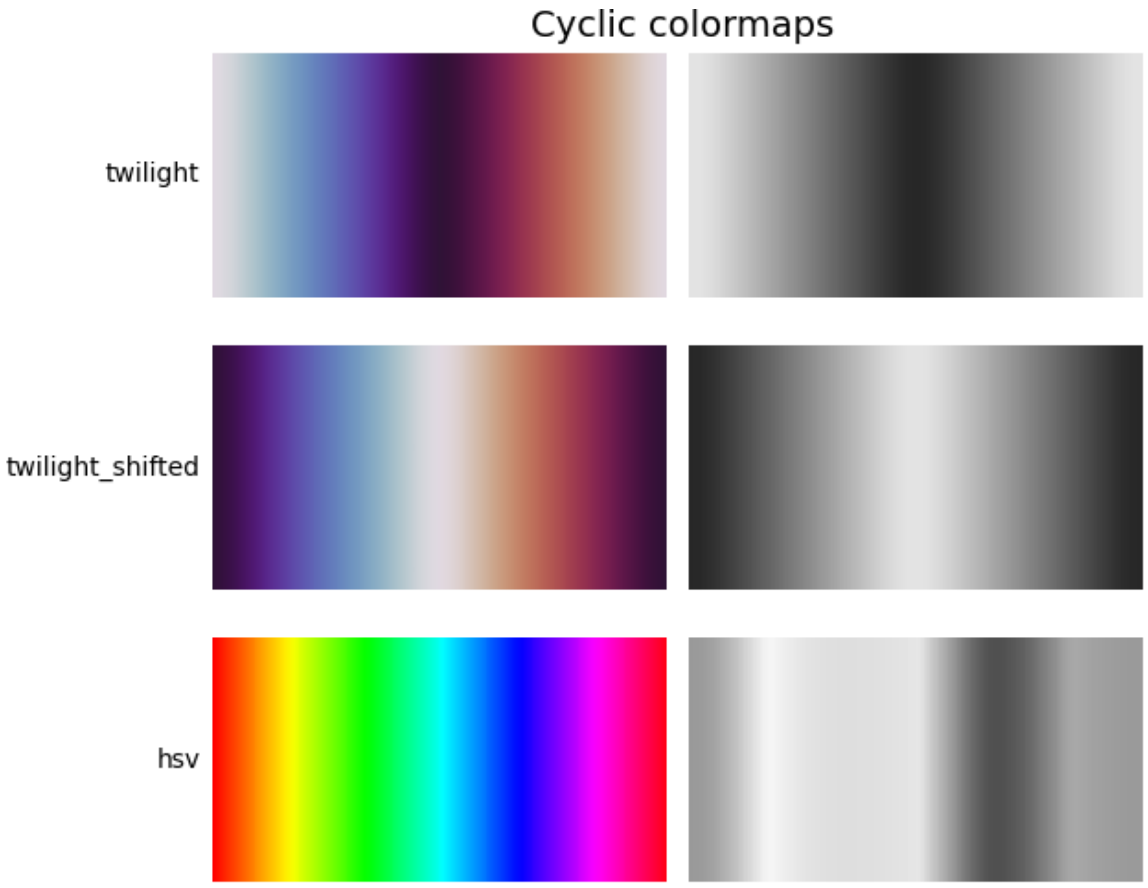
```
        fig.text(x_text, y_text, name, va='center', ha='right', fontsize=10)

    # Turn off *all* ticks & spines, not just the ones with colormaps.
    for ax in axes.flat:
        ax.set_axis_off()

    plt.show()


for cmap_category, cmap_list in cmaps.items():

    plot_color_gradients(cmap_category, cmap_list)
```

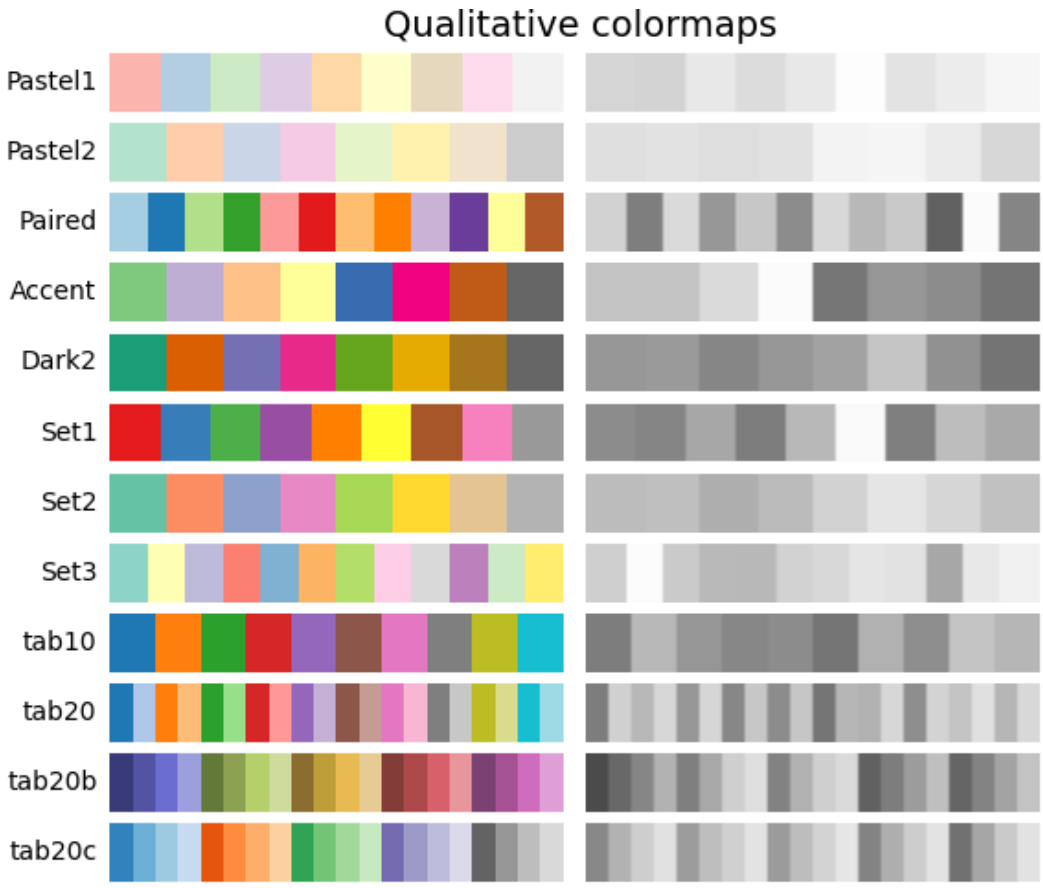


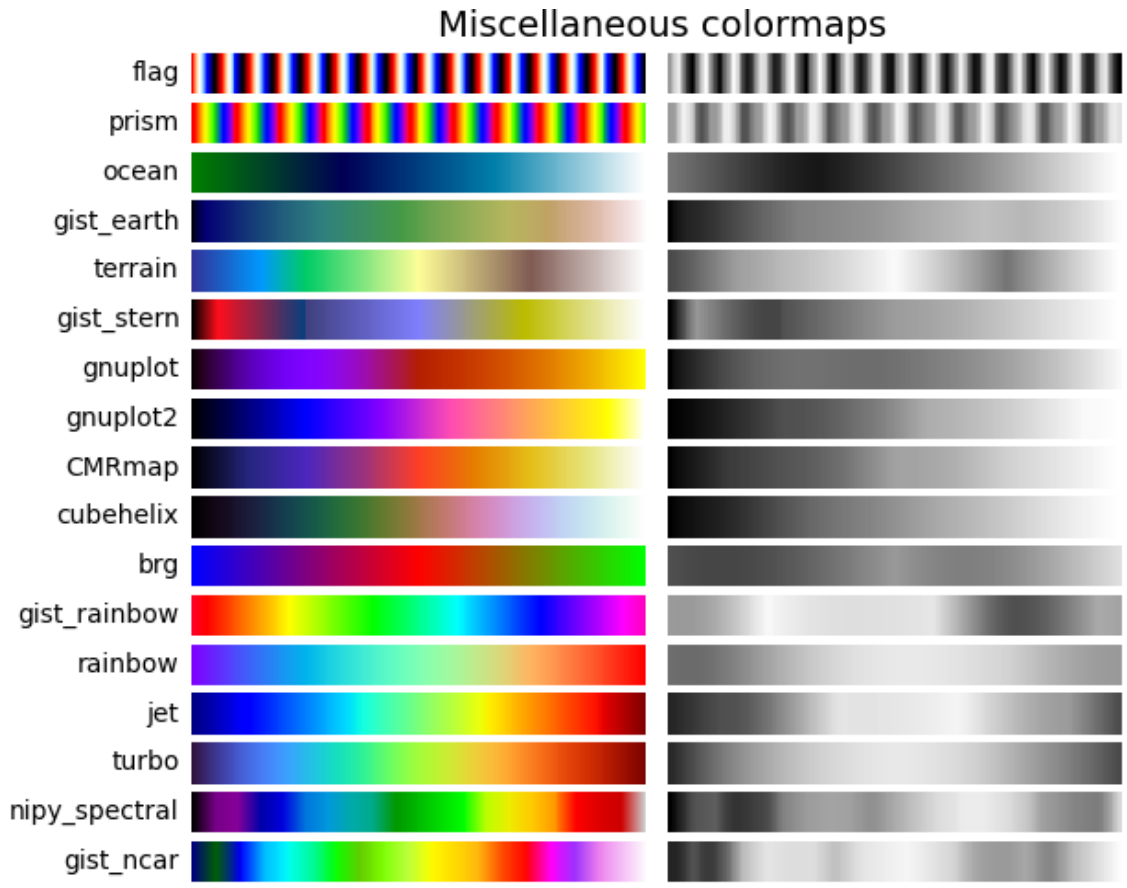

Sequential colormaps
Greys
Purples
Blues
Greens
Oranges
Reds
YlOrBr
YlOrRd
OrRd
PuRd
RdPu
BuPu
GnBu
PuBu
YlGnBu
PuBuGn
BuGn
YlGn


Sequential (2) colormaps
binary
gist_yarg
gist_gray
gray
bone
pink
spring
summer
autumn
winter
cool
Wistia
hot
afmhot
gist_heat
copper


Diverging colormaps
PiYG
PRGn
BrBG
PuOr
RdGy
RdBu
RdYlBu
RdYlGn
Spectral
coolwarm
bwr
seismic


Cyclic colormaps
twilight
twilight_shifted
hsv


Qualitative colormaps
Pastel1
Pastel2
Paired
Accent
Dark2
Set1
Set2
Set3
tab10
tab20
tab20b
tab20c



Color vision deficiencies

There is a lot of information available about color blindness (*e.g.*, [colorblindness]). Additionally, there are tools available to convert images to how they look for different types of color vision deficiencies.

The most common form of color vision deficiency involves differentiating between red and green. Thus, avoiding colormaps with both red and green will avoid many problems in general.

References

Total running time of the script: (0 minutes 13.396 seconds)

2.5 Provisional

These tutorials cover proposed APIs of any complexity. These are here to document features that we have released, but want to get user feedback on before committing to them. Please have a look, try them out and give us feedback on gitter, discourse, or the the mailing list! But, be aware that we may change the APIs without warning in subsequent versions.

2.5.1 Complex and semantic figure composition

Warning: This tutorial documents experimental / provisional API. We are releasing this in v3.3 to get user feedback. We may make breaking changes in future versions with no warning.

Laying out Axes in a Figure in a non uniform grid can be both tedious and verbose. For dense, even grids we have *Figure.subplots* but for more complex layouts, such as Axes that span multiple columns / rows of the layout or leave some areas of the Figure blank, you can use *gridspec.GridSpec* (see *Customizing Figure Layouts Using GridSpec and Other Functions*) or manually place your axes. *Figure.subplot_mosaic* aims to provide an interface to visually lay out your axes (as either ASCII art or nested lists) to streamline this process.

This interface naturally supports naming your axes. *Figure.subplot_mosaic* returns a dictionary keyed on the labels used to lay out the Figure. By returning data structures with names, it is easier to write plotting code that is independent of the Figure layout.

This is inspired by a proposed MEP and the patchwork library for R. While we do not implement the operator overloading style, we do provide a Pythonic API for specifying (nested) Axes layouts.

```
import matplotlib.pyplot as plt
import numpy as np


# Helper function used for visualization in the following examples
def identify_axes(ax_dict, fontsize=48):
    """
    Helper to identify the Axes in the examples below.

    Draws the label in a large font in the center of the Axes.
```

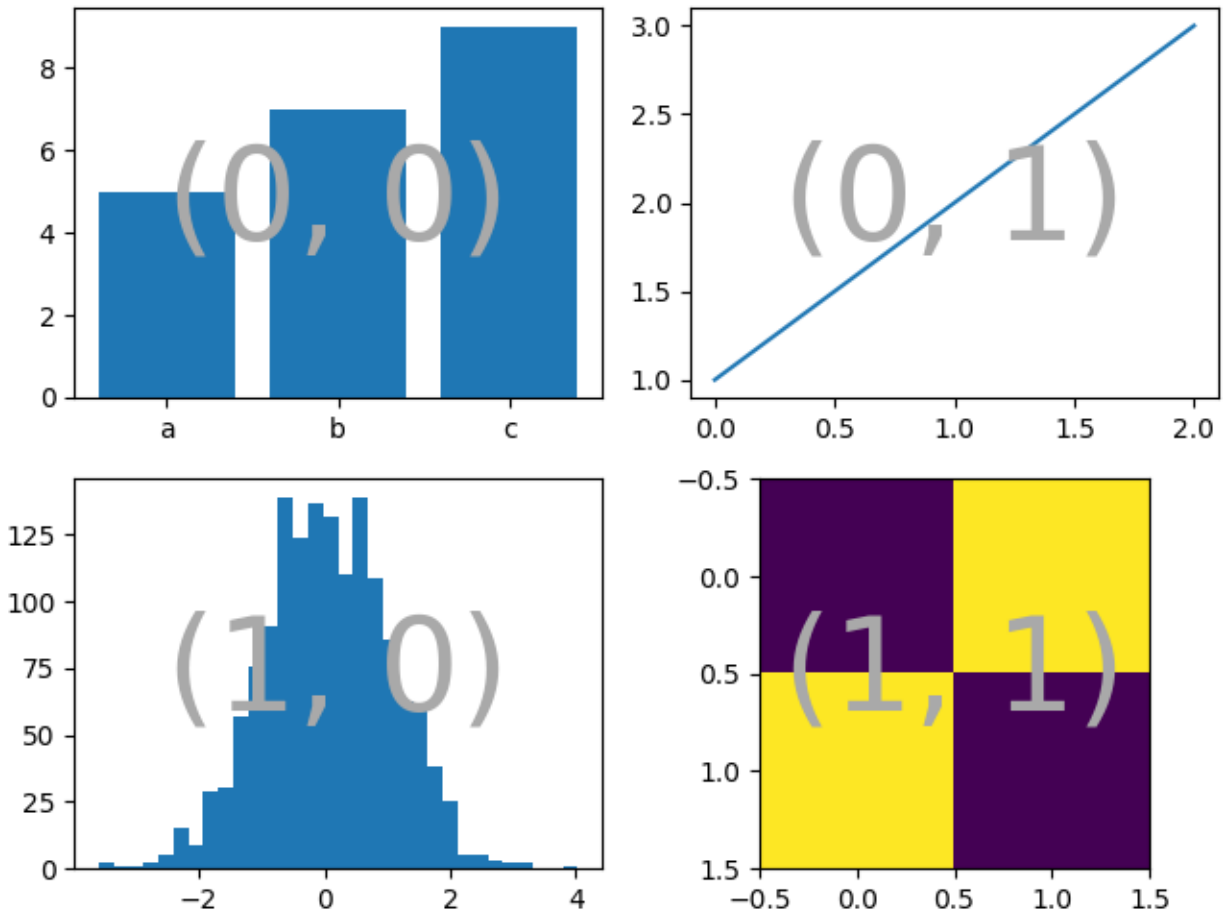
(continues on next page)

(continued from previous page)

```
    Parameters
    ----------
    ax_dict : Dict[str, Axes]
        Mapping between the title / label and the Axes.

    fontsize : int, optional
        How big the label should be
    """
    kw = dict(ha="center", va="center", fontsize=fontsize, color="darkgrey")
    for k, ax in ax_dict.items():
        ax.text(0.5, 0.5, k, transform=ax.transAxes, **kw)
```

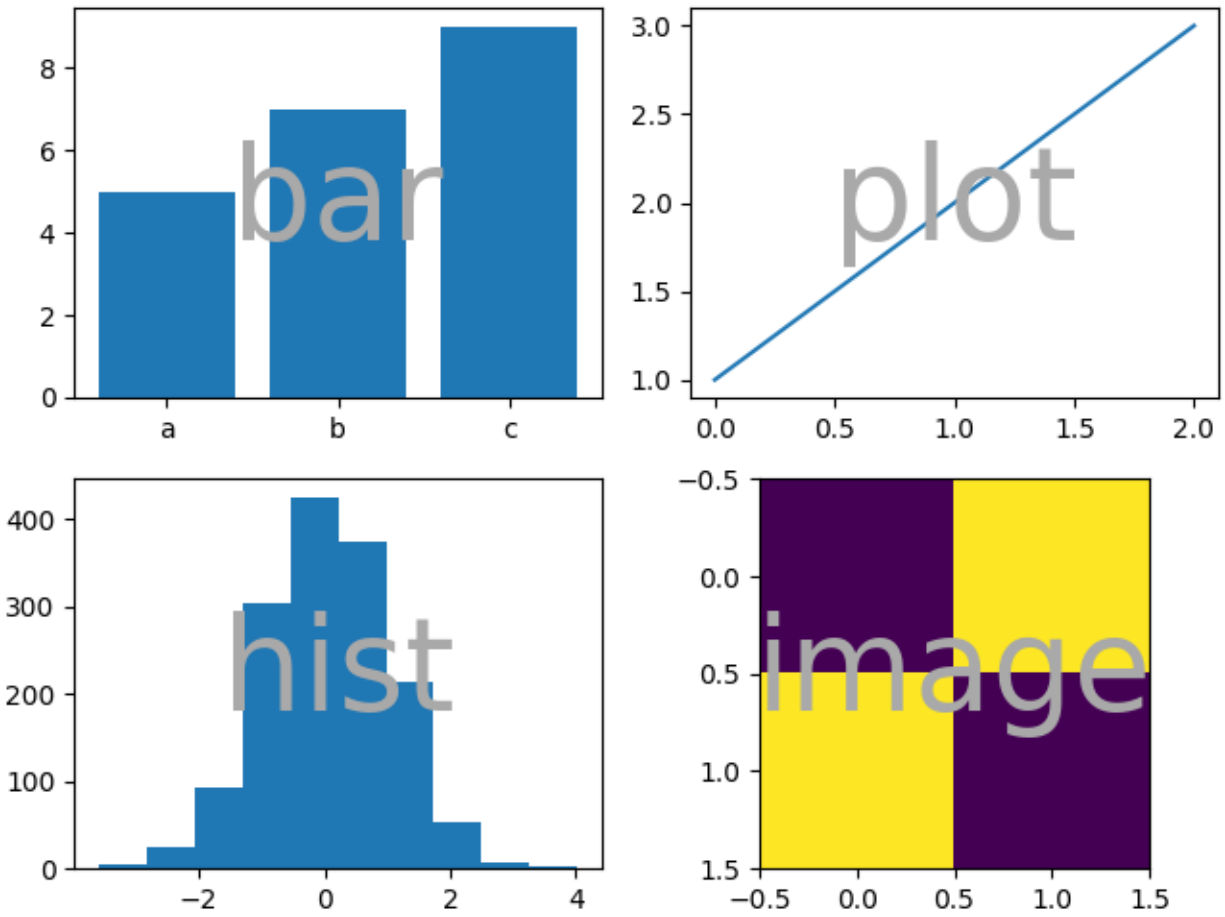
If we want a 2x2 grid we can use *Figure.subplots* which returns a 2D array of *axes.Axes* which we can index into to do our plotting.

```
np.random.seed(19680801)
hist_data = np.random.randn(1_500)


fig = plt.figure(constrained_layout=True)
ax_array = fig.subplots(2, 2, squeeze=False)

ax_array[0, 0].bar(['a', 'b', 'c'], [5, 7, 9])
ax_array[0, 1].plot([1, 2, 3])
ax_array[1, 0].hist(hist_data, bins='auto')
ax_array[1, 1].imshow([[1, 2], [2, 1]])

identify_axes(
    {(j, k): a for j, r in enumerate(ax_array) for k, a in enumerate(r)}
)
```



Using *Figure.subplot_mosaic* we can produce the same layout but give the axes semantic names

```
fig = plt.figure(constrained_layout=True)
ax_dict = fig.subplot_mosaic(
    [['bar',  'plot'],
     ['hist', 'image']])
ax_dict['bar'].bar(['a', 'b', 'c'], [5, 7, 9])
ax_dict['plot'].plot([1, 2, 3])
ax_dict['hist'].hist(hist_data)
ax_dict['image'].imshow([[1, 2], [2, 1]])
identify_axes(ax_dict)
```



A key difference between *Figure.subplots* and *Figure.subplot_mosaic* is the return value. While the former returns an array for index access, the latter returns a dictionary mapping the labels to the *axes.Axes* instances created

```
print(ax_dict)
```

Out:

```
{'hist': <AxesSubplot:label='hist'>, 'plot': <AxesSubplot:label='plot'>, 'image':
↪<AxesSubplot:label='image'>, 'bar': <AxesSubplot:label='bar'>}
```

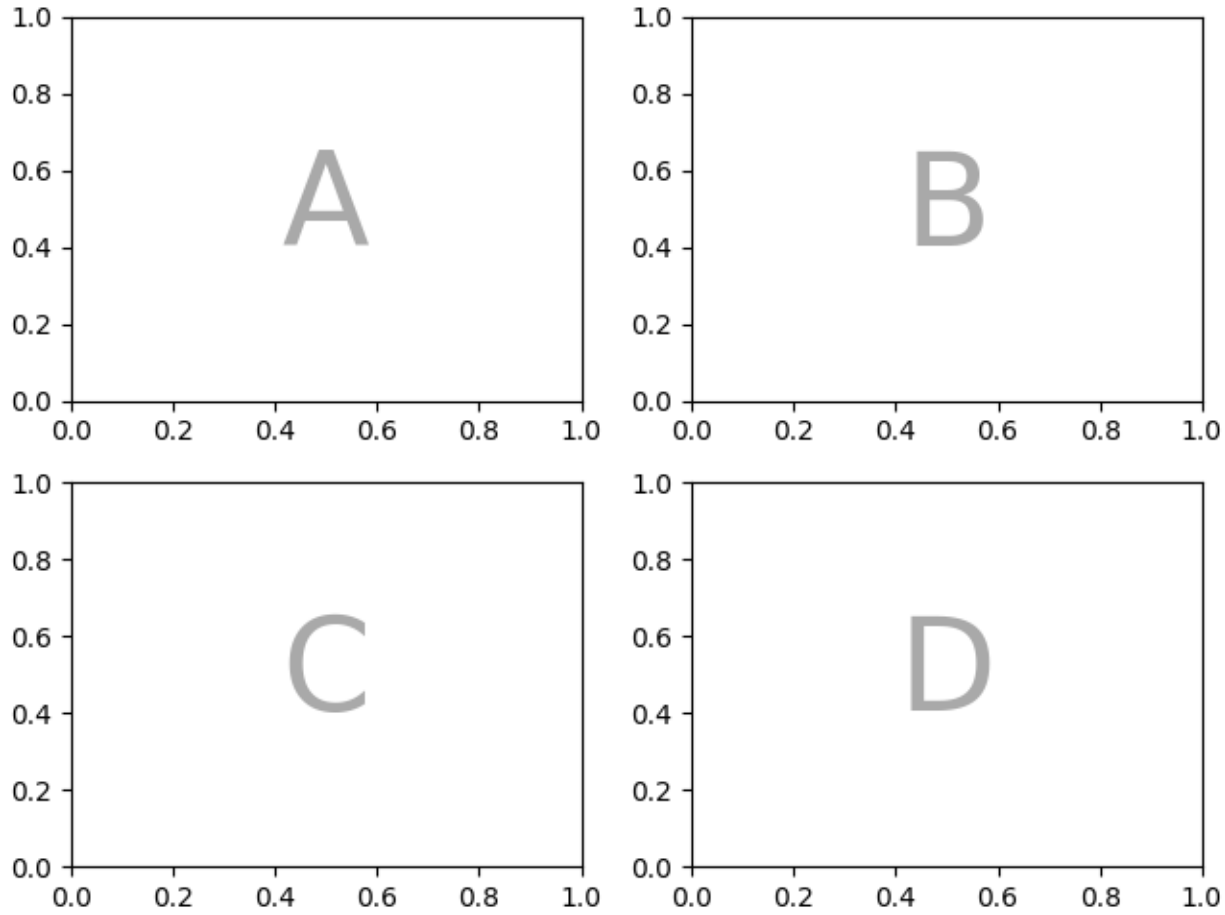
String short-hand

By restricting our axes labels to single characters we can use Using we can "draw" the Axes we want as "ASCII art". The following

```
layout = """
    AB
    CD
    """
```

will give us 4 Axes laid out in a 2x2 grid and generates the same figure layout as above (but now labeled with `{"A", "B", "C", "D"}` rather than `{"bar", "plot", "hist", "image"}`).

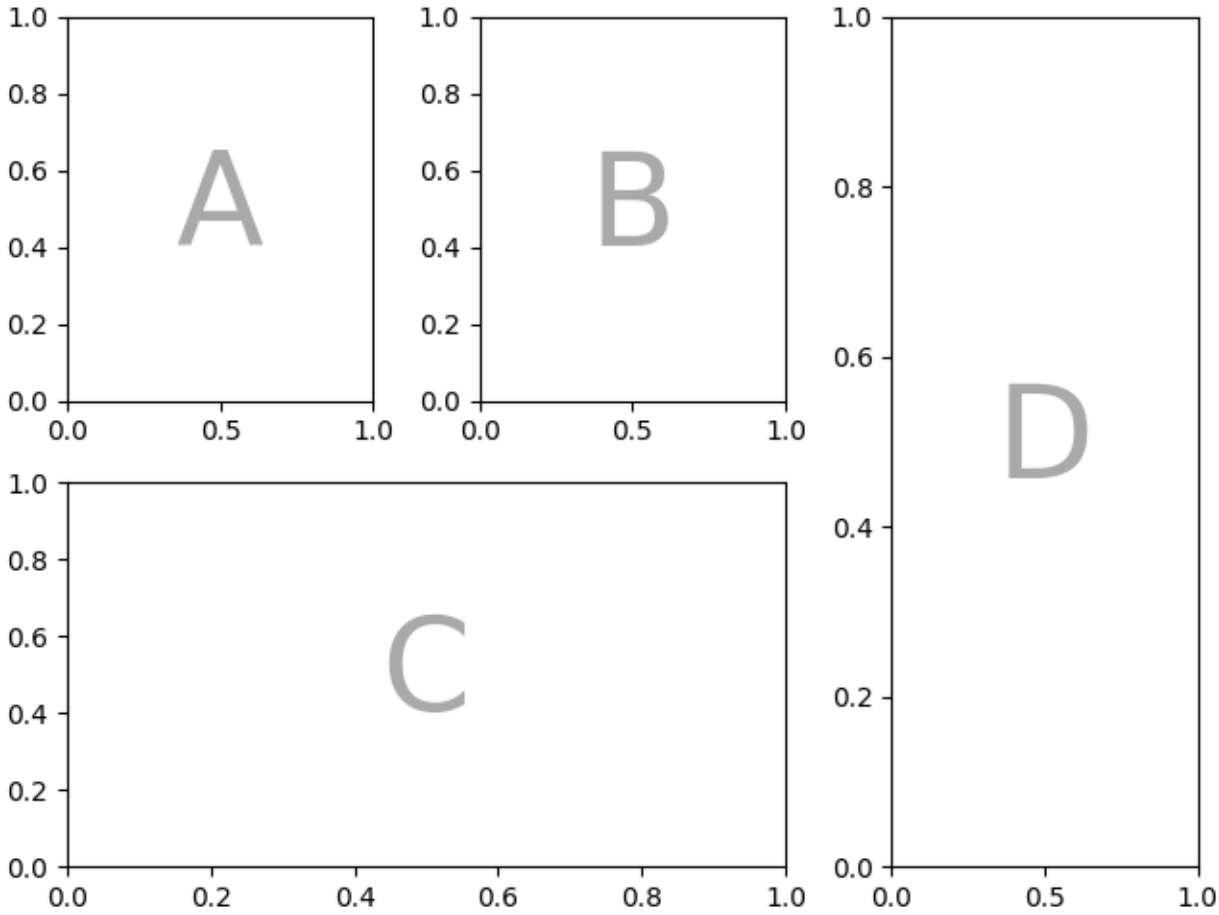
```
fig = plt.figure(constrained_layout=True)
ax_dict = fig.subplot_mosaic(layout)
identify_axes(ax_dict)
```



Something we can do with *Figure.subplot_mosaic* that you can not do with *Figure.subplots* is specify that an Axes should span several rows or columns.

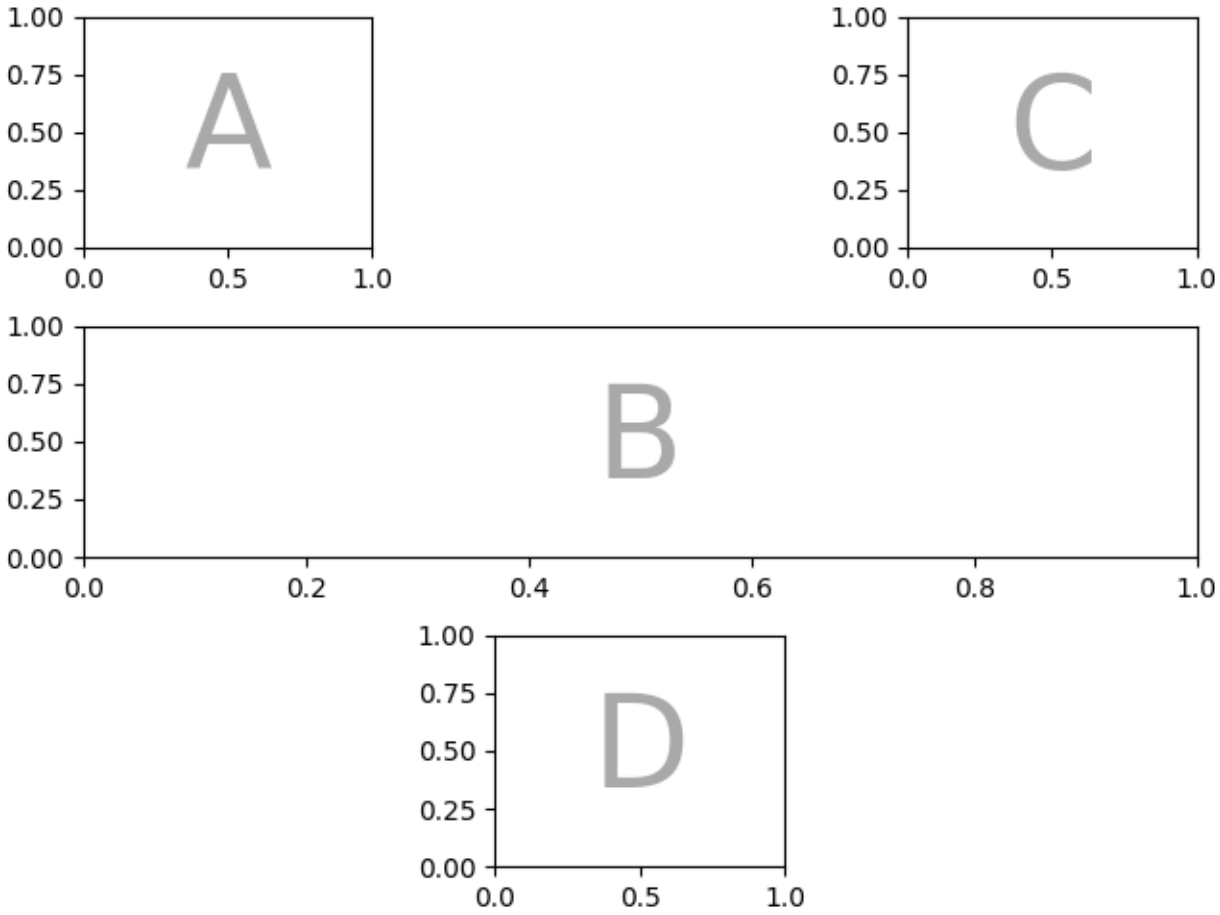
If we want to re-arrange our four Axes to have C be a horizontal span on the bottom and D be a vertical span on the right we would do

```
axd = plt.figure(constrained_layout=True).subplot_mosaic(
    """
    ABD
    CCD
    """
)
identify_axes(axd)
```



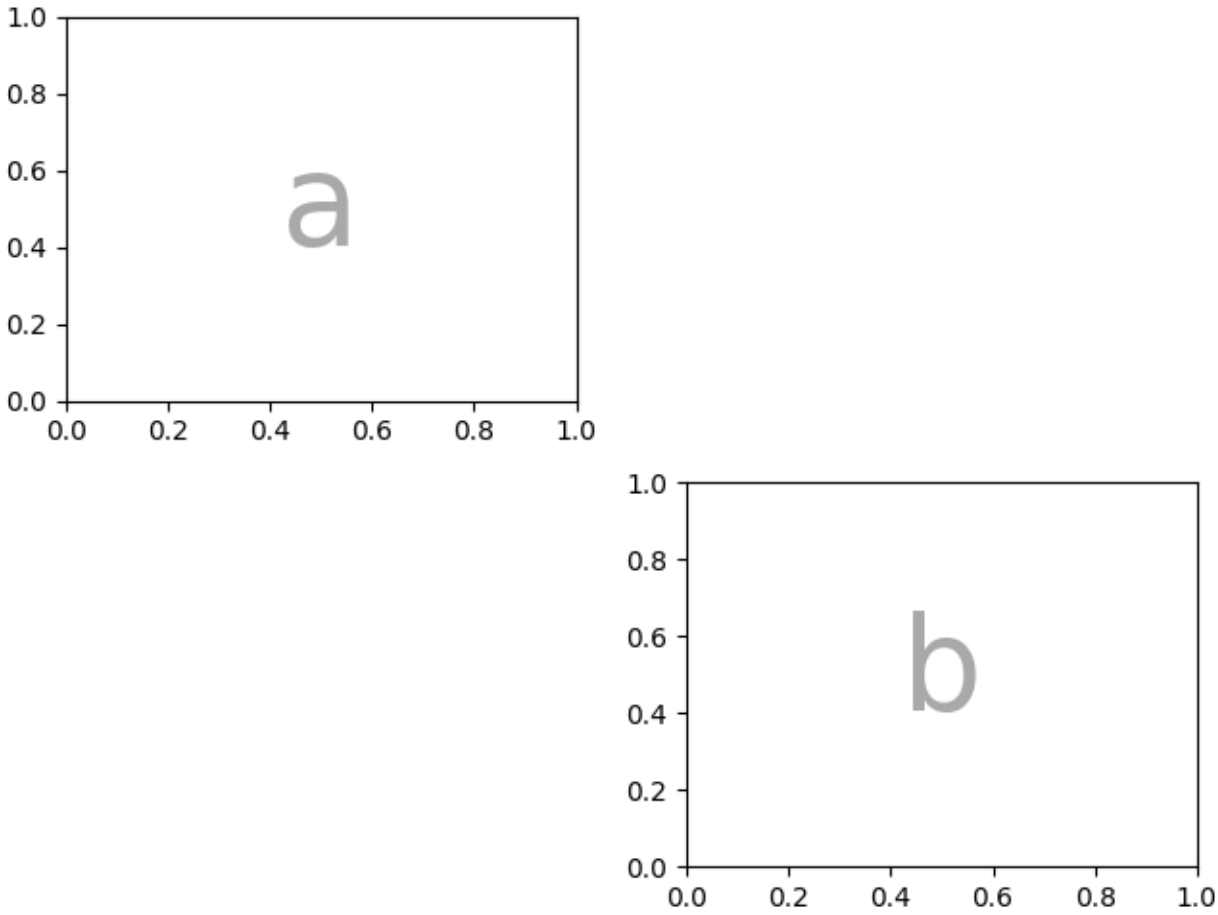
If we do not want to fill in all the spaces in the Figure with Axes, we can specify some spaces in the grid to be blank

```
axd = plt.figure(constrained_layout=True).subplot_mosaic(
    """
    A.C
    BBB
    .D.
    """
)
identify_axes(axd)
```



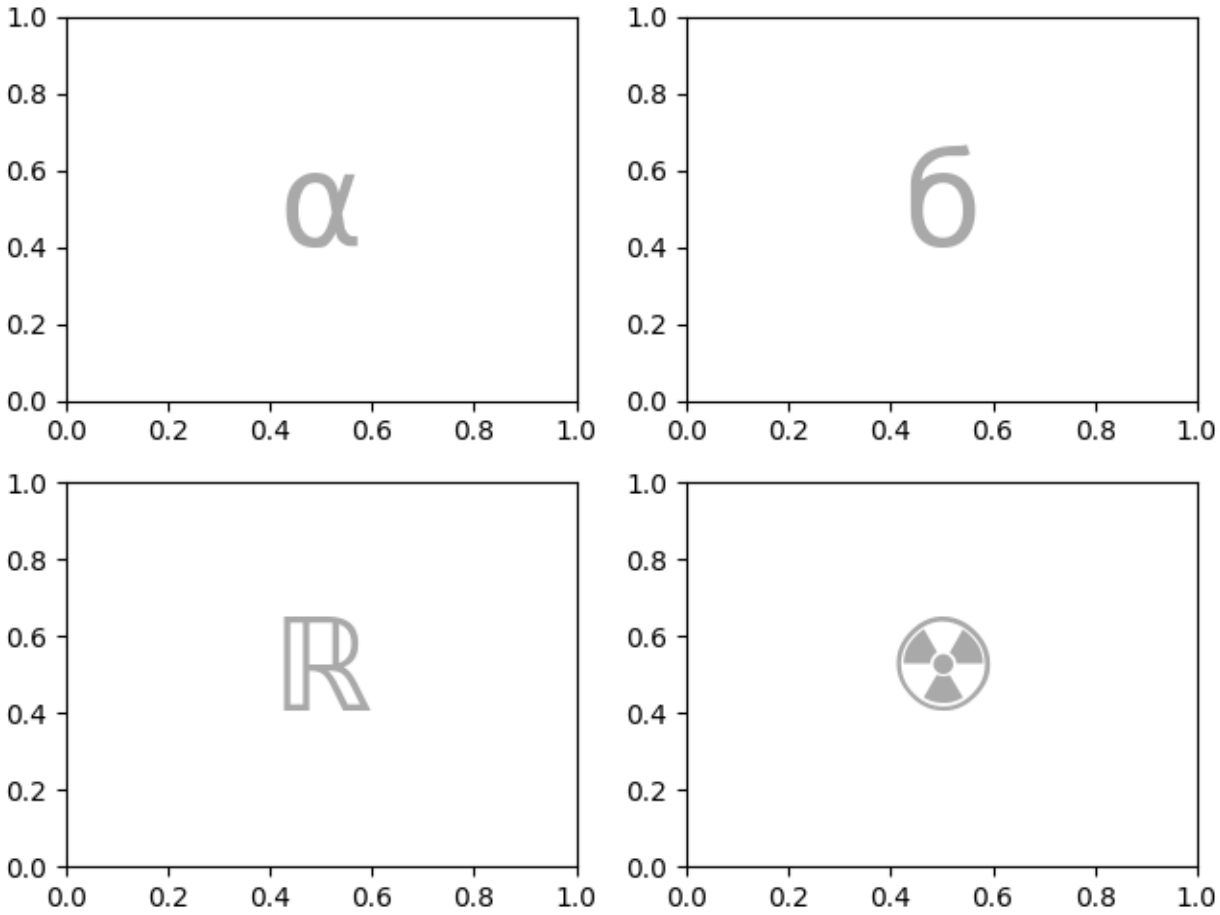
If we prefer to use another character (rather than a period `"."`) to mark the empty space, we can use *empty_sentinel* to specify the character to use.

```
axd = plt.figure(constrained_layout=True).subplot_mosaic(
    """
    aX
    Xb
    """,
    empty_sentinel="X",
)
identify_axes(axd)
```



Internally there is no meaning attached to the letters we use, any Unicode code point is valid!

```
axd = plt.figure(constrained_layout=True).subplot_mosaic(
    """
        """
)
identify_axes(axd)
```



It is not recommended to use white space as either a label or an empty sentinel with the string shorthand because it may be stripped while processing the input.

Controlling layout and subplot creation

This feature is built on top of *gridspec* and you can pass the keyword arguments through to the underlying *gridspec.GridSpec* (the same as *Figure.subplots*).

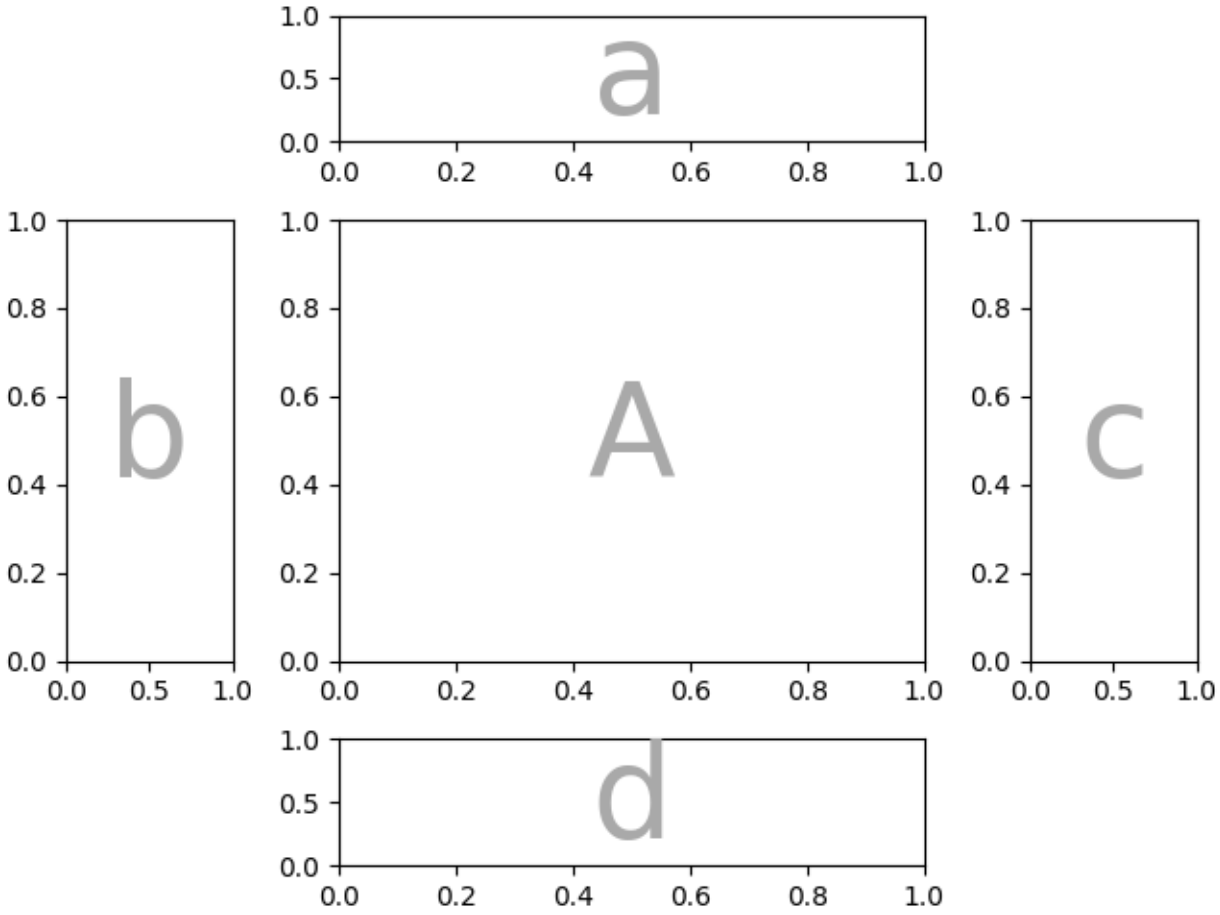
In this case we want to use the input to specify the arrangement, but set the relative widths of the rows / columns via *gridspec_kw*.

```
axd = plt.figure(constrained_layout=True).subplot_mosaic(
    """
    .a.
    bAc
    .d.
    """,
    gridspec_kw={
        # set the height ratios between the rows
        "height_ratios": [1, 3.5, 1],
        # set the width ratios between the columns
        "width_ratios": [1, 3.5, 1],
```

(continues on next page)

(continued from previous page)

```
    },
)
identify_axes(axd)
```



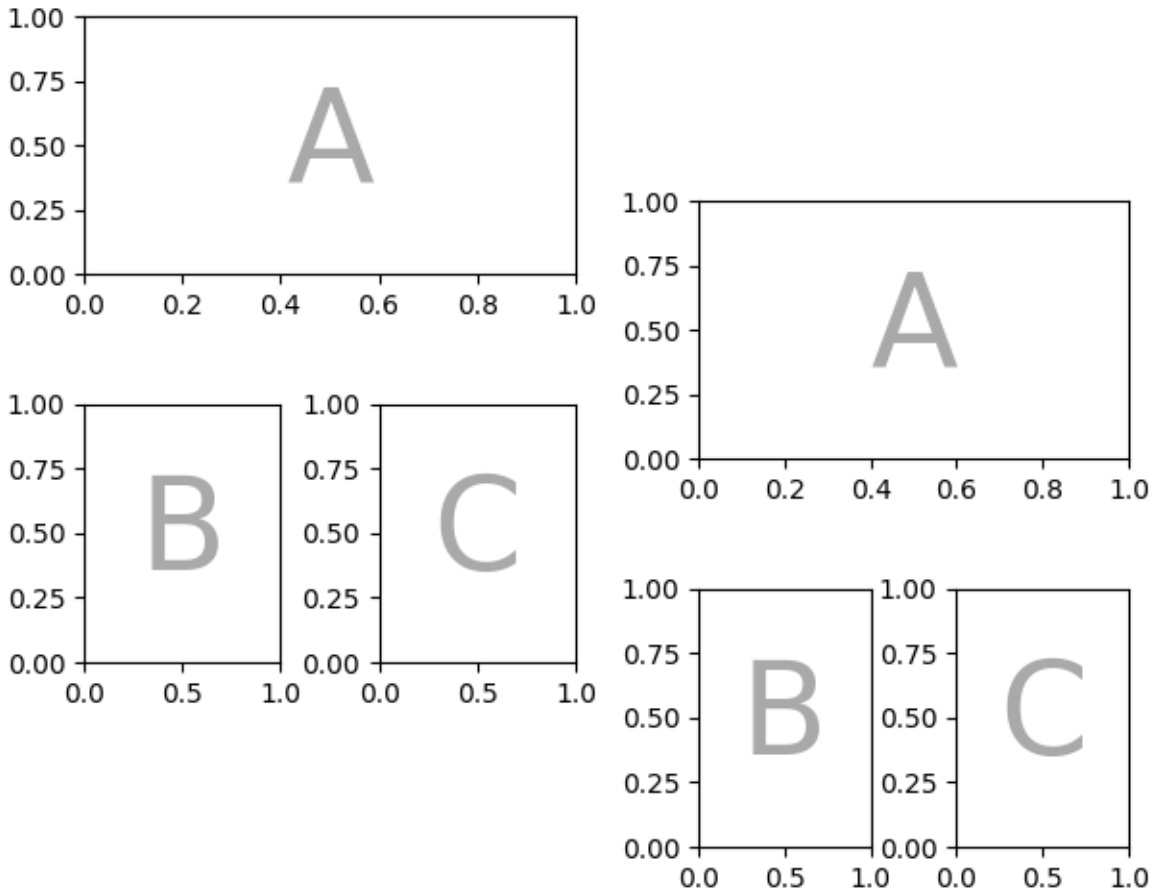
Or use the {*left*, *right*, *bottom*, *top*} keyword arguments to position the overall layout to put multiple versions of the same layout in a figure

```
layout = """AA
            BC"""
fig = plt.figure()
axd = fig.subplot_mosaic(
    layout,
    gridspec_kw={
        "bottom": 0.25,
        "top": 0.95,
        "left": 0.1,
        "right": 0.5,
        "wspace": 0.5,
        "hspace": 0.5,
    },
)
identify_axes(axd)
```

(continues on next page)

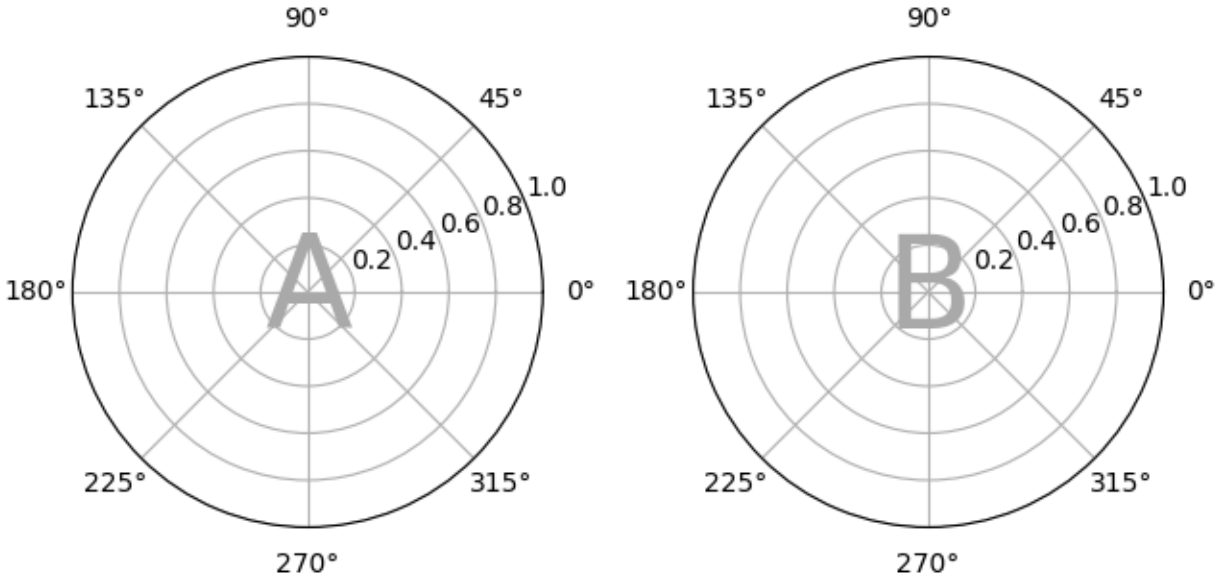
(continued from previous page)

```
axd = fig.subplot_mosaic(
    layout,
    gridspec_kw={
        "bottom": 0.05,
        "top": 0.75,
        "left": 0.6,
        "right": 0.95,
        "wspace": 0.5,
        "hspace": 0.5,
    },
)
identify_axes(axd)
```



We can also pass through arguments used to create the subplots (again, the same as *Figure.subplots*).

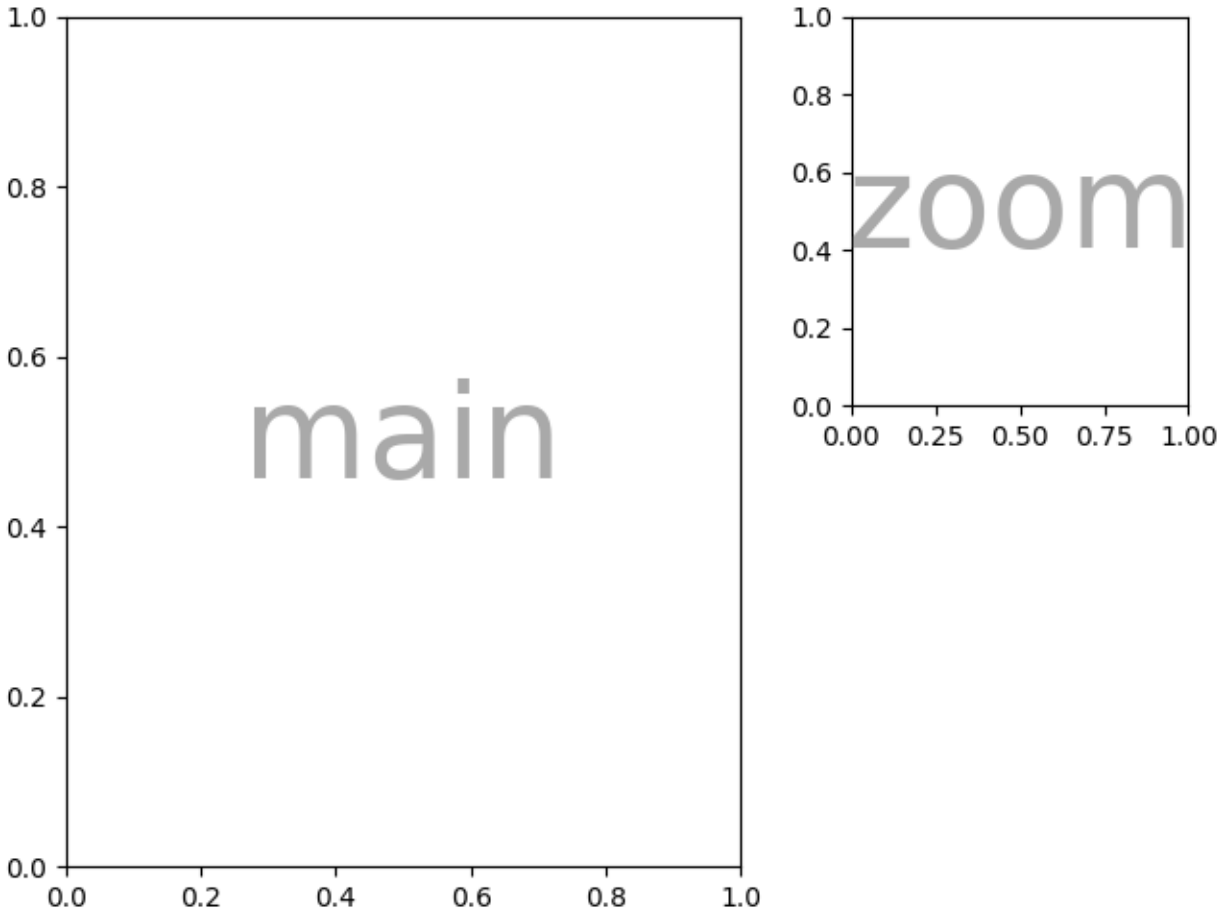
```
axd = plt.figure(constrained_layout=True).subplot_mosaic(
    "AB", subplot_kw={"projection": "polar"}
)
identify_axes(axd)
```



Nested List input

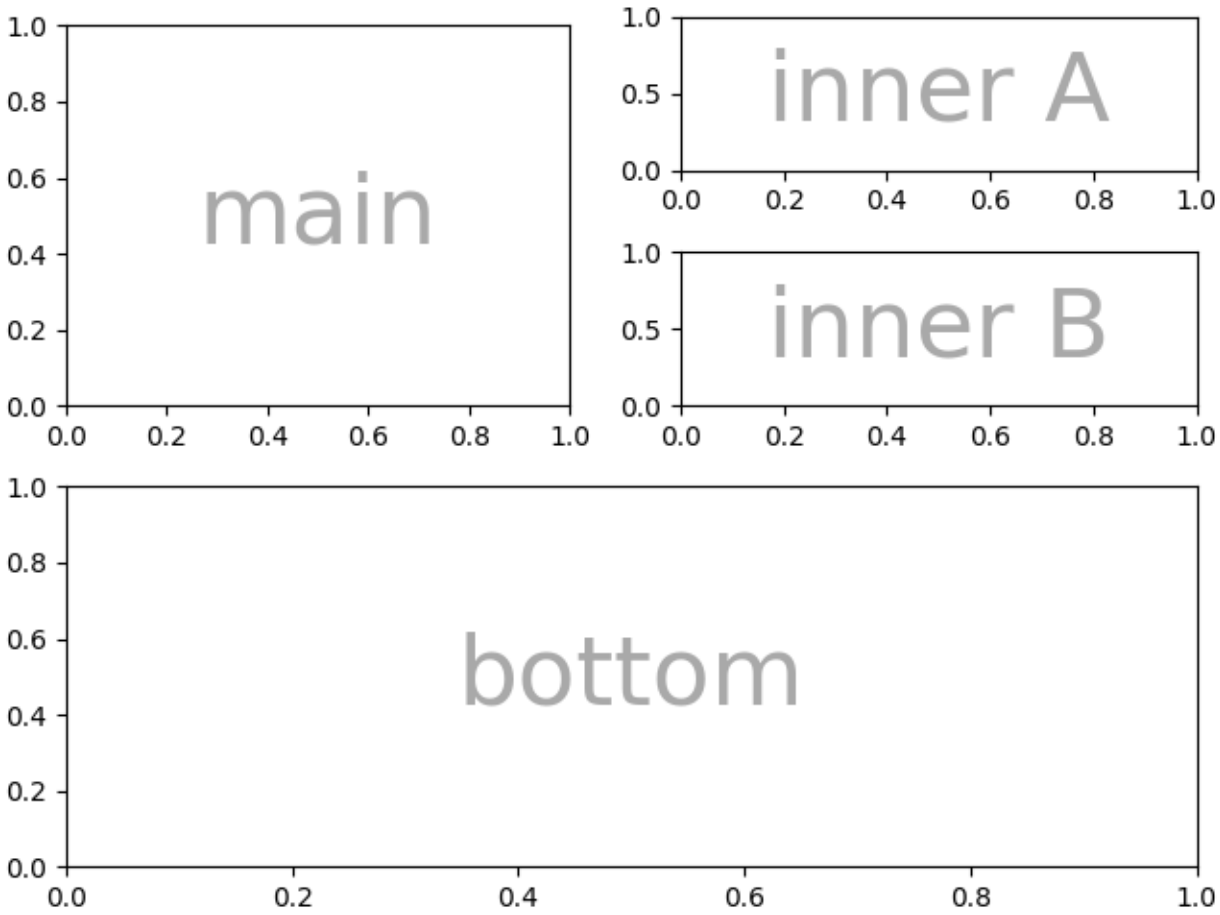
Everything we can do with the string short-hand we can also do when passing in a list (internally we convert the string shorthand to a nested list), for example using spans, blanks, and *gridspec_kw*:

```
axd = plt.figure(constrained_layout=True).subplot_mosaic(
    [["main", "zoom"],
     ["main", "BLANK"]
     ],
    empty_sentinel="BLANK",
    gridspec_kw={"width_ratios": [2, 1]}
)
identify_axes(axd)
```



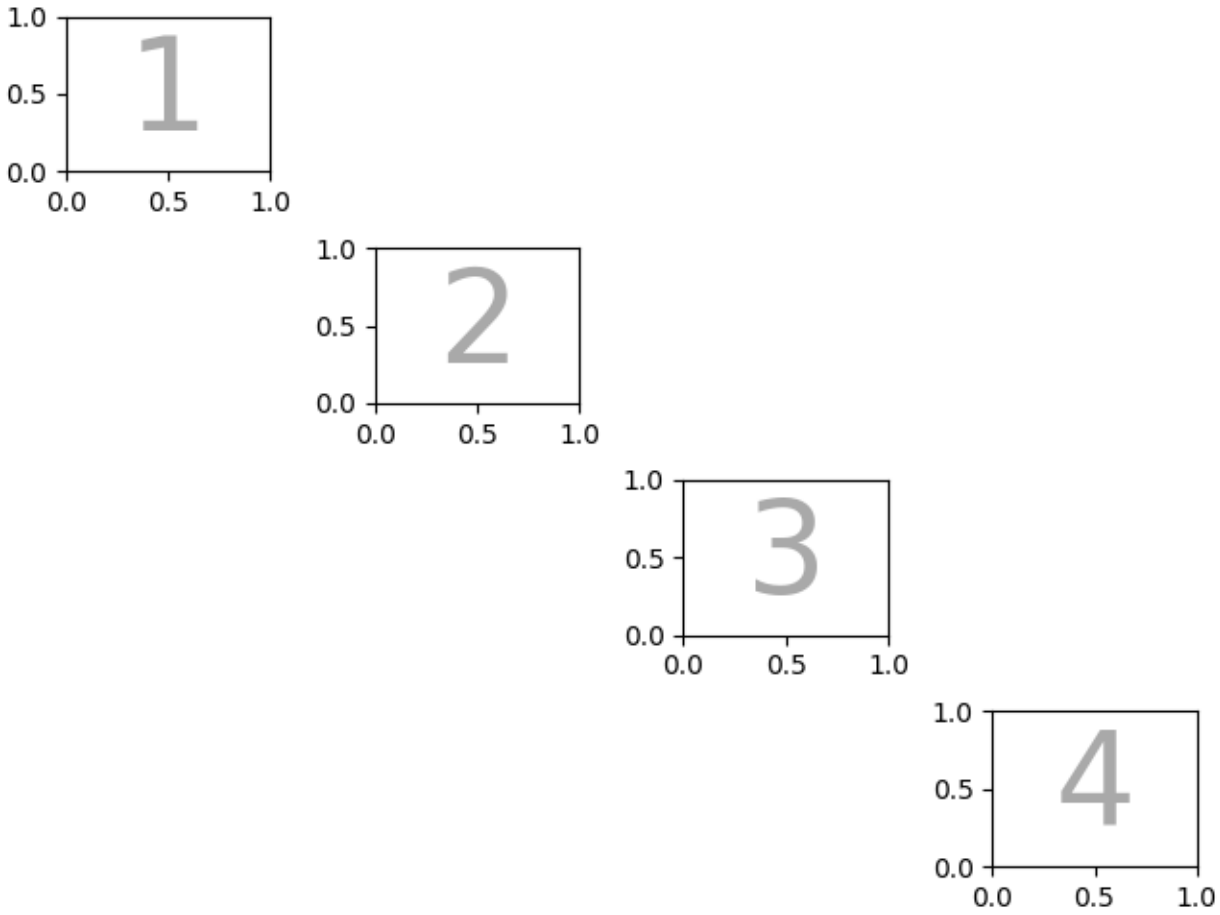
In addition, using the list input we can specify nested layouts. Any element of the inner list can be another set of nested lists:

```
inner = [
    ["inner A"],
    ["inner B"],
]

outer_nested_layout = [
    ["main", inner],
    ["bottom", "bottom"],
]
axd = plt.figure(constrained_layout=True).subplot_mosaic(
    outer_nested_layout, empty_sentinel=None
)
identify_axes(axd, fontsize=36)
```



We can also pass in a 2D NumPy array to do things like

```
layout = np.zeros((4, 4), dtype=int)
for j in range(4):
    layout[j, j] = j + 1
axd = plt.figure(constrained_layout=True).subplot_mosaic(
    layout, empty_sentinel=0
)
identify_axes(axd)
```



Total running time of the script: (0 minutes 5.518 seconds)

2.6 Text

matplotlib has extensive text support, including support for mathematical expressions, truetype support for raster and vector outputs, newline separated text with arbitrary rotations, and unicode support. These tutorials cover the basics of working with text in Matplotlib.

2.6.1 Text in Matplotlib Plots

Introduction to plotting and working with text in Matplotlib.

Matplotlib has extensive text support, including support for mathematical expressions, truetype support for raster and vector outputs, newline separated text with arbitrary rotations, and unicode support.

Because it embeds fonts directly in output documents, e.g., for postscript or PDF, what you see on the screen is what you get in the hardcopy. FreeType support produces very nice, antialiased fonts, that look good even at small raster sizes. Matplotlib

includes its own *matplotlib.font_manager* (thanks to Paul Barrett), which implements a cross platform, W3C compliant font finding algorithm.

The user has a great deal of control over text properties (font size, font weight, text location and color, etc.) with sensible defaults set in the *rc file*. And significantly, for those interested in mathematical or scientific figures, Matplotlib implements a large number of TeX math symbols and commands, supporting *mathematical expressions* anywhere in your figure.

Basic text commands

The following commands are used to create text in the pyplot interface and the object-oriented API:

pyplot API	OO API	description
text	*text*	Add text at an arbitrary location of the *Axes*.
annotate	*annotate*	Add an annotation, with an optional arrow, at an arbitrary location of the *Axes*.
xlabel	*set_xlabel*	Add a label to the *Axes*'s x-axis.
ylabel	*set_ylabel*	Add a label to the *Axes*'s y-axis.
title	*set_title*	Add a title to the *Axes*.
figtext	*text*	Add text at an arbitrary location of the *Figure*.
suptitle	*suptitle*	Add a title to the *Figure*.

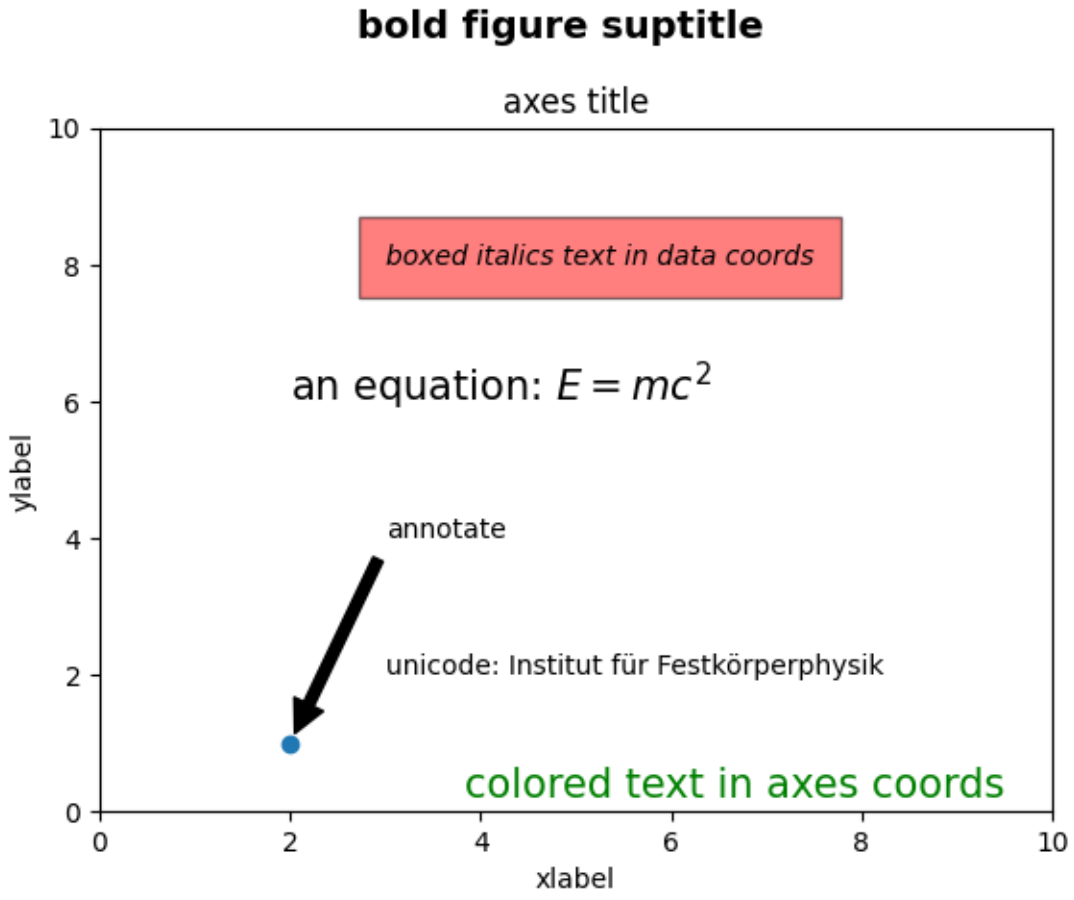
All of these functions create and return a *Text* instance, which can be configured with a variety of font and other properties. The example below shows all of these commands in action, and more detail is provided in the sections that follow.

```
import matplotlib
import matplotlib.pyplot as plt

fig = plt.figure()
ax = fig.add_subplot(111)
fig.subplots_adjust(top=0.85)

# Set titles for the figure and the subplot respectively
fig.suptitle('bold figure suptitle', fontsize=14, fontweight='bold')
ax.set_title('axes title')

ax.set_xlabel('xlabel')
ax.set_ylabel('ylabel')

# Set both x- and y-axis limits to [0, 10] instead of default [0, 1]
ax.axis([0, 10, 0, 10])

ax.text(3, 8, 'boxed italics text in data coords', style='italic',
        bbox={'facecolor': 'red', 'alpha': 0.5, 'pad': 10})
```

(continues on next page)

(continued from previous page)

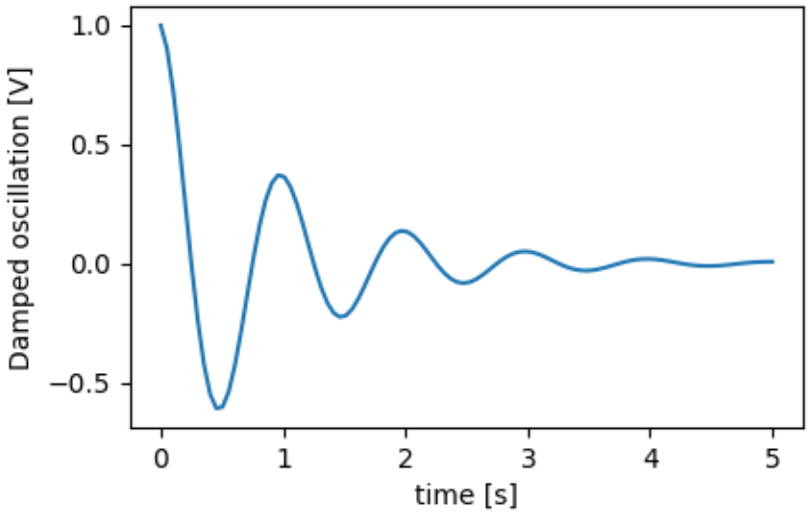
```
ax.text(2, 6, r'an equation: $E=mc^2$', fontsize=15)

ax.text(3, 2, 'unicode: Institut für Festkörperphysik')

ax.text(0.95, 0.01, 'colored text in axes coords',
        verticalalignment='bottom', horizontalalignment='right',
        transform=ax.transAxes,
        color='green', fontsize=15)

ax.plot([2], [1], 'o')
ax.annotate('annotate', xy=(2, 1), xytext=(3, 4),
            arrowprops=dict(facecolor='black', shrink=0.05))

plt.show()
```



Labels for x- and y-axis

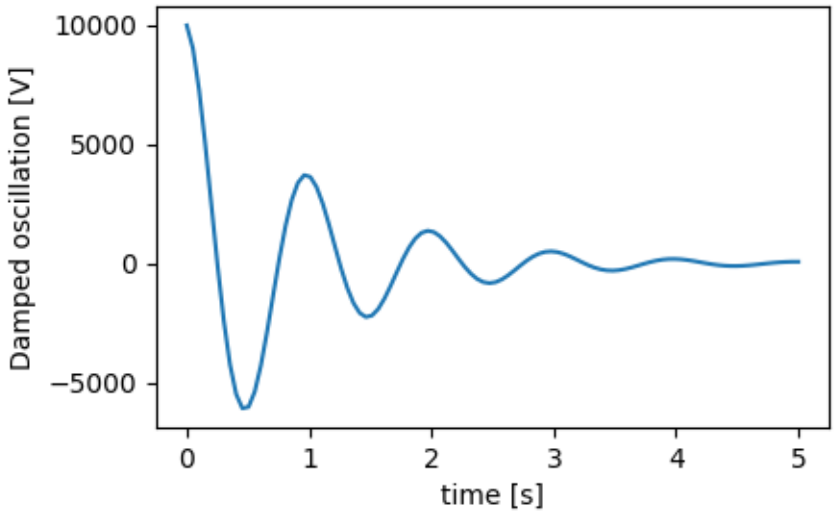
Specifying the labels for the x- and y-axis is straightforward, via the *set_xlabel* and *set_ylabel* methods.

```
import matplotlib.pyplot as plt
import numpy as np

x1 = np.linspace(0.0, 5.0, 100)
y1 = np.cos(2 * np.pi * x1) * np.exp(-x1)

fig, ax = plt.subplots(figsize=(5, 3))
fig.subplots_adjust(bottom=0.15, left=0.2)
ax.plot(x1, y1)
ax.set_xlabel('time [s]')
ax.set_ylabel('Damped oscillation [V]')

plt.show()
```



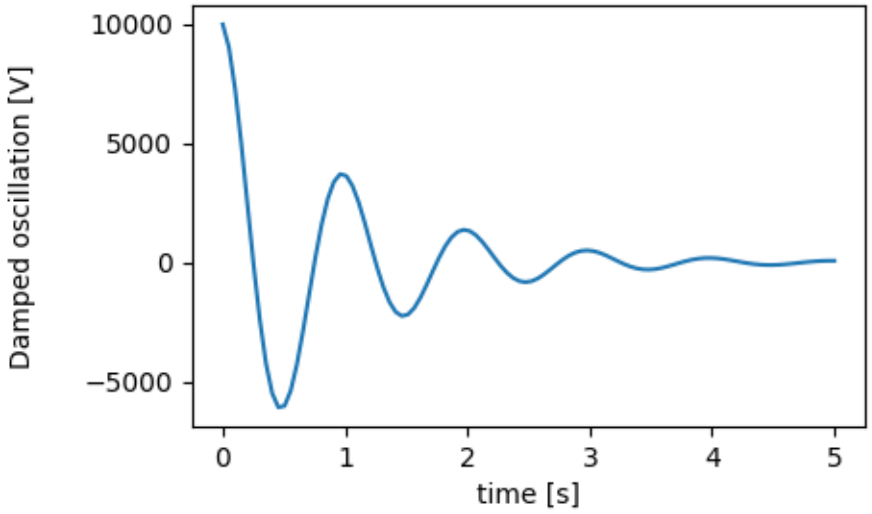
The x- and y-labels are automatically placed so that they clear the x- and y-ticklabels. Compare the plot below with that above, and note the y-label is to the left of the one above.

```
fig, ax = plt.subplots(figsize=(5, 3))
fig.subplots_adjust(bottom=0.15, left=0.2)
ax.plot(x1, y1*10000)
ax.set_xlabel('time [s]')
ax.set_ylabel('Damped oscillation [V]')

plt.show()
```



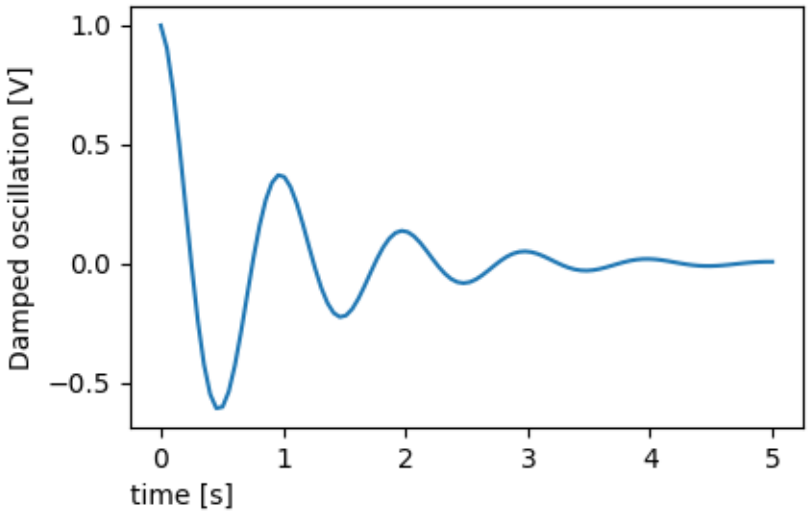
If you want to move the labels, you can specify the *labelpad* keyword argument, where the value is points (1/72", the same unit used to specify fontsizes).

```
fig, ax = plt.subplots(figsize=(5, 3))
fig.subplots_adjust(bottom=0.15, left=0.2)
ax.plot(x1, y1*10000)
ax.set_xlabel('time [s]')
ax.set_ylabel('Damped oscillation [V]', labelpad=18)

plt.show()
```



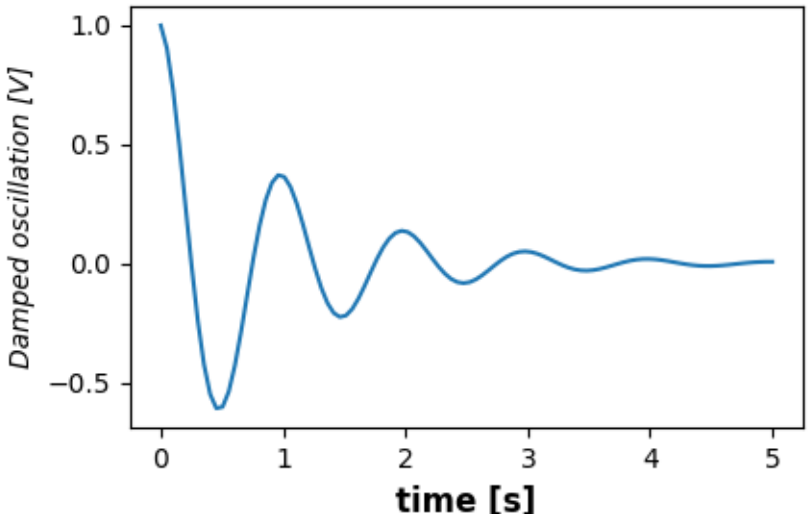
Or, the labels accept all the *Text* keyword arguments, including *position*, via which we can manually specify the label positions. Here we put the xlabel to the far left of the axis. Note, that the y-coordinate of this position has no effect - to adjust the y-position we need to use the *labelpad* kwarg.

```
fig, ax = plt.subplots(figsize=(5, 3))
fig.subplots_adjust(bottom=0.15, left=0.2)
ax.plot(x1, y1)
ax.set_xlabel('time [s]', position=(0., 1e6), horizontalalignment='left')
ax.set_ylabel('Damped oscillation [V]')

plt.show()
```



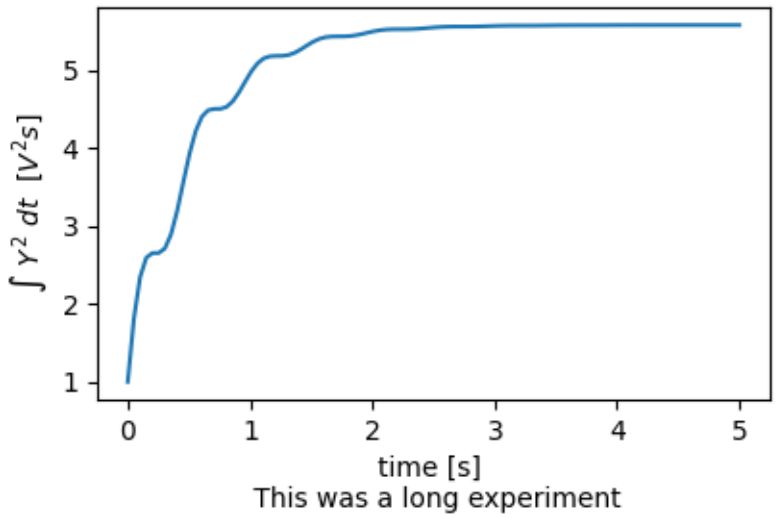
All the labelling in this tutorial can be changed by manipulating the *matplotlib.font_manager.FontProperties* method, or by named kwargs to *set_xlabel*

```
from matplotlib.font_manager import FontProperties

font = FontProperties()
font.set_family('serif')
font.set_name('Times New Roman')
font.set_style('italic')

fig, ax = plt.subplots(figsize=(5, 3))
fig.subplots_adjust(bottom=0.15, left=0.2)
ax.plot(x1, y1)
ax.set_xlabel('time [s]', fontsize='large', fontweight='bold')
ax.set_ylabel('Damped oscillation [V]', fontproperties=font)

plt.show()
```



Finally, we can use native TeX rendering in all text objects and have multiple lines:

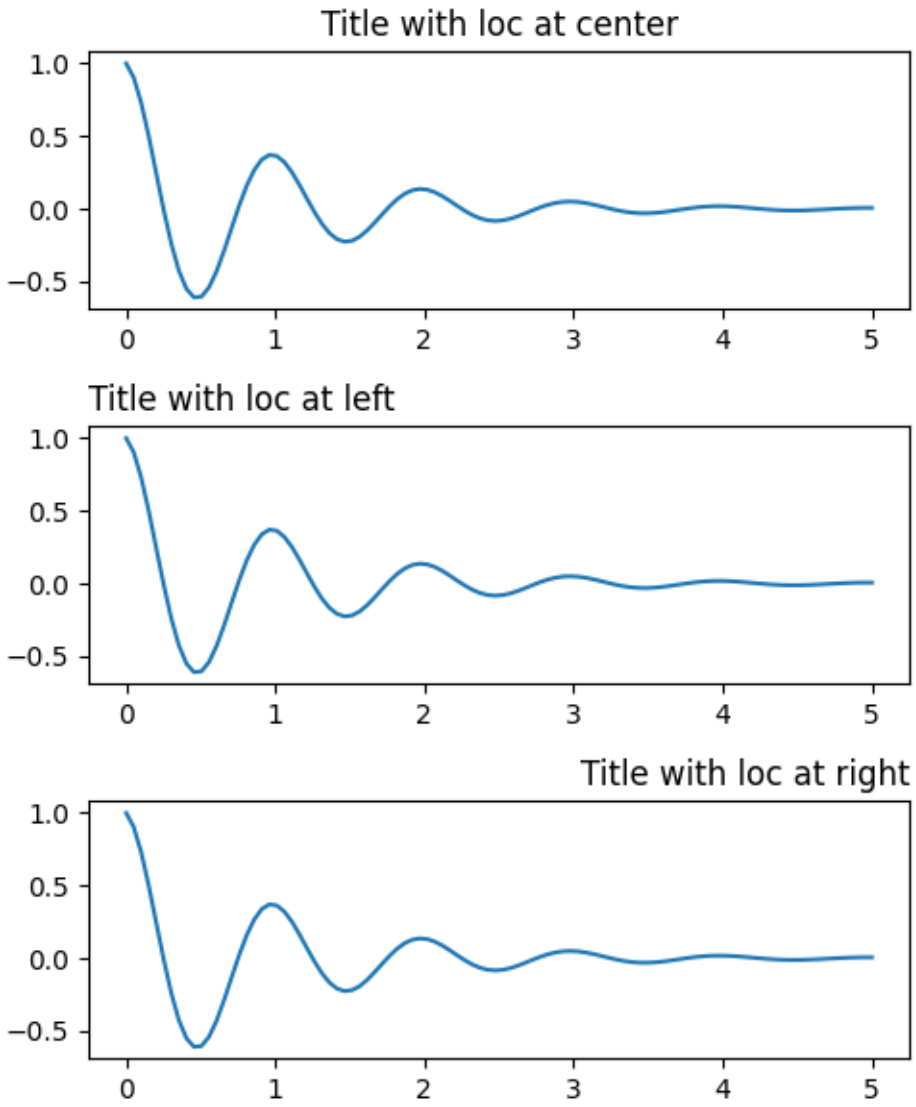
```
fig, ax = plt.subplots(figsize=(5, 3))
fig.subplots_adjust(bottom=0.2, left=0.2)
ax.plot(x1, np.cumsum(y1**2))
ax.set_xlabel('time [s] \n This was a long experiment')
ax.set_ylabel(r'$\int\ Y^2\ dt\ \ [V^2 s]$')
plt.show()
```



Titles

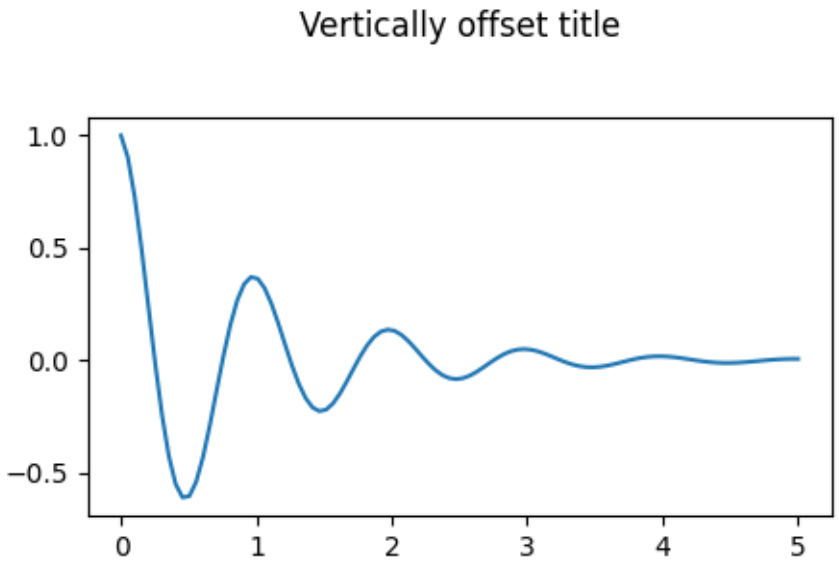
Subplot titles are set in much the same way as labels, but there is the *loc* keyword arguments that can change the position and justification from the default value of `loc=center`.

```
fig, axs = plt.subplots(3, 1, figsize=(5, 6), tight_layout=True)
locs = ['center', 'left', 'right']
for ax, loc in zip(axs, locs):
    ax.plot(x1, y1)
    ax.set_title('Title with loc at '+loc, loc=loc)
plt.show()
```



Vertical spacing for titles is controlled via `rcParams["axes.titlepad"]` (default: `6.0`), which defaults to 5 points. Setting to a different value moves the title.

```
fig, ax = plt.subplots(figsize=(5, 3))
fig.subplots_adjust(top=0.8)
ax.plot(x1, y1)
ax.set_title('Vertically offset title', pad=30)
plt.show()
```



Ticks and ticklabels

Placing ticks and ticklabels is a very tricky aspect of making a figure. Matplotlib does its best to accomplish the task automatically, but it also offers a very flexible framework for determining the choices for tick locations, and how they are labelled.

Terminology

Axes have an *matplotlib.axis.Axis* object for the `ax.xaxis` and `ax.yaxis` that contain the information about how the labels in the axis are laid out.

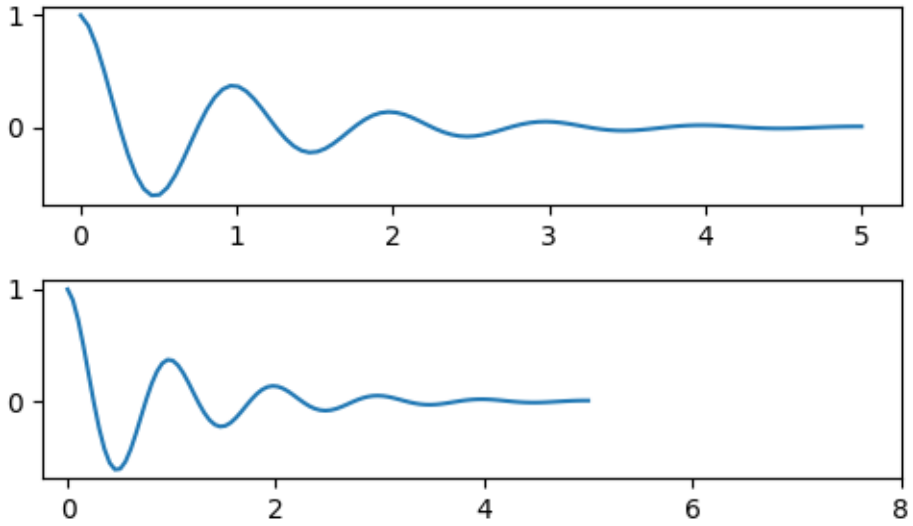
The axis API is explained in detail in the documentation to *axis*.

An Axis object has major and minor ticks. The Axis has *Axis.set_major_locator* and *Axis.set_minor_locator* methods that use the data being plotted to determine the location of major and minor ticks. There are also *Axis.set_major_formatter* and *Axis.set_minor_formatter* methods that format the tick labels.

Simple ticks

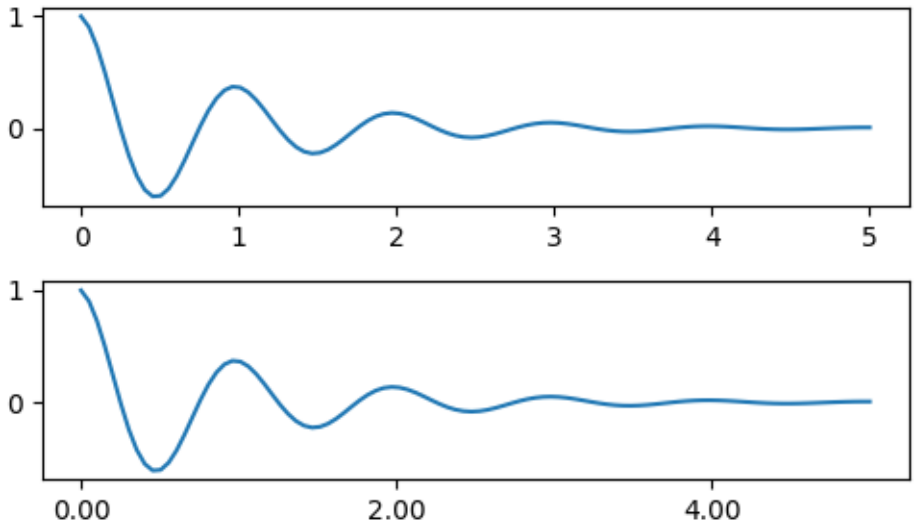
It often is convenient to simply define the tick values, and sometimes the tick labels, overriding the default locators and formatters. This is discouraged because it breaks interactive navigation of the plot. It also can reset the axis limits: note that the second plot has the ticks we asked for, including ones that are well outside the automatic view limits.

```
fig, axs = plt.subplots(2, 1, figsize=(5, 3), tight_layout=True)
axs[0].plot(x1, y1)
axs[1].plot(x1, y1)
axs[1].xaxis.set_ticks(np.arange(0., 8.1, 2.))
plt.show()
```



We can of course fix this after the fact, but it does highlight a weakness of hard-coding the ticks. This example also changes the format of the ticks:

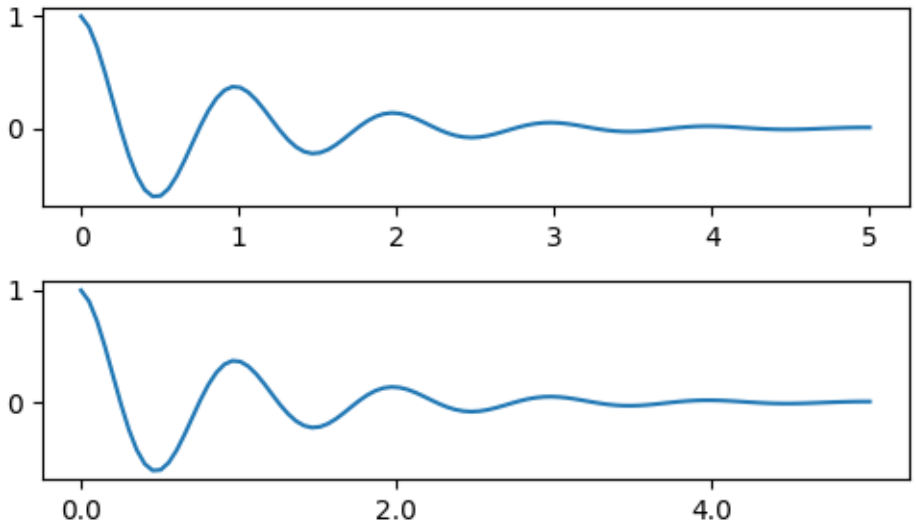
```
fig, axs = plt.subplots(2, 1, figsize=(5, 3), tight_layout=True)
axs[0].plot(x1, y1)
axs[1].plot(x1, y1)
ticks = np.arange(0., 8.1, 2.)
# list comprehension to get all tick labels...
tickla = [f'{tick:1.2f}' for tick in ticks]
axs[1].xaxis.set_ticks(ticks)
axs[1].xaxis.set_ticklabels(tickla)
axs[1].set_xlim(axs[0].get_xlim())
plt.show()
```



Tick Locators and Formatters

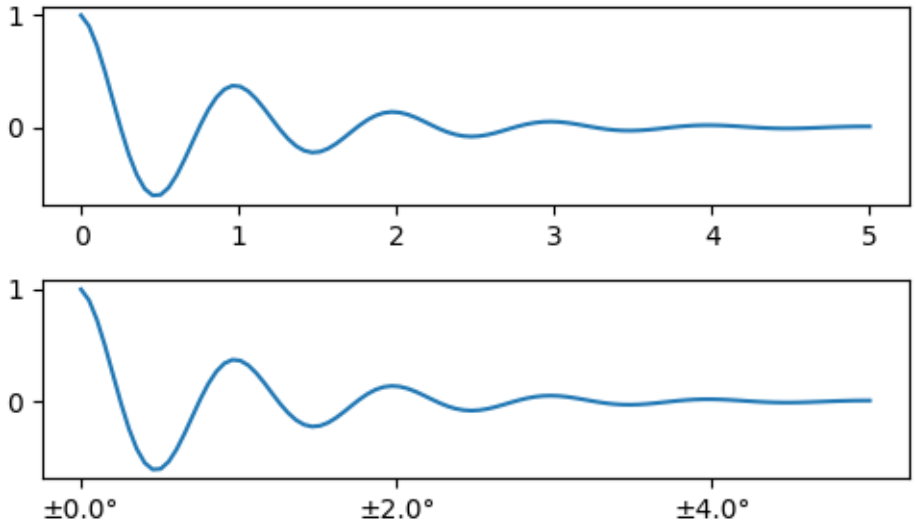
Instead of making a list of all the tickalbels, we could have used *matplotlib.ticker.StrMethodFormatter* (new-style `str.format()` format string) or *matplotlib.ticker.FormatStrFormatter* (old-style '%' format string) and passed it to the `ax.xaxis`. A *matplotlib.ticker.StrMethodFormatter* can also be created by passing a `str` without having to explicitly create the formatter.

```
fig, axs = plt.subplots(2, 1, figsize=(5, 3), tight_layout=True)
axs[0].plot(x1, y1)
axs[1].plot(x1, y1)
ticks = np.arange(0., 8.1, 2.)
axs[1].xaxis.set_ticks(ticks)
axs[1].xaxis.set_major_formatter('{x:1.1f}')
axs[1].set_xlim(axs[0].get_xlim())
plt.show()
```



And of course we could have used a non-default locator to set the tick locations. Note we still pass in the tick values, but the x-limit fix used above is *not* needed.

```
fig, axs = plt.subplots(2, 1, figsize=(5, 3), tight_layout=True)
axs[0].plot(x1, y1)
axs[1].plot(x1, y1)
locator = matplotlib.ticker.FixedLocator(ticks)
axs[1].xaxis.set_major_locator(locator)
axs[1].xaxis.set_major_formatter('±{x}°')
plt.show()
```

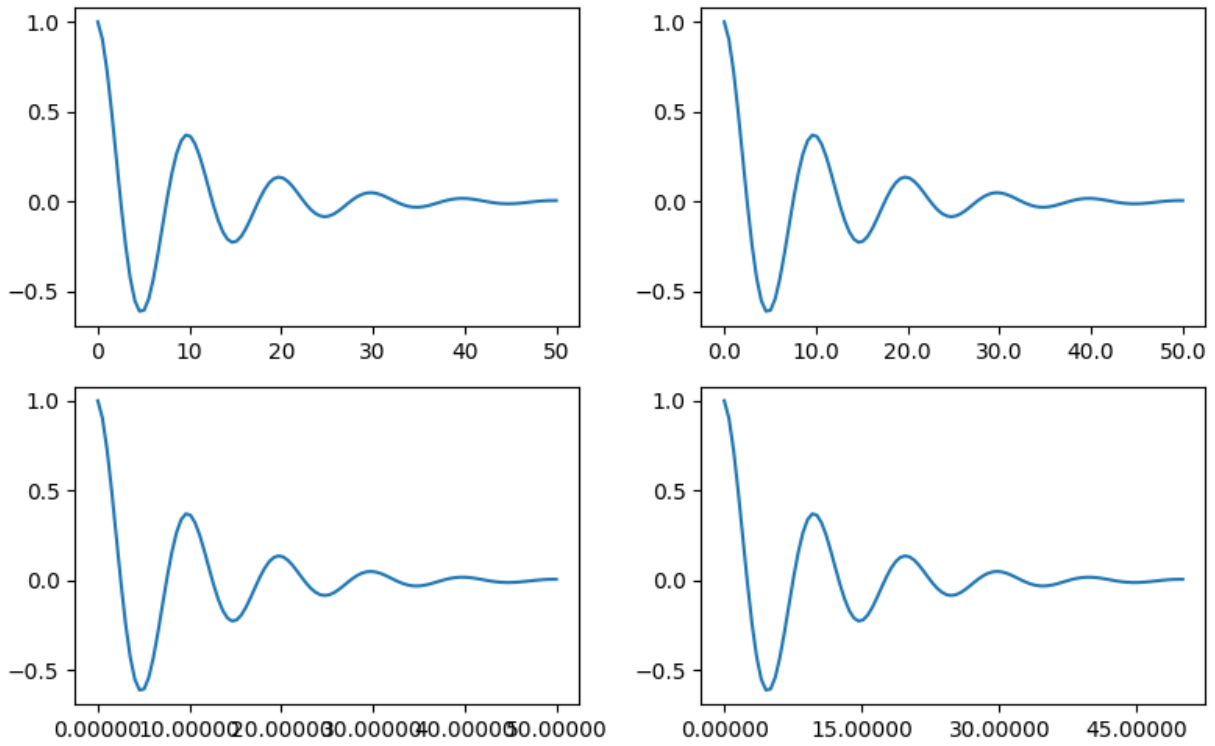


The default formatter is the *matplotlib.ticker.MaxNLocator* called as `ticker.MaxNLocator(self, nbins='auto', steps=[1, 2, 2.5, 5, 10])` The *steps* keyword contains a list of multiples that can be used for tick values. i.e. in this case, 2, 4, 6 would be acceptable ticks, as would 20, 40, 60 or 0.2, 0.4, 0.6. However, 3, 6, 9

would not be acceptable because 3 doesn't appear in the list of steps.

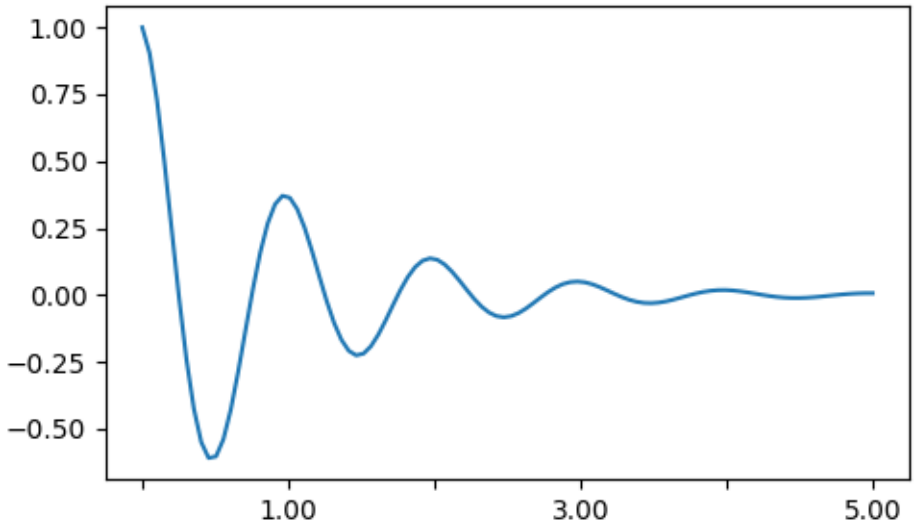
`nbins=auto` uses an algorithm to determine how many ticks will be acceptable based on how long the axis is. The fontsize of the ticklabel is taken into account, but the length of the tick string is not (because its not yet known.) In the bottom row, the ticklabels are quite large, so we set `nbins=4` to make the labels fit in the right-hand plot.

```
fig, axs = plt.subplots(2, 2, figsize=(8, 5), tight_layout=True)
for n, ax in enumerate(axs.flat):
    ax.plot(x1*10., y1)

formatter = matplotlib.ticker.FormatStrFormatter('%1.1f')
locator = matplotlib.ticker.MaxNLocator(nbins='auto', steps=[1, 4, 10])
axs[0, 1].xaxis.set_major_locator(locator)
axs[0, 1].xaxis.set_major_formatter(formatter)

formatter = matplotlib.ticker.FormatStrFormatter('%1.5f')
locator = matplotlib.ticker.AutoLocator()
axs[1, 0].xaxis.set_major_formatter(formatter)
axs[1, 0].xaxis.set_major_locator(locator)

formatter = matplotlib.ticker.FormatStrFormatter('%1.5f')
locator = matplotlib.ticker.MaxNLocator(nbins=4)
axs[1, 1].xaxis.set_major_formatter(formatter)
axs[1, 1].xaxis.set_major_locator(locator)

plt.show()
```



Finally, we can specify functions for the formatter using *matplotlib.ticker.FuncFormatter*. Further, like *matplotlib.ticker.StrMethodFormatter*, passing a function will automatically create a *matplotlib.ticker.FuncFormatter*.

```
def formatoddticks(x, pos):
    """Format odd tick positions."""
    if x % 2:
        return f'{x:1.2f}'
    else:
        return ''


fig, ax = plt.subplots(figsize=(5, 3), tight_layout=True)
ax.plot(x1, y1)
locator = matplotlib.ticker.MaxNLocator(nbins=6)
ax.xaxis.set_major_formatter(formatoddticks)
ax.xaxis.set_major_locator(locator)

plt.show()
```



Dateticks

Matplotlib can accept `datetime.datetime` and `numpy.datetime64` objects as plotting arguments. Dates and times require special formatting, which can often benefit from manual intervention. In order to help, dates have special Locators and Formatters, defined in the *matplotlib.dates* module.

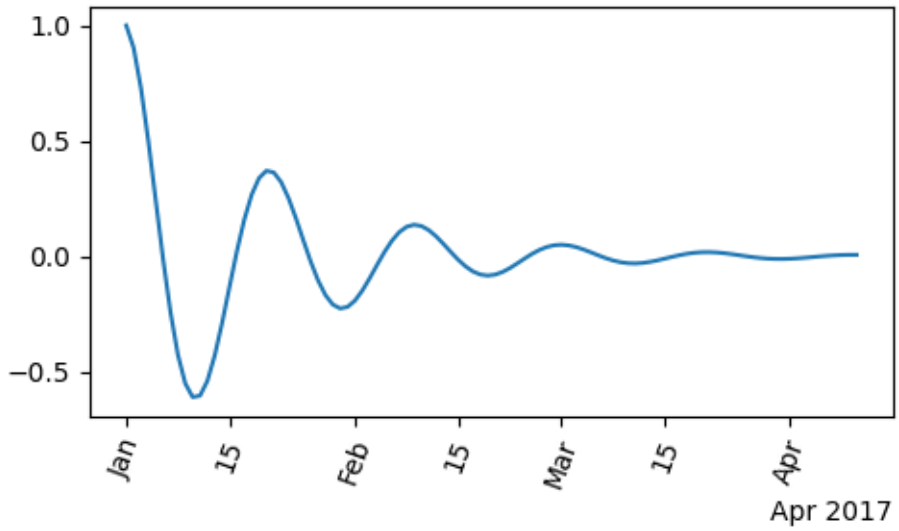
A simple example is as follows. Note how we have to rotate the tick labels so that they don't over-run each other.

```
import datetime
```

(continues on next page)

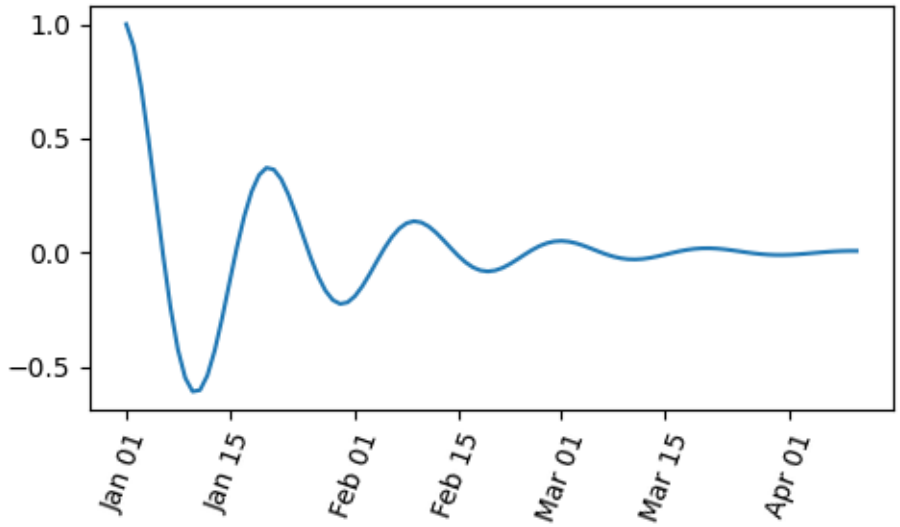
(continued from previous page)

```
fig, ax = plt.subplots(figsize=(5, 3), tight_layout=True)
base = datetime.datetime(2017, 1, 1, 0, 0, 1)
time = [base + datetime.timedelta(days=x) for x in range(len(x1))]

ax.plot(time, y1)
ax.tick_params(axis='x', rotation=70)
plt.show()
```



We can pass a format to *matplotlib.dates.DateFormatter*. Also note that the 29th and the next month are very close together. We can fix this by using the *dates.DayLocator* class, which allows us to specify a list of days of the month to use. Similar formatters are listed in the *matplotlib.dates* module.

```
import matplotlib.dates as mdates

locator = mdates.DayLocator(bymonthday=[1, 15])
formatter = mdates.DateFormatter('%b %d')

fig, ax = plt.subplots(figsize=(5, 3), tight_layout=True)
ax.xaxis.set_major_locator(locator)
ax.xaxis.set_major_formatter(formatter)
ax.plot(time, y1)
ax.tick_params(axis='x', rotation=70)
plt.show()
```



Legends and Annotations

- Legends: *Legend guide*
- Annotations: *Annotations*

Total running time of the script: (0 minutes 4.725 seconds)

2.6.2 Text properties and layout

Controlling properties of text and its layout with Matplotlib.

matplotlib.text.Text instances have a variety of properties which can be configured via keyword arguments to *set_title*, *set_xlabel*, *text*, etc.

Property	Value Type
alpha	float
backgroundcolor	any matplotlib *color*
bbox	*Rectangle* prop dict plus key 'pad' which is a pad in points
clip_box	a matplotlib.transform.Bbox instance
clip_on	bool
clip_path	a *Path* instance and a *Transform* instance, a *Patch*
color	any matplotlib *color*
family	['serif' | 'sans-serif' | 'cursive' | 'fantasy' | 'monospace']
fontproperties	*FontProperties*
horizontalalignment or ha	['center' | 'right' | 'left']
label	any string
linespacing	float
multialignment	['left' | 'right' | 'center']
name or fontname	string e.g., ['Sans' | 'Courier' | 'Helvetica' ...]
picker	[None|float|bool|callable]
position	(x, y)
rotation	[angle in degrees | 'vertical' | 'horizontal']
size or fontsize	[size in points | relative size, e.g., 'smaller', 'x-large']
style or fontstyle	['normal' | 'italic' | 'oblique']
text	string or anything printable with '%s' conversion
transform	*Transform* subclass
variant	['normal' | 'small-caps']
verticalalignment or va	['center' | 'top' | 'bottom' | 'baseline']
visible	bool
weight or fontweight	['normal' | 'bold' | 'heavy' | 'light' | 'ultrabold' | 'ultralight']
x	float
y	float
zorder	any number

You can lay out text with the alignment arguments `horizontalalignment`, `verticalalignment`, and `multialignment`. `horizontalalignment` controls whether the x positional argument for the text indicates the left, center or right side of the text bounding box. `verticalalignment` controls whether the y positional argument for the text indicates the bottom, center or top side of the text bounding box. `multialignment`, for newline separated strings only, controls whether the different lines are left, center or right justified. Here is an example which uses the *text()* command to show the various alignment possibilities. The use of `transform=ax.transAxes` throughout the code indicates that the coordinates are given relative to the axes bounding box, with (0, 0) being the lower left of the axes and (1, 1) the upper right.

```
import matplotlib.pyplot as plt
```

(continues on next page)

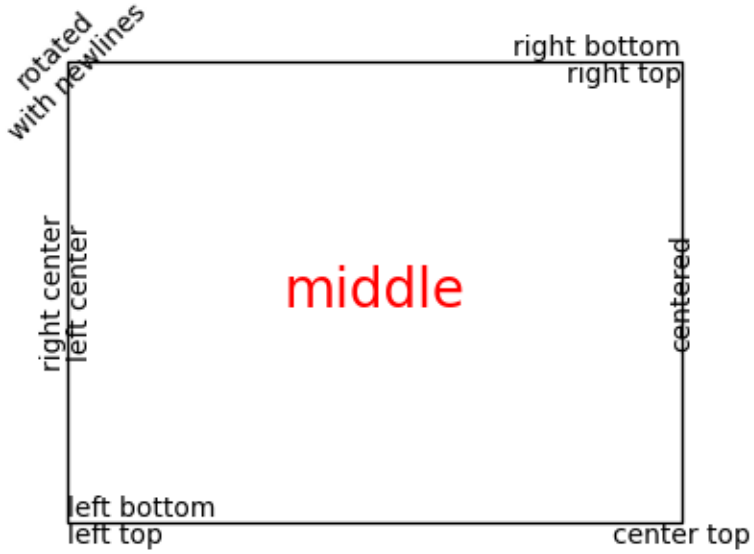
(continued from previous page)

```
import matplotlib.patches as patches

# build a rectangle in axes coords
left, width = .25, .5
bottom, height = .25, .5
right = left + width
top = bottom + height

fig = plt.figure()
ax = fig.add_axes([0, 0, 1, 1])

# axes coordinates: (0, 0) is bottom left and (1, 1) is upper right
p = patches.Rectangle(
    (left, bottom), width, height,
    fill=False, transform=ax.transAxes, clip_on=False
    )

ax.add_patch(p)

ax.text(left, bottom, 'left top',
        horizontalalignment='left',
        verticalalignment='top',
        transform=ax.transAxes)

ax.text(left, bottom, 'left bottom',
        horizontalalignment='left',
        verticalalignment='bottom',
        transform=ax.transAxes)

ax.text(right, top, 'right bottom',
        horizontalalignment='right',
        verticalalignment='bottom',
        transform=ax.transAxes)

ax.text(right, top, 'right top',
        horizontalalignment='right',
        verticalalignment='top',
        transform=ax.transAxes)

ax.text(right, bottom, 'center top',
        horizontalalignment='center',
        verticalalignment='top',
        transform=ax.transAxes)

ax.text(left, 0.5*(bottom+top), 'right center',
        horizontalalignment='right',
        verticalalignment='center',
        rotation='vertical',
        transform=ax.transAxes)

ax.text(left, 0.5*(bottom+top), 'left center',
```

(continues on next page)

(continued from previous page)

```
        horizontalalignment='left',
        verticalalignment='center',
        rotation='vertical',
        transform=ax.transAxes)

ax.text(0.5*(left+right), 0.5*(bottom+top), 'middle',
        horizontalalignment='center',
        verticalalignment='center',
        fontsize=20, color='red',
        transform=ax.transAxes)

ax.text(right, 0.5*(bottom+top), 'centered',
        horizontalalignment='center',
        verticalalignment='center',
        rotation='vertical',
        transform=ax.transAxes)

ax.text(left, top, 'rotated\nwith newlines',
        horizontalalignment='center',
        verticalalignment='center',
        rotation=45,
        transform=ax.transAxes)

ax.set_axis_off()
plt.show()
```



2.6.3 Default Font

The base default font is controlled by a set of rcParams. To set the font for mathematical expressions, use the rcParams beginning with `mathtext` (see *mathtext*).

rcParam	usage
`'font.family'`	List of either names of font or `{'cursive', 'fantasy', 'monospace', 'sans', 'sans serif', 'sans-serif', 'serif'}`.
`'font.style'`	The default style, ex `'normal'`, `'italic'`.
`'font.variant'`	Default variant, ex `'normal'`, `'small-caps'` (untested)
`'font.stretch'`	Default stretch, ex `'normal'`, `'condensed'` (incomplete)
`'font.weight'`	Default weight. Either string or integer
`'font.size'`	Default font size in points. Relative font sizes (`'large'`, `'x-small'`) are computed against this size.

The mapping between the family aliases ({'cursive', 'fantasy', 'monospace', 'sans', 'sans serif', 'sans-serif', 'serif'}) and actual font names is controlled by the following rcParams:

family alias	rcParam with mappings
'serif'	'font.serif'
'monospace'	'font.monospace'
'fantasy'	'font.fantasy'
'cursive'	'font.cursive'
{'sans', 'sans serif', 'sans-serif'}	'font.sans-serif'

which are lists of font names.

Text with non-latin glyphs

As of v2.0 the *default font*, DejaVu, contains glyphs for many western alphabets, but not other scripts, such as Chinese, Korean, or Japanese.

To set the default font to be one that supports the code points you need, prepend the font name to 'font.family' or the desired alias lists

```
matplotlib.rcParams['font.sans-serif'] = ['Source Han Sans TW', 'sans-serif']
```

or set it in your .matplotlibrc file:

```
font.sans-serif: Source Han Sans TW, Arial, sans-serif
```

To control the font used on per-artist basis use the 'name', 'fontname' or 'fontproperties' kwargs documented *above*.

On linux, fc-list can be a useful tool to discover the font name; for example

```
$ fc-list :lang=zh family
Noto to Sans Mono CJK TC,Noto Sans Mono CJK TC Bold
Noto Sans CJK TC,Noto Sans CJK TC Medium
Noto Sans CJK TC,Noto Sans CJK TC DemiLight
Noto Sans CJK KR,Noto Sans CJK KR Black
Noto Sans CJK TC,Noto Sans CJK TC Black
Noto Sans Mono CJK TC,Noto Sans Mono CJK TC Regular
Noto Sans CJK SC,Noto Sans CJK SC Light
```

lists all of the fonts that support Chinese.

2.6.4 Annotations

Annotating text with Matplotlib.

Table of Contents

Basic annotation

The uses of the basic *text()* will place text at an arbitrary position on the Axes. A common use case of text is to annotate some feature of the plot, and the *annotate()* method provides helper functionality to make annotations easy. In an annotation, there are two points to consider: the location being annotated represented by the argument *xy* and the location of the text *xytext*. Both of these arguments are `(x, y)` tuples.

In this example, both the *xy* (arrow tip) and *xytext* locations (text location) are in data coordinates. There are a variety of other coordinate systems one can choose -- you can specify the coordinate system of *xy* and *xytext* with one of the following strings for *xycoords* and *textcoords* (default is 'data')

argument	coordinate system
'figure points'	points from the lower left corner of the figure
'figure pixels'	pixels from the lower left corner of the figure
'figure fraction'	(0, 0) is lower left of figure and (1, 1) is upper right
'axes points'	points from lower left corner of axes
'axes pixels'	pixels from lower left corner of axes
'axes fraction'	(0, 0) is lower left of axes and (1, 1) is upper right
'data'	use the axes data coordinate system

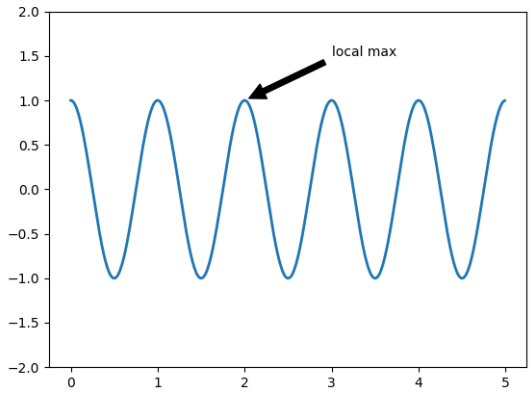


Fig. 23: Annotation Basic

For example to place the text coordinates in fractional axes coordinates, one could do:

```
ax.annotate('local max', xy=(3, 1),  xycoords='data',
            xytext=(0.8, 0.95), textcoords='axes fraction',
            arrowprops=dict(facecolor='black', shrink=0.05),
            horizontalalignment='right', verticalalignment='top',
            )
```

For physical coordinate systems (points or pixels) the origin is the bottom-left of the figure or axes.

Optionally, you can enable drawing of an arrow from the text to the annotated point by giving a dictionary of arrow properties in the optional keyword argument *arrowprops*.

arrowprops key	description
width	the width of the arrow in points
frac	the fraction of the arrow length occupied by the head
headwidth	the width of the base of the arrow head in points
shrink	move the tip and base some percent away from the annotated point and text
**kwargs	any key for *matplotlib.patches.Polygon*, e.g., `facecolor`

In the example below, the *xy* point is in native coordinates (*xycoords* defaults to 'data'). For a polar axes, this is in (theta, radius) space. The text in this example is placed in the fractional figure coordinate system. *matplotlib.text.Text* keyword arguments like *horizontalalignment*, *verticalalignment* and *fontsize* are passed from *annotate* to the `Text` instance.

For more on all the wild and wonderful things you can do with annotations, including fancy arrows, see *Advanced Annotations* and /gallery/text_labels_and_annotations/annotation_demo.

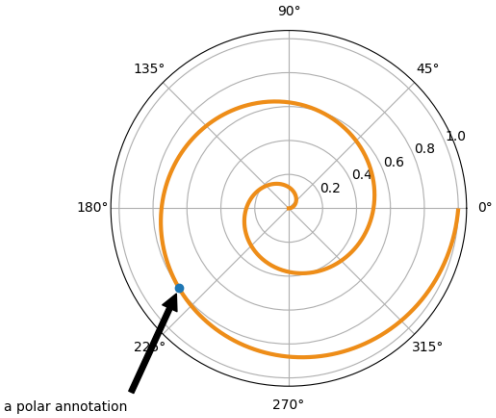


Fig. 24: Annotation Polar

Do not proceed unless you have already read *Basic annotation*, *text()* and *annotate()*!

Advanced Annotations

Annotating with Text with Box

Let's start with a simple example.

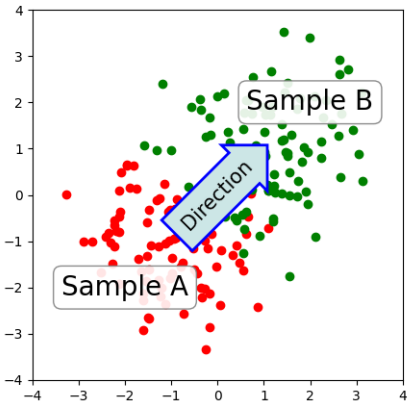


Fig. 25: Annotate Text Arrow

text takes a *bbox* keyword argument, which draws a box around the text:

```
t = ax.text(
    0, 0, "Direction", ha="center", va="center", rotation=45, size=15,
    bbox=dict(boxstyle="rarrow,pad=0.3", fc="cyan", ec="b", lw=2))
```

The patch object associated with the text can be accessed by:

```
bb = t.get_bbox_patch()
```

The return value is a *FancyBboxPatch*; patch properties (facecolor, edgewidth, etc.) can be accessed and modified as usual. *FancyBboxPatch.set_boxstyle* sets the box shape:

```
bb.set_boxstyle("rarrow", pad=0.6)
```

The arguments are the name of the box style with its attributes as keyword arguments. Currently, following box styles are implemented.

Class	Name	Attrs
Circle	`circle`	pad=0.3
DArrow	`darrow`	pad=0.3
LArrow	`larrow`	pad=0.3
RArrow	`rarrow`	pad=0.3
Round	`round`	pad=0.3,rounding_size=None
Round4	`round4`	pad=0.3,rounding_size=None
Roundtooth	`roundtooth`	pad=0.3,tooth_size=None
Sawtooth	`sawtooth`	pad=0.3,tooth_size=None
Square	`square`	pad=0.3

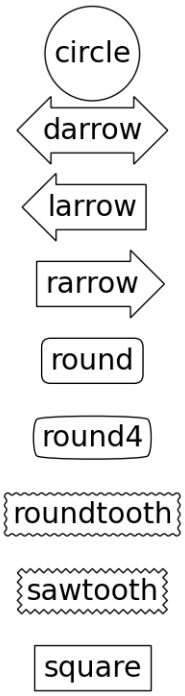


Fig. 26: Fancybox Demo

Note that the attribute arguments can be specified within the style name with separating comma (this form can be used as "boxstyle" value of bbox argument when initializing the text instance)

```
bb.set_boxstyle("rarrow,pad=0.6")
```

Annotating with Arrow

annotate draws an arrow connecting two points in an axes:

```
ax.annotate("Annotation",
            xy=(x1, y1), xycoords='data',
            xytext=(x2, y2), textcoords='offset points',
            )
```

This annotates a point at *xy* in the given coordinate (*xycoords*) with the text at *xytext* given in *textcoords*. Often, the annotated point is specified in the *data* coordinate and the annotating text in *offset points*. See *annotate* for available coordinate systems.

An arrow connecting *xy* to *xytext* can be optionally drawn by specifying the *arrowprops* argument. To draw only an arrow, use empty string as the first argument.

```
ax.annotate("",
            xy=(0.2, 0.2), xycoords='data',
            xytext=(0.8, 0.8), textcoords='data',
            arrowprops=dict(arrowstyle="->",
                            connectionstyle="arc3"),
            )
```

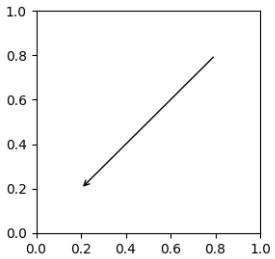


Fig. 27: Annotate Simple01

The arrow is drawn as follows:

1. A path connecting the two points is created, as specified by the *connectionstyle* parameter.
2. The path is clipped to avoid patches *patchA* and *patchB*, if these are set.
3. The path is further shrunk by *shrinkA* and *shrinkB* (in pixels).
4. The path is transmuted to an arrow patch, as specified by the *arrowstyle* parameter.

The creation of the connecting path between two points is controlled by `connectionstyle` key and the following styles are available.

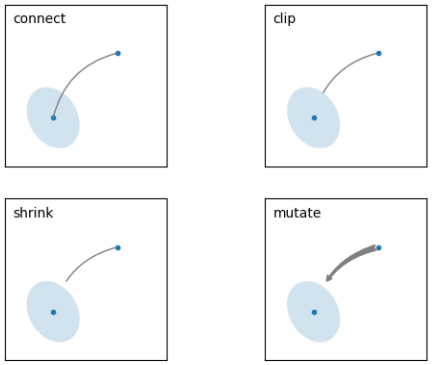


Fig. 28: Annotate Explain

Name	Attrs
`angle`	angleA=90,angleB=0,rad=0.0
`angle3`	angleA=90,angleB=0
`arc`	angleA=0,angleB=0,armA=None,armB=None,rad=0.0
`arc3`	rad=0.0
`bar`	armA=0.0,armB=0.0,fraction=0.3,angle=None

Note that "3" in `angle3` and `arc3` is meant to indicate that the resulting path is a quadratic spline segment (three control points). As will be discussed below, some arrow style options can only be used when the connecting path is a quadratic spline.

The behavior of each connection style is (limitedly) demonstrated in the example below. (Warning: The behavior of the `bar` style is currently not well defined, it may be changed in the future).

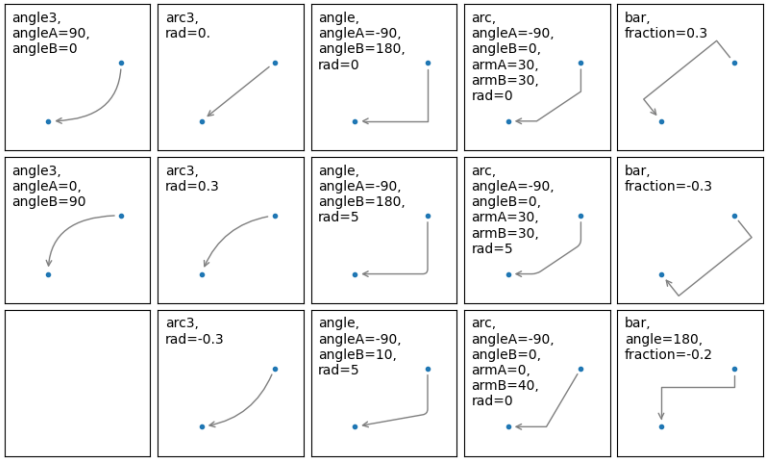


Fig. 29: Connectionstyle Demo

The connecting path (after clipping and shrinking) is then mutated to an arrow patch,

according to the given `arrowstyle`.

Name	Attrs
`-`	None
`->`	head_length=0.4,head_width=0.2
`-[`	widthB=1.0,lengthB=0.2,angleB=None
`|-|`	widthA=1.0,widthB=1.0
`-|>`	head_length=0.4,head_width=0.2
`<-`	head_length=0.4,head_width=0.2
`<->`	head_length=0.4,head_width=0.2
`<|-`	head_length=0.4,head_width=0.2
`<|-|>`	head_length=0.4,head_width=0.2
`fancy`	head_length=0.4,head_width=0.4,tail_width=0.4
`simple`	head_length=0.5,head_width=0.5,tail_width=0.2
`wedge`	tail_width=0.3,shrink_factor=0.5

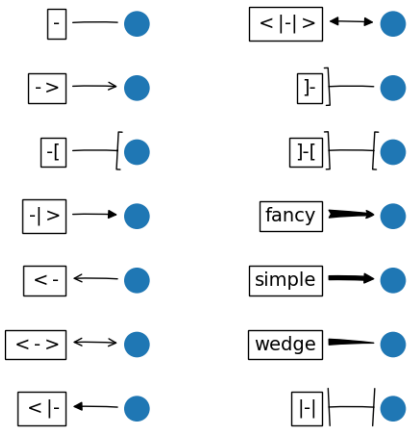


Fig. 30: Fancyarrow Demo

Some arrowstyles only work with connection styles that generate a quadratic-spline segment. They are `fancy`, `simple`, and `wedge`. For these arrow styles, you must use the "angle3" or "arc3" connection style.

If the annotation string is given, the patchA is set to the bbox patch of the text by default.

As with *text*, a box around the text can be drawn using the *bbox* argument.

By default, the starting point is set to the center of the text extent. This can be adjusted with `relpos` key value. The values are normalized to the extent of the text. For example, (0, 0) means lower-left corner and (1, 1) means top-right.

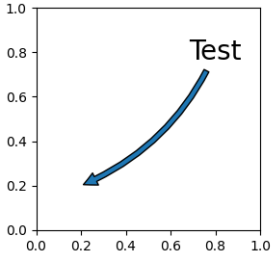


Fig. 31: Annotate Simple02

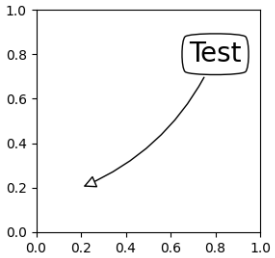


Fig. 32: Annotate Simple03

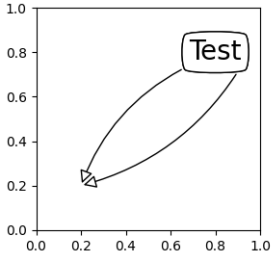


Fig. 33: Annotate Simple04

Placing Artist at the anchored location of the Axes

There are classes of artists that can be placed at an anchored location in the Axes. A common example is the legend. This type of artist can be created by using the *OffsetBox* class. A few predefined classes are available in *matplotlib.offsetbox* and in *mpl_toolkits.axes_grid1.anchored_artists*.

```
from matplotlib.offsetbox import AnchoredText
at = AnchoredText("Figure 1a",
                  prop=dict(size=15), frameon=True,
                  loc='upper left',
                  )
at.patch.set_boxstyle("round,pad=0.,rounding_size=0.2")
ax.add_artist(at)
```

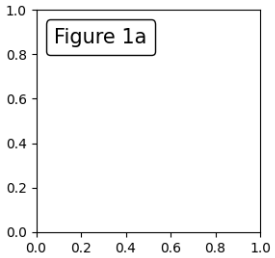


Fig. 34: Anchored Box01

The *loc* keyword has same meaning as in the legend command.

A simple application is when the size of the artist (or collection of artists) is known in pixel size during the time of creation. For example, If you want to draw a circle with fixed size of 20 pixel x 20 pixel (radius = 10 pixel), you can utilize `AnchoredDrawingArea`. The instance is created with a size of the drawing area (in pixels), and arbitrary artists can added to the drawing area. Note that the extents of the artists that are added to the drawing area are not related to the placement of the drawing area itself. Only the initial size matters.

```
from mpl_toolkits.axes_grid1.anchored_artists import AnchoredDrawingArea

ada = AnchoredDrawingArea(20, 20, 0, 0,
                          loc='upper right', pad=0., frameon=False)
p1 = Circle((10, 10), 10)
ada.drawing_area.add_artist(p1)
p2 = Circle((30, 10), 5, fc="r")
ada.drawing_area.add_artist(p2)
```

The artists that are added to the drawing area should not have a transform set (it will be overridden) and the dimensions of those artists are interpreted as a pixel coordinate, i.e., the radius of the circles in above example are 10 pixels and 5 pixels, respectively.

Sometimes, you want your artists to scale with the data coordinate (or coordinates

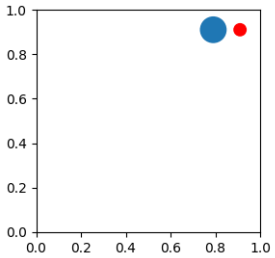


Fig. 35: Anchored Box02

other than canvas pixels). You can use `AnchoredAuxTransformBox` class. This is similar to `AnchoredDrawingArea` except that the extent of the artist is determined during the drawing time respecting the specified transform.

```
from mpl_toolkits.axes_grid1.anchored_artists import AnchoredAuxTransformBox

box = AnchoredAuxTransformBox(ax.transData, loc='upper left')
el = Ellipse((0, 0), width=0.1, height=0.4, angle=30)  # in data coordinates!
box.drawing_area.add_artist(el)
```

The ellipse in the above example will have width and height corresponding to 0.1 and 0.4 in data coordinates and will be automatically scaled when the view limits of the axes change.

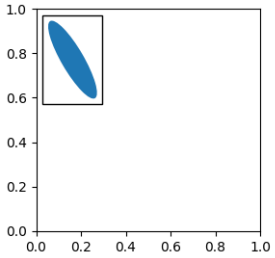


Fig. 36: Anchored Box03

As in the legend, the bbox_to_anchor argument can be set. Using the HPacker and VPacker, you can have an arrangement(?) of artist as in the legend (as a matter of fact, this is how the legend is created).

Note that unlike the legend, the `bbox_transform` is set to IdentityTransform by default.

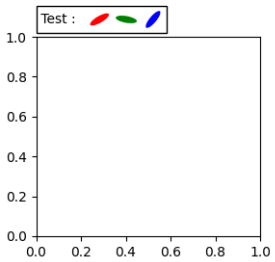


Fig. 37: Anchored Box04

Using Complex Coordinates with Annotations

The Annotation in matplotlib supports several types of coordinates as described in *Basic annotation*. For an advanced user who wants more control, it supports a few other options.

1. A *Transform* instance. For example,

```
ax.annotate("Test", xy=(0.5, 0.5), xycoords=ax.transAxes)
```

   is identical to

```
ax.annotate("Test", xy=(0.5, 0.5), xycoords="axes fraction")
```

   This allows annotating a point in another axes:

```
ax1, ax2 = subplot(121), subplot(122)
ax2.annotate("Test", xy=(0.5, 0.5), xycoords=ax1.transData,
             xytext=(0.5, 0.5), textcoords=ax2.transData,
             arrowprops=dict(arrowstyle="->"))
```

2. An *Artist* instance. The *xy* value (or *xytext*) is interpreted as a fractional coordinate of the bbox (return value of *get_window_extent*) of the artist:

```
an1 = ax.annotate("Test 1", xy=(0.5, 0.5), xycoords="data",
                  va="center", ha="center",
                  bbox=dict(boxstyle="round", fc="w"))
an2 = ax.annotate("Test 2", xy=(1, 0.5), xycoords=an1,  # (1, 0.5) of the an1's bbox
                  xytext=(30, 0), textcoords="offset points",
                  va="center", ha="left",
                  bbox=dict(boxstyle="round", fc="w"),
                  arrowprops=dict(arrowstyle="->"))
```

   Note that you must ensure that the extent of the coordinate artist (*an1* in above example) is determined before *an2* gets drawn. Usually, this means that *an2* needs to be drawn after *an1*.

3. A callable object that takes the renderer instance as single argument, and returns either a *Transform* or a *BboxBase*. The return value is then handled as in (1), for transforms, or in (2), for bboxes. For example,

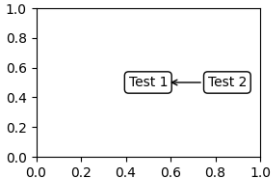


Fig. 38: Annotation with Simple Coordinates

```
an2 = ax.annotate("Test 2", xy=(1, 0.5), xycoords=an1,
                  xytext=(30, 0), textcoords="offset points")
```

is identical to:

```
an2 = ax.annotate("Test 2", xy=(1, 0.5), xycoords=an1.get_window_extent,
                  xytext=(30, 0), textcoords="offset points")
```

4. A pair of coordinate specifications -- the first for the x-coordinate, and the second is for the y-coordinate; e.g.

```
annotate("Test", xy=(0.5, 1), xycoords=("data", "axes fraction"))
```

Here, 0.5 is in data coordinates, and 1 is in normalized axes coordinates. Each of the coordinate specifications can also be an artist or a transform. For example,

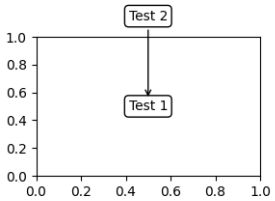


Fig. 39: Annotation with Simple Coordinates 2

5. Sometimes, you want your annotation with some "offset points", not from the annotated point but from some other point. *text.OffsetFrom* is a helper for such cases.

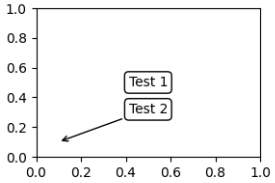


Fig. 40: Annotation with Simple Coordinates 3

You may take a look at this example /gallery/text_labels_and_annotations/annotation_demo.

Using ConnectionPatch

ConnectionPatch is like an annotation without text. While *annotate* is sufficient in most situations, ConnectionPatch is useful when you want to connect points in different axes.

```
from matplotlib.patches import ConnectionPatch
xy = (0.2, 0.2)
con = ConnectionPatch(xyA=xy, coordsA=ax1.transData,
                      xyB=xy, coordsB=ax2.transData)
fig.add_artist(con)
```

The above code connects point *xy* in the data coordinates of `ax1` to point *xy* in the data coordinates of `ax2`. Here is a simple example.

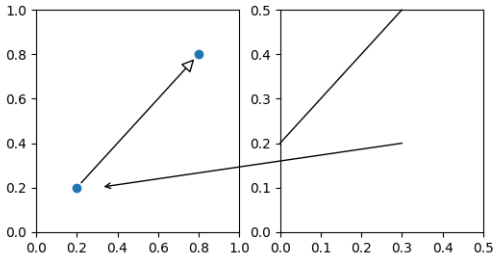


Fig. 41: Connect Simple01

Here, we added the ConnectionPatch to the *figure* (with *add_artist*) rather than to either axes: this ensures that it is drawn on top of both axes, and is also necessary if using *constrained_layout* for positioning the axes.

Advanced Topics

Zoom effect between Axes

`mpl_toolkits.axes_grid1.inset_locator` defines some patch classes useful for interconnecting two axes. Understanding the code requires some knowledge of Matplotlib's transform system.

Define Custom BoxStyle

You can use a custom box style. The value for the `boxstyle` can be a callable object in the following forms.:

```
def __call__(self, x0, y0, width, height, mutation_size,
             aspect_ratio=1.):
    '''
```

(continues on next page)

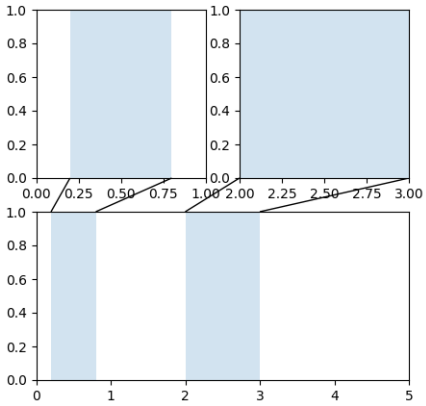


Fig. 42: Axes Zoom Effect

(continued from previous page)

```
    Given the location and size of the box, return the path of
    the box around it.

      - *x0*, *y0*, *width*, *height* : location and size of the box
      - *mutation_size* : a reference scale for the mutation.
      - *aspect_ratio* : aspect-ratio for the mutation.
    '''
    path = ...
    return path
```

Here is a complete example.

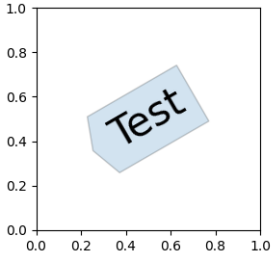


Fig. 43: Custom Boxstyle01

Similarly, you can define a custom ConnectionStyle and a custom ArrowStyle. See the source code of `lib/matplotlib/patches.py` and check how each style class is defined.

2.6.5 Writing mathematical expressions

An introduction to writing mathematical expressions in Matplotlib.

You can use a subset TeX markup in any matplotlib text string by placing it inside a pair of dollar signs ($).

Note that you do not need to have TeX installed, since Matplotlib ships its own TeX expression parser, layout engine, and fonts. The layout engine is a fairly direct adaptation of the layout algorithms in Donald Knuth's TeX, so the quality is quite good (matplotlib also provides a `usetex` option for those who do want to call out to TeX to generate their text (see *Text rendering With LaTeX*).

Any text element can use math text. You should use raw strings (precede the quotes with an `'r'`), and surround the math text with dollar signs ($), as in TeX. Regular text and mathtext can be interleaved within the same string. Mathtext can use DejaVu Sans (default), DejaVu Serif, the Computer Modern fonts (from (La)TeX), STIX fonts (with are designed to blend well with Times), or a Unicode font that you provide. The mathtext font can be selected with the customization variable `mathtext.fontset` (see *Customizing Matplotlib with style sheets and rcParams*)

Here is a simple example:

```
# plain text
plt.title('alpha > beta')
```

produces "alpha > beta".

Whereas this:

```
# math text
plt.title(r'$\alpha > \beta$')
```

produces "$\alpha > \beta$".

Note: Mathtext should be placed between a pair of dollar signs ($). To make it easy to display monetary values, e.g., "$100.00", if a single dollar sign is present in the entire string, it will be displayed verbatim as a dollar sign. This is a small change from regular TeX, where the dollar sign in non-math text would have to be escaped ('\$').

Note: While the syntax inside the pair of dollar signs ($) aims to be TeX-like, the text outside does not. In particular, characters such as:

```
# $ % & ~ _ ^ \ { } \( \) \[ \]
```

have special meaning outside of math mode in TeX. Therefore, these characters will behave differently depending on `rcParams["text.usetex"]` (default: `False`). See the

usetex tutorial for more information.

Subscripts and superscripts

To make subscripts and superscripts, use the `'_'` and `'^'` symbols:

```
r'$\alpha_i > \beta_i$'
```

$$\alpha_i > \beta_i$$

To display multi-letter subscripts or superscripts correctly, you should put them in curly braces `{...}`:

```
r'$\alpha^{ic} > \beta_{ic}$'
```

$$\alpha^{ic} > \beta_{ic}$$

Some symbols automatically put their sub/superscripts under and over the operator. For example, to write the sum of x_i from 0 to ∞, you could do:

```
r'$\sum_{i=0}^\infty x_i$'
```

$$\sum_{i=0}^{\infty} x_i$$

Fractions, binomials, and stacked numbers

Fractions, binomials, and stacked numbers can be created with the `\frac{}{}`, `\binom{}{}` and `\genfrac{}{}{}{}{}{}` commands, respectively:

```
r'$\frac{3}{4} \binom{3}{4} \genfrac{}{}{0}{}{3}{4}$'
```

produces

$$\frac{3}{4}\binom{3}{4}\genfrac{}{}{0}{}{3}{4}$$

Fractions can be arbitrarily nested:

```
r'$\frac{5 - \frac{1}{x}}{4}$'
```

produces

$$\frac{5 - \frac{1}{x}}{4}$$

Note that special care needs to be taken to place parentheses and brackets around fractions. Doing things the obvious way produces brackets that are too small:

```
r'$(\frac{5 - \frac{1}{x}}{4})$'
```

$$(\frac{5 - \frac{1}{x}}{4})$$

The solution is to precede the bracket with `\left` and `\right` to inform the parser that those brackets encompass the entire object.:

```
r'$\left(\frac{5 - \frac{1}{x}}{4}\right)$'
```

$$\left(\frac{5 - \frac{1}{x}}{4}\right)$$

Radicals

Radicals can be produced with the `\sqrt[]{}` command. For example:

```
r'$\sqrt{2}$'
```

$$\sqrt{2}$$

Any base can (optionally) be provided inside square brackets. Note that the base must be a simple expression, and can not contain layout commands such as fractions or sub/superscripts:

```
r'$\sqrt[3]{x}$'
```

$$\sqrt[3]{x}$$

Fonts

The default font is *italics* for mathematical symbols.

Note: This default can be changed using `rcParams["mathtext.default"]` (default: `'it'`). This is useful, for example, to use the same font as regular non-math text for math text, by setting it to `regular`.

To change fonts, e.g., to write "sin" in a Roman font, enclose the text in a font command:

```
r'$s(t) = \mathcal{A}\mathrm{sin}(2 \omega t)$'
```

$$s(t) = \mathcal{A}\mathrm{sin}(2\omega t)$$

More conveniently, many commonly used function names that are typeset in a Roman font have shortcuts. So the expression above could be written as follows:

```
r'$s(t) = \mathcal{A}\sin(2 \omega t)$'
```

$$s(t) = \mathcal{A}\sin(2\omega t)$$

Here "s" and "t" are variable in italics font (default), "sin" is in Roman font, and the amplitude "A" is in calligraphy font. Note in the example above the calligraphy `A` is squished into the `sin`. You can use a spacing command to add a little whitespace between them:

```
r's(t) = \mathcal{A}\/\sin(2 \omega t)'
```

$$s(t) = \mathcal{A}\,\sin(2\omega t)$$

The choices available with all fonts are:

Command	Result
`\mathrm{Roman}`	Roman
`\mathit{Italic}`	*Italic*
`\mathtt{Typewriter}`	`Typewriter`
`\mathcal{CALLIGRAPHY}`	$\mathcal{CALLIGRAPHY}$

When using the STIX fonts, you also have the choice of:

Command	Result
`\mathbb{blackboard}`	$\mathbb{blackboard}$
`\mathrm{\mathbb{blackboard}}`	$\mathrm{\mathbb{blackboard}}$
`\mathfrak{Fraktur}`	$\mathfrak{Fraktur}$
`\mathsf{sansserif}`	sansserif
`\mathrm{\mathsf{sansserif}}`	sansserif

There are also five global "font sets" to choose from, which are selected using the `mathtext.fontset` parameter in *matplotlibrc*.

`dejavusans`: DejaVu Sans

$$\mathcal{R}\prod_{i=\alpha}^{\infty} a_i \sin\left(2\pi f x_i\right) \tag{2.1}$$

`dejavuserif`: DejaVu Serif

$$\mathcal{R}\prod_{i=\alpha}^{\infty} a_i \sin\left(2\pi f x_i\right) \tag{2.2}$$

`cm`: Computer Modern (TeX)

$$\mathcal{R}\prod_{i=\alpha}^{\infty} a_i \sin\left(2\pi f x_i\right) \tag{2.3}$$

`stix`: STIX (designed to blend well with Times)

$$\mathcal{R}\prod_{i=\alpha}^{\infty} a_i \sin\left(2\pi f x_i\right) \tag{2.4}$$

`stixsans`: STIX sans-serif

$$\mathcal{R}\prod_{i=\alpha}^{\infty} a_i \sin\left(2\pi f x_i\right) \tag{2.5}$$

Additionally, you can use `\mathdefault{...}` or its alias `\mathregular{...}` to use the font used for regular text outside of mathtext. There are a number of limitations to this approach, most notably that far fewer symbols will be available, but it can be useful to make math expressions blend well with other text in the plot.

Custom fonts

mathtext also provides a way to use custom fonts for math. This method is fairly tricky to use, and should be considered an experimental feature for patient users only. By setting `rcParams["mathtext.fontset"]` (default: `'dejavusans'`) to `custom`, you can then set the following parameters, which control which font file to use for a particular set of math characters.

Parameter	Corresponds to
`mathtext.it`	`\mathit{}` or default italic
`mathtext.rm`	`\mathrm{}` Roman (upright)
`mathtext.tt`	`\mathtt{}` Typewriter (monospace)
`mathtext.bf`	`\mathbf{}` bold italic
`mathtext.cal`	`\mathcal{}` calligraphic
`mathtext.sf`	`\mathsf{}` sans-serif

Each parameter should be set to a fontconfig font descriptor (as defined in the yet-to-be-written font chapter).

The fonts used should have a Unicode mapping in order to find any non-Latin characters, such as Greek. If you want to use a math symbol that is not contained in your custom fonts, you can set `rcParams["mathtext.fallback"]` (default: `'cm'`) to either `'cm'`, `'stix'` or `'stixsans'` which will cause the mathtext system to use characters from an alternative font whenever a particular character can not be found in the custom font.

Note that the math glyphs specified in Unicode have evolved over time, and many fonts may not have glyphs in the correct place for mathtext.

Accents

An accent command may precede any symbol to add an accent above it. There are long and short forms for some of them.

Command	Result
`\acute a` or `\'a`	$\acute{a}$
`\bar a`	$\bar{a}$
`\breve a`	$\breve{a}$
`\ddot a` or `\''a`	$\ddot{a}$
`\dot a` or `\.a`	$\dot{a}$
`\grave a` or `\`a`	$\grave{a}$
`\hat a` or `\^a`	$\hat{a}$
`\tilde a` or `\~a`	$\tilde{a}$
`\vec a`	$\vec{a}$
`\overline{abc}`	$\overline{abc}$

In addition, there are two special accents that automatically adjust to the width of the symbols below:

Command	Result
`\widehat{xyz}`	$\widehat{xyz}$
`\widetilde{xyz}`	$\widetilde{xyz}$

Care should be taken when putting accents on lower-case i's and j's. Note that in the following `\imath` is used to avoid the extra dot over the i:

```
r"$\hat i\ \ \hat \imath$"
```

$\hat{i}\ \hat{\imath}$

Symbols

You can also use a large number of the TeX symbols, as in `\infty`, `\leftarrow`, `\sum`, `\int`.

Lower-case Greek

α \alpha	β \beta	χ \chi	δ \delta	ƒ \digamma	ε \epsilon
η \eta	γ \gamma	ι \iota	κ \kappa	λ \lambda	μ \mu
ν \nu	ω \omega	φ \phi	π \pi	ψ \psi	ρ \rho
σ \sigma	τ \tau	θ \theta	υ \upsilon	ε \varepsilon	ϰ \varkappa
φ \varphi	ϖ \varpi	ϱ \varrho	ς \varsigma	ϑ \vartheta	ξ \xi
ζ \zeta					

Upper-case Greek

Δ \Delta	Γ \Gamma	Λ \Lambda	Ω \Omega	Φ \Phi	Π \Pi	Ψ \Psi	Σ \Sigma
Θ \Theta	Υ \Upsilon	Ξ \Xi	℧ \mho	∇ \nabla			

Hebrew

ℵ \aleph	ℶ \beth	ℸ \daleth	ℷ \gimel		

Delimiters

/ /	[[	⇓ \Downarrow	⇑ \Uparrow	‖ \Vert	\backslash
↓ \downarrow	⟨ \langle	⌈ \lceil	⌊ \lfloor	⌞ \llcorner	⌟ \lrcorner
⟩ \rangle	⌉ \rceil	⌋ \rfloor	⌜ \ulcorner	↑ \uparrow	⌜ \urcorner
\vert	{ \{	|	} \}	]]	|

Big symbols

⋂ \bigcap	⋃ \bigcup	⨃ \bigodot	⨁ \bigoplus	⨂ \bigotimes	⨄ \biguplus
⋁ \bigvee	⋀ \bigwedge	∐ \coprod	∫ \int	∮ \oint	∏ \prod
∑ \sum					

Standard function names

Pr \Pr	arccos \arccos	arcsin \arcsin	arctan \arctan	arg \arg	cos \cos
cosh \cosh	cot \cot	coth \coth	csc \csc	deg \deg	det \det
dim \dim	exp \exp	gcd \gcd	hom \hom	inf \inf	ker \ker
lg \lg	lim \lim	liminf \liminf	limsup \limsup	ln \ln	log \log
max \max	min \min	sec \sec	sin \sin	sinh \sinh	sup \sup
tan \tan	tanh \tanh				

Binary operation and relation symbols

□ \Bumpeq	□ \Cap	□ \Cup	≑ \Doteq
□ \Join	□ \Subset	□ \Supset	⊩ \Vdash
⊪ \Vvdash	≈ \approx	□ \approxeq	∗ \ast
□ \asymp	϶ \backepsilon	∽ \backsim	□ \backsimeq
□ \barwedge	□ \because	□ \between	○ \bigcirc
▽ \bigtriangledown	△ \bigtriangleup	◀ \blacktriangleleft	▶ \blacktriangleright
⊥ \bot	□ \bowtie	⊡ \boxdot	⊟ \boxminus
⊞ \boxplus	⊠ \boxtimes	• \bullet	□ \bumpeq
∩ \cap	· \cdot	∘ \circ	□ \circeq
≔ \coloneq	□ \cong	∪ \cup	□ \curlyeqprec
□ \curlyeqsucc	□ \curlyvee	□ \curlywedge	† \dag
⊣ \dashv	‡ \ddag	⋄ \diamond	÷ \div
□ \divideontimes	≐ \doteq	≑ \doteqdot	∔ \dotplus
□ \doublebarwedge	□ \eqcirc	≕ \eqcolon	≂ \eqsim
□ \eqslantgtr	□ \eqslantless	≡ \equiv	≒ \fallingdotseq
□ \frown	≥ \geq	□ \geqq	□ \geqslant
□ \gg	□ \ggg	□ \gnapprox	□ \gneqq
□ \gnsim	□ \gtrapprox	□ \gtrdot	□ \gtreqless
□ \gtreqqless	□ \gtrless	□ \gtrsim	∈ \in
□ \intercal	□ \leftthreetimes	≤ \leq	□ \leqq
□ \leqslant	□ \lessapprox	□ \lessdot	□ \lesseqgtr
□ \lesseqqgtr	□ \lessgtr	□ \lesssim	□ \ll
□ \lll	□ \lnapprox	□ \lneqq	□ \lnsim
□ \ltimes	∣ \mid	⊨ \models	∓ \mp
⊮ \nVDash	⊯ \nVdash	≉ \napprox	□ \ncong
≠ \ne	≠ \neq	≠ \neq	≢ \nequiv
≱ \ngeq	≯ \ngtr	∋ \ni	≰ \nleq
≮ \nless	∤ \nmid	∉ \notin	∦ \nparallel
□ \nprec	≁ \nsim	⊄ \nsubset	⊈ \nsubseteq
□ \nsucc	⊅ \nsupset	⊉ \nsupseteq	□ \ntriangleleft

continues on next page

Table 2 – continued from previous page

□ \ntrianglelefteq	□ \ntriangleright	□ \ntrianglerighteq	⊯ \nvDash
⊮ \nvdash	⊙ \odot	⊖ \ominus	⊕ \oplus
⊘ \oslash	⊗ \otimes	∥ \parallel	□ \perp
□ \pitchfork	± \pm	□ \prec	□ \precapprox
□ \preccurlyeq	□ \preceq	□ \precnapprox	□ \precnsim
□ \precsim	∝ \propto	□ \rightthreetimes	≓ \risingdotseq
□ \rtimes	∼ \sim	≃ \simeq	/ \slash
□ \smile	⊓ \sqcap	⊔ \sqcup	⊏ \sqsubset
⊏ \sqsubset	⊑ \sqsubseteq	⊐ \sqsupset	⊐ \sqsupset
⊒ \sqsupseteq	□ \star	⊂ \subset	⊆ \subseteq
□ \subseteqq	□ \subsetneq	□ \subsetneqq	□ \succ
□ \succapprox	□ \succcurlyeq	□ \succeq	□ \succnapprox
□ \succnsim	□ \succsim	⊃ \supset	⊇ \supseteq
□ \supseteqq	□ \supsetneq	□ \supsetneqq	□ \therefore
× \times	⊤ \top	◁ \triangleleft	□ \trianglelefteq
□ \triangleq	▷ \triangleright	□ \trianglerighteq	⊎ \uplus
⊨ \vDash	∝ \varpropto	□ \vartriangleleft	□ \vartriangleright
⊢ \vdash	∨ \vee	□ \veebar	∧ \wedge
□ \wr			

Arrow symbols

⇓ \Downarrow	⇐ \Leftarrow	⇔ \Leftrightarrow	⇚ \Lleftarrow
⟸ \Longleftarrow	⟺ \Longleftrightarrow	⟹ \Longrightarrow	↰ \Lsh
⇗ \Nearrow	⇖ \Nwarrow	⇒ \Rightarrow	⇛ \Rrightarrow
↱ \Rsh	⇘ \Searrow	⇙ \Swarrow	⇑ \Uparrow
⇕ \Updownarrow	↺ \circlearrowleft	↻ \circlearrowright	↶ \curvearrowleft
↷ \curvearrowright	⇠ \dashleftarrow	⇢ \dashrightarrow	↓ \downarrow
⇊ \downdownarrows	⇃ \downharpoonleft	⇂ \downharpoonright	↩ \hookleftarrow
↪ \hookrightarrow	⇝ \leadsto	← \leftarrow	↢ \leftarrowtail
↼ \leftharpoondown	↼ \leftharpoonup	⇆ \leftleftarrows	↔ \leftrightarrow
⇆ \leftrightarrows	⇋ \leftrightharpoons	↭ \leftrightsquigarrow	↫ \leftsquigarrow
⟵ \longleftarrow	⟷ \longleftrightarrow	⟼ \longmapsto	⟶ \longrightarrow
↫ \looparrowleft	↬ \looparrowright	↦ \mapsto	⊸ \multimap
⇍ \nLeftarrow	⇎ \nLeftrightarrow	⇏ \nRightarrow	↗ \nearrow
↚ \nleftarrow	↮ \nleftrightarrow	↛ \nrightarrow	↖ \nwarrow
→ \rightarrow	↣ \rightarrowtail	⇁ \rightharpoondown	⇀ \rightharpoonup
⇄ \rightleftarrows	⇄ \rightleftarrows	⇌ \rightleftharpoons	⇌ \rightleftharpoons
⇉ \rightrightarrows	⇉ \rightrightarrows	⇝ \rightsquigarrow	↘ \searrow
↙ \swarrow	→ \to	↞ \twoheadleftarrow	↠ \twoheadrightarrow
↑ \uparrow	↕ \updownarrow	↕ \updownarrow	↿ \upharpoonleft
↾ \upharpoonright	⇈ \upuparrows		

Miscellaneous symbols

$ \$	Å \AA	ⅎ \Finv	⅁ \Game
ℑ \Im	¶ \P	ℜ \Re	§ \S
∠ \angle	` \backprime	⋆ \bigstar	■ \blacksquare
▲ \blacktriangle	▼ \blacktriangledown	⋯ \cdots	✓ \checkmark
® \circledR	Ⓢ \circledS	♣ \clubsuit	∁ \complement
© \copyright	⋱ \ddots	♢ \diamondsuit	ℓ \ell
∅ \emptyset	ð \eth	∃ \exists	♭ \flat
∀ \forall	ℏ \hbar	♡ \heartsuit	ℏ \hslash
∭ \iiint	∬ \iint	ɪ \imath	∞ \infty
ȷ \jmath	… \ldots	∡ \measuredangle	♮ \natural
¬ \neg	∄ \nexists	∰ \oiiint	∂ \partial
′ \prime	♯ \sharp	♠ \spadesuit	∢ \sphericalangle
ß \ss	▽ \triangledown	∅ \varnothing	△ \vartriangle
⋮ \vdots	℘ \wp	¥ \yen	

If a particular symbol does not have a name (as is true of many of the more obscure symbols in the STIX fonts), Unicode characters can also be used:

```
r'$\u23ce$'
```

Example

Here is an example illustrating many of these features in context.

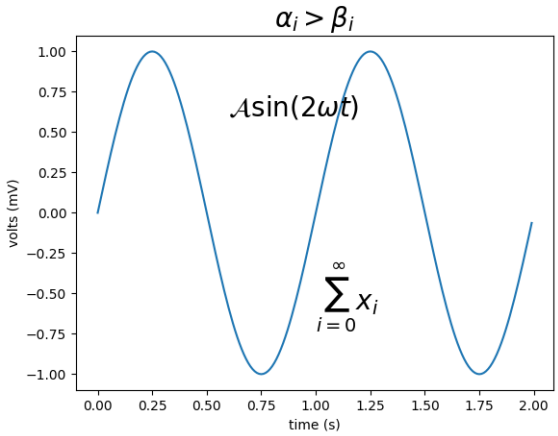


Fig. 44: Pyplot Mathtext

2.6.6 Typesetting With XeLaTeX/LuaLaTeX

How to typeset text with the `pgf` backend in Matplotlib.

Using the `pgf` backend, Matplotlib can export figures as pgf drawing commands that can be processed with pdflatex, xelatex or lualatex. XeLaTeX and LuaLaTeX have full Unicode support and can use any font that is installed in the operating system, making use of advanced typographic features of OpenType, AAT and Graphite. Pgf pictures created by `plt.savefig('figure.pgf')` can be embedded as raw commands in LaTeX documents. Figures can also be directly compiled and saved to PDF with `plt.savefig('figure.pdf')` by switching the backend

```
matplotlib.use('pgf')
```

or by explicitly requesting the use of the `pgf` backend

```
plt.savefig('figure.pdf', backend='pgf')
```

or by registering it for handling pdf output

```
from matplotlib.backends.backend_pgf import FigureCanvasPgf
matplotlib.backend_bases.register_backend('pdf', FigureCanvasPgf)
```

The last method allows you to keep using regular interactive backends and to save xelatex, lualatex or pdflatex compiled PDF files from the graphical user interface.

Matplotlib's pgf support requires a recent LaTeX installation that includes the TikZ/PGF packages (such as TeXLive), preferably with XeLaTeX or LuaLaTeX installed. If either pdftocairo or ghostscript is present on your system, figures can optionally be saved to PNG images as well. The executables for all applications must be located on your *PATH*.

rcParams that control the behavior of the pgf backend:

Parameter	Documentation
pgf.preamble	Lines to be included in the LaTeX preamble
pgf.rcfonts	Setup fonts from rc params using the fontspec package
pgf.texsystem	Either "xelatex" (default), "lualatex" or "pdflatex"

Note: TeX defines a set of special characters, such as:

```
# $ % & ~ _ ^ \ { }
```

Generally, these characters must be escaped correctly. For convenience, some characters (_, ^, %) are automatically escaped outside of math environments.

Multi-Page PDF Files

The pgf backend also supports multipage pdf files using *PdfPages*

```
from matplotlib.backends.backend_pgf import PdfPages
import matplotlib.pyplot as plt

with PdfPages('multipage.pdf', metadata={'author': 'Me'}) as pdf:

    fig1, ax1 = plt.subplots()
    ax1.plot([1, 5, 3])
    pdf.savefig(fig1)

    fig2, ax2 = plt.subplots()
    ax2.plot([1, 5, 3])
    pdf.savefig(fig2)
```

Font specification

The fonts used for obtaining the size of text elements or when compiling figures to PDF are usually defined in the *rcParams*. You can also use the LaTeX default Computer Modern fonts by clearing the lists for `rcParams["font.serif"]` (default: `['DejaVu Serif', 'Bitstream Vera Serif', 'Computer Modern Roman', 'New Century Schoolbook', 'Century Schoolbook L', 'Utopia', 'ITC Bookman', 'Bookman', 'Nimbus Roman No9 L', 'Times New Roman', 'Times', 'Palatino', 'Charter', 'serif']`), `rcParams["font.sans-serif"]` (default: `['DejaVu Sans', 'Bitstream Vera Sans', 'Computer Modern Sans Serif', 'Lucida Grande', 'Verdana', 'Geneva', 'Lucid', 'Arial', 'Helvetica', 'Avant Garde', 'sans-serif']`) or `rcParams["font.monospace"]` (default: `['DejaVu Sans Mono', 'Bitstream Vera Sans Mono', 'Computer Modern Typewriter', 'Andale Mono', 'Nimbus Mono L', 'Courier New', 'Courier', 'Fixed', 'Terminal', 'monospace']`). Please note that the glyph coverage of these fonts is very limited. If you want to keep the Computer Modern font face but require extended Unicode support, consider installing the Computer Modern Unicode fonts *CMU Serif*, *CMU Sans Serif*, etc.

When saving to `.pgf`, the font configuration Matplotlib used for the layout of the figure is included in the header of the text file.

```
"""
*********
Pgf Fonts
*********

"""

import matplotlib.pyplot as plt
plt.rcParams.update({
    "font.family": "serif",
    "font.serif": [],                   # use latex default serif font
```

(continues on next page)

(continued from previous page)

```
    "font.sans-serif": ["DejaVu Sans"],  # use a specific sans-serif font
})

plt.figure(figsize=(4.5, 2.5))
plt.plot(range(5))
plt.text(0.5, 3., "serif")
plt.text(0.5, 2., "monospace", family="monospace")
plt.text(2.5, 2., "sans-serif", family="sans-serif")
plt.text(2.5, 1., "comic sans", family="Comic Sans MS")
plt.xlabel("µ is not $\\mu$")
plt.tight_layout(.5)
```

Custom preamble

Full customization is possible by adding your own commands to the preamble. Use `rcParams["pgf.preamble"]` (default: `''`) if you want to configure the math fonts, using `unicode-math` for example, or for loading additional packages. Also, if you want to do the font configuration yourself instead of using the fonts specified in the rc parameters, make sure to disable `rcParams["pgf.rcfonts"]` (default: `True`).

```
"""
============
Pgf Preamble
============

"""

import matplotlib as mpl
mpl.use("pgf")
```

Choosing the TeX system

The TeX system to be used by Matplotlib is chosen by `rcParams["pgf.texsystem"]` (default: `'xelatex'`). Possible values are `'xelatex'` (default), `'lualatex'` and `'pdflatex'`. Please note that when selecting pdflatex, the fonts and Unicode handling must be configured in the preamble.

```
"""
=============
Pgf Texsystem
=============

"""

import matplotlib.pyplot as plt
```

(continues on next page)

(continued from previous page)

```
plt.rcParams.update({
    "pgf.texsystem": "pdflatex",
    "pgf.preamble": "\n".join([
         r"\usepackage[utf8x]{inputenc}",
         r"\usepackage[T1]{fontenc}",
         r"\usepackage{cmbright}",
    ]),
})

plt.figure(figsize=(4.5, 2.5))
plt.plot(range(5))
plt.text(0.5, 3., "serif", family="serif")
plt.text(0.5, 2., "monospace", family="monospace")
plt.text(2.5, 2., "sans-serif", family="sans-serif")
plt.xlabel(r"µ is not $\mu$")
plt.tight_layout(.5)
```

Troubleshooting

- Please note that the TeX packages found in some Linux distributions and MiKTeX installations are dramatically outdated. Make sure to update your package catalog and upgrade or install a recent TeX distribution.
- On Windows, the *PATH* environment variable may need to be modified to include the directories containing the latex, dvipng and ghostscript executables. See *Environment Variables* and *Setting environment variables in Windows* for details.
- A limitation on Windows causes the backend to keep file handles that have been opened by your application open. As a result, it may not be possible to delete the corresponding files until the application closes (see #1324).
- Sometimes the font rendering in figures that are saved to png images is very bad. This happens when the pdftocairo tool is not available and ghostscript is used for the pdf to png conversion.
- Make sure what you are trying to do is possible in a LaTeX document, that your LaTeX syntax is valid and that you are using raw strings if necessary to avoid unintended escape sequences.
- `rcParams["pgf.preamble"]` (default: `''`) provides lots of flexibility, and lots of ways to cause problems. When experiencing problems, try to minimalize or disable the custom preamble.
- Configuring an `unicode-math` environment can be a bit tricky. The TeXLive distribution for example provides a set of math fonts which are usually not installed system-wide. XeTeX, unlike LuaLatex, cannot find these fonts by their name, which is why you might have to specify `\setmathfont{xits-math.otf}` instead of `\setmathfont{XITS Math}` or alternatively make the fonts available to your OS. See this tex.stackexchange.com question for more details.

- If the font configuration used by Matplotlib differs from the font setting in yout LaTeX document, the alignment of text elements in imported figures may be off. Check the header of your `.pgf` file if you are unsure about the fonts Matplotlib used for the layout.
- Vector images and hence `.pgf` files can become bloated if there are a lot of objects in the graph. This can be the case for image processing or very big scatter graphs. In an extreme case this can cause TeX to run out of memory: "TeX capacity exceeded, sorry" You can configure latex to increase the amount of memory available to generate the `.pdf` image as discussed on tex.stackexchange.com. Another way would be to "rasterize" parts of the graph causing problems using either the `rasterized=True` keyword, or `.set_rasterized(True)` as per this example.
- If you still need help, please see *Getting help*

2.6.7 Text rendering With LaTeX

Rendering text with LaTeX in Matplotlib.

Matplotlib has the option to use LaTeX to manage all text layout. This option is available with the following backends:

- Agg
- PS
- PDF

The LaTeX option is activated by setting `text.usetex : True` in your rc settings. Text handling with matplotlib's LaTeX support is slower than matplotlib's very capable *mathtext*, but is more flexible, since different LaTeX packages (font packages, math packages, etc.) can be used. The results can be striking, especially when you take care to use the same fonts in your figures as in the main document.

Matplotlib's LaTeX support requires a working LaTeX installation, dvipng (which may be included with your LaTeX installation), and Ghostscript (GPL Ghostscript 9.0 or later is required). The executables for these external dependencies must all be located on your *PATH*.

There are a couple of options to mention, which can be changed using *rc settings*. Here is an example matplotlibrc file:

```
font.family        : serif
font.serif         : Times, Palatino, New Century Schoolbook, Bookman, Computer Modern␣
↪Roman
font.sans-serif    : Helvetica, Avant Garde, Computer Modern Sans serif
font.cursive       : Zapf Chancery
font.monospace     : Courier, Computer Modern Typewriter

text.usetex        : true
```

The first valid font in each family is the one that will be loaded. If the fonts are not specified, the Computer Modern fonts are used by default. All of the other fonts are Adobe fonts. Times and Palatino each have their own accompanying math fonts, while the other Adobe serif fonts make use of the Computer Modern math fonts. See the PSNFSS documentation for more details.

To use LaTeX and select Helvetica as the default font, without editing matplotlibrc use:

```
import matplotlib as mpl
plt.rcParams.update({
    "text.usetex": True,
    "font.family": "sans-serif",
    "font.sans-serif": ["Helvetica"]})
## for Palatino and other serif fonts use:
plt.rcParams.update({
    "text.usetex": True,
    "font.family": "serif",
    "font.serif": ["Palatino"],
})
```

Here is the standard example, `/gallery/text_labels_and_annotations/tex_demo`:

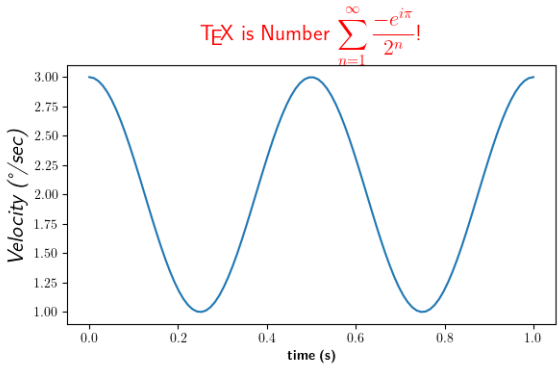


Fig. 45: TeX Demo

Note that display math mode (`$$ e=mc^2 $$`) is not supported, but adding the command `\displaystyle`, as in the above demo, will produce the same results.

Note: Certain characters require special escaping in TeX, such as:

```
# $ % & ~ _ ^ \ { } \( \) \[ \]
```

Therefore, these characters will behave differently depending on `rcParams["text.usetex"]` (default: `False`).

---

usetex with unicode

It is also possible to use unicode strings with the LaTeX text manager, here is an example taken from `/gallery/text_labels_and_annotations/tex_demo`. The axis labels include Unicode text:

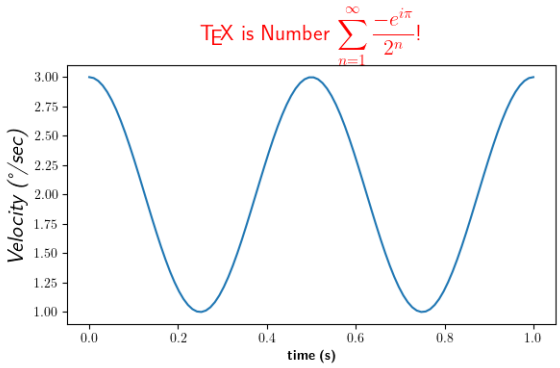


Fig. 46: TeX Unicode Demo

Postscript options

In order to produce encapsulated postscript files that can be embedded in a new LaTeX document, the default behavior of matplotlib is to distill the output, which removes some postscript operators used by LaTeX that are illegal in an eps file. This step produces results which may be unacceptable to some users, because the text is coarsely rasterized and converted to bitmaps, which are not scalable like standard postscript, and the text is not searchable. One workaround is to set `ps.distiller.res` to a higher value (perhaps 6000) in your rc settings, which will produce larger files but may look better and scale reasonably. A better workaround, which requires Poppler or Xpdf, can be activated by changing the `ps.usedistiller` rc setting to `xpdf`. This alternative produces postscript without rasterizing text, so it scales properly, can be edited in Adobe Illustrator, and searched text in pdf documents.

Possible hangups

- On Windows, the *PATH* environment variable may need to be modified to include the directories containing the latex, dvipng and ghostscript executables. See *Environment Variables* and *Setting environment variables in Windows* for details.
- Using MiKTeX with Computer Modern fonts, if you get odd *Agg and PNG results, go to MiKTeX/Options and update your format files
- On Ubuntu and Gentoo, the base texlive install does not ship with the type1cm package. You may need to install some of the extra packages to get all the goodies that come bundled with other latex distributions.

- Some progress has been made so matplotlib uses the dvi files directly for text layout. This allows latex to be used for text layout with the pdf and svg backends, as well as the *Agg and PS backends. In the future, a latex installation may be the only external dependency.

Troubleshooting

- Try deleting your `.matplotlib/tex.cache` directory. If you don't know where to find `.matplotlib`, see *matplotlib configuration and cache directory locations*.
- Make sure LaTeX, dvipng and ghostscript are each working and on your *PATH*.
- Make sure what you are trying to do is possible in a LaTeX document, that your LaTeX syntax is valid and that you are using raw strings if necessary to avoid unintended escape sequences.
- Most problems reported on the mailing list have been cleared up by upgrading Ghostscript. If possible, please try upgrading to the latest release before reporting problems to the list.
- The `text.latex.preamble` rc setting is not officially supported. This option provides lots of flexibility, and lots of ways to cause problems. Please disable this option before reporting problems to the mailing list.
- If you still need help, please see *Getting help*

2.7 Toolkits

These tutorials cover toolkits designed to extend the functionality of Matplotlib in order to accomplish specific goals.

2.7.1 Overview of axes_grid1 toolkit

Controlling the layout of plots with the *mpl_toolkits.axes_grid1* toolkit.

What is axes_grid1 toolkit?

mpl_toolkits.axes_grid1 is a collection of helper classes to ease displaying (multiple) images with matplotlib. In matplotlib, the axes location (and size) is specified in the normalized figure coordinates, which may not be ideal for displaying images that needs to have a given aspect ratio. For example, it helps if you have a colorbar whose height always matches that of the image. *ImageGrid*, *RGB Axes* and *AxesDivider* are helper classes that deal with adjusting the location of (multiple) Axes. They provides a framework to adjust the position of multiple axes at the drawing time. *ParasiteAxes*

provides twinx(or twiny)-like features so that you can plot different data (e.g., different y-scale) in a same Axes. *AnchoredArtists* includes custom artists which are placed at some anchored position, like the legend.

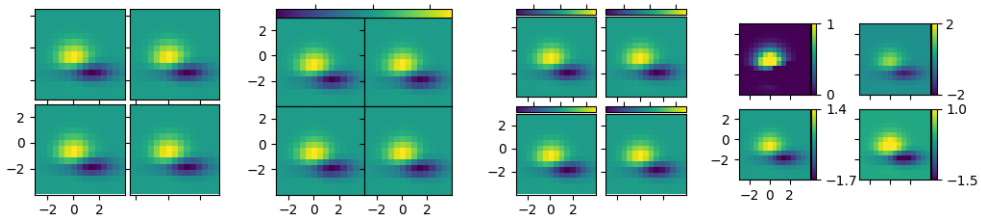


Fig. 47: Demo Axes Grid

axes_grid1

ImageGrid

A grid of Axes.

In Matplotlib, the axes location (and size) is specified in normalized figure coordinates. This may not be ideal for images that needs to be displayed with a given aspect ratio; for example, it is difficult to display multiple images of a same size with some fixed padding between them. *ImageGrid* can be used in such a case; see its docs for a detailed list of the parameters it accepts.

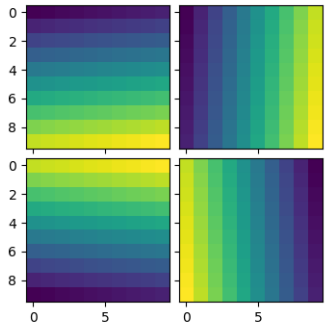


Fig. 48: Simple Axesgrid

- The position of each axes is determined at the drawing time (see *AxesDivider*), so that the size of the entire grid fits in the given rectangle (like the aspect of axes). Note that in this example, the paddings between axes are fixed even if you changes the figure size.
- axes in the same column has a same axes width (in figure coordinate), and similarly, axes in the same row has a same height. The widths (height) of the axes in the same row (column) are scaled according to their view limits (xlim or ylim).

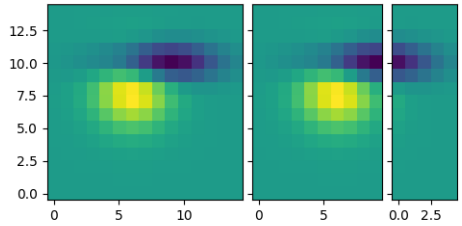


Fig. 49: Simple Axes Grid

- xaxis are shared among axes in a same column. Similarly, yaxis are shared among axes in a same row. Therefore, changing axis properties (view limits, tick location, etc. either by plot commands or using your mouse in interactive backends) of one axes will affect all other shared axes.

The examples below show what you can do with ImageGrid.

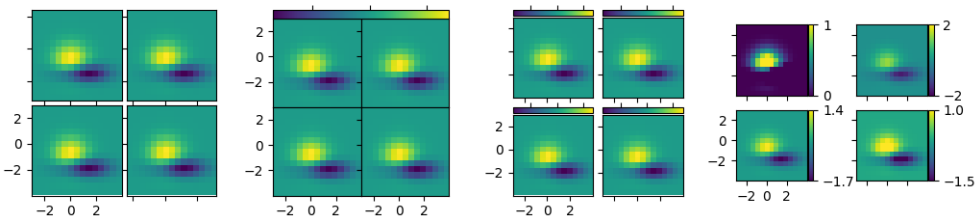


Fig. 50: Demo Axes Grid

AxesDivider Class

Behind the scene, the ImageGrid class and the RGBAxes class utilize the *AxesDivider* class, whose role is to calculate the location of the axes at drawing time. Direct use of the AxesDivider class will not be necessary for most users. The axes_divider module provides a helper function *make_axes_locatable*, which can be useful. It takes a existing axes instance and create a divider for it.

```
ax = subplot(1, 1, 1)
divider = make_axes_locatable(ax)
```

make_axes_locatable returns an instance of the *AxesDivider* class. It provides an *append_axes* method that creates a new axes on the given side of ("top", "right", "bottom" and "left") of the original axes.

colorbar whose height (or width) in sync with the master axes

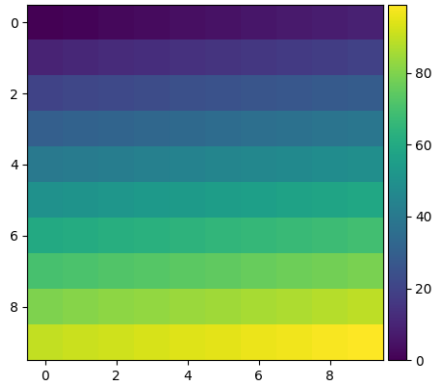


Fig. 51: Simple Colorbar

scatter_hist.py with AxesDivider

The /gallery/lines_bars_and_markers/scatter_hist example can be rewritten using *make_axes_locatable*:

```
axScatter = subplot(111)
axScatter.scatter(x, y)
axScatter.set_aspect(1.)

# create new axes on the right and on the top of the current axes.
divider = make_axes_locatable(axScatter)
axHistx = divider.append_axes("top", size=1.2, pad=0.1, sharex=axScatter)
axHisty = divider.append_axes("right", size=1.2, pad=0.1, sharey=axScatter)

# the scatter plot:
# histograms
bins = np.arange(-lim, lim + binwidth, binwidth)
axHistx.hist(x, bins=bins)
axHisty.hist(y, bins=bins, orientation='horizontal')
```

See the full source code below.

The /gallery/axes_grid1/scatter_hist_locatable_axes using the AxesDivider has some advantage over the original /gallery/lines_bars_and_markers/scatter_hist in Matplotlib. For example, you can set the aspect ratio of the scatter plot, even with the x-axis or y-axis is shared accordingly.

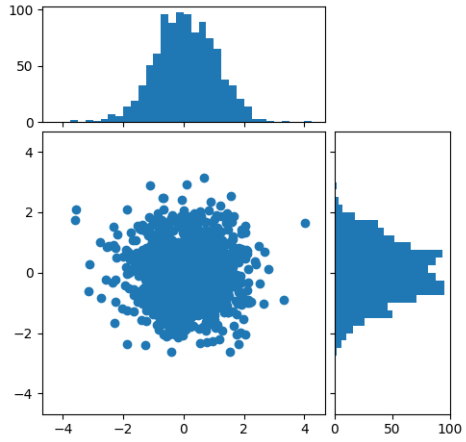


Fig. 52: Scatter Hist

ParasiteAxes

The ParasiteAxes is an axes whose location is identical to its host axes. The location is adjusted in the drawing time, thus it works even if the host change its location (e.g., images).

In most cases, you first create a host axes, which provides a few method that can be used to create parasite axes. They are *twinx*, *twiny* (which are similar to twinx and twiny in the matplotlib) and *twin*. *twin* takes an arbitrary transformation that maps between the data coordinates of the host axes and the parasite axes. *draw* method of the parasite axes are never called. Instead, host axes collects artists in parasite axes and draw them as if they belong to the host axes, i.e., artists in parasite axes are merged to those of the host axes and then drawn according to their zorder. The host and parasite axes modifies some of the axes behavior. For example, color cycle for plot lines are shared between host and parasites. Also, the legend command in host, creates a legend that includes lines in the parasite axes. To create a host axes, you may use *host_subplot* or *host_axes* command.

Example 1. twinx

Example 2. twin

twin without a transform argument assumes that the parasite axes has the same data transform as the host. This can be useful when you want the top(or right)-axis to have different tick-locations, tick-labels, or tick-formatter for bottom(or left)-axis.

```
ax2 = ax.twin() # now, ax2 is responsible for "top" axis and "right" axis
ax2.set_xticks([0., .5*np.pi, np.pi, 1.5*np.pi, 2*np.pi])
```

(continues on next page)

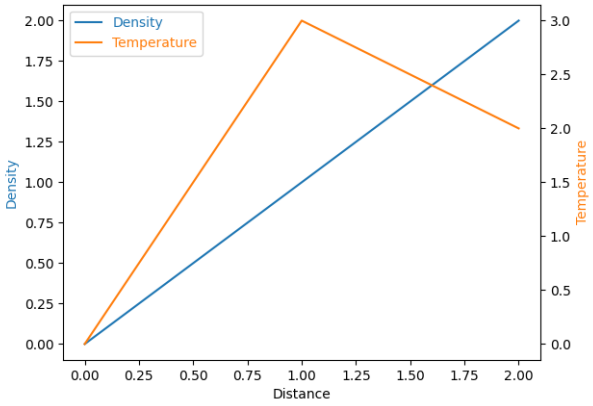


Fig. 53: Parasite Simple

(continued from previous page)

```
ax2.set_xticklabels(["0", r"$\frac{1}{2}\pi$",
                     r"$\pi$", r"$\frac{3}{2}\pi$", r"$2\pi$"])
```

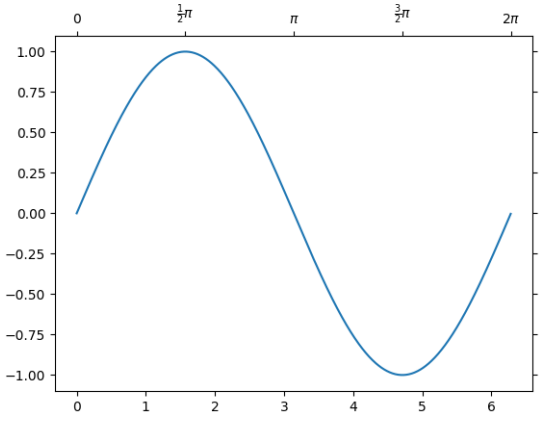


Fig. 54: Simple Axisline4

A more sophisticated example using twin. Note that if you change the x-limit in the host axes, the x-limit of the parasite axes will change accordingly.

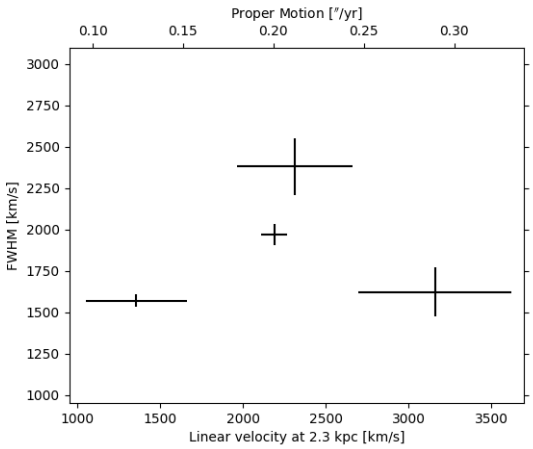


Fig. 55: Parasite Simple2

AnchoredArtists

It's a collection of artists whose location is anchored to the (axes) bbox, like the legend. It is derived from *OffsetBox* in Matplotlib, and artist need to be drawn in the canvas coordinate. But, there is a limited support for an arbitrary transform. For example, the ellipse in the example below will have width and height in the data coordinate.

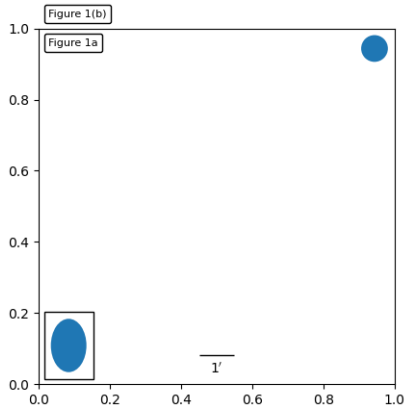


Fig. 56: Simple Anchored Artists

InsetLocator

mpl_toolkits.axes_grid1.inset_locator provides helper classes and functions to place your (inset) axes at the anchored position of the parent axes, similarly to AnchoredArtist.

Using *mpl_toolkits.axes_grid1.inset_locator.inset_axes()*, you can have inset axes whose size is either fixed, or a fixed proportion of the parent axes:

```
inset_axes = inset_axes(parent_axes,
                        width="30%", # width = 30% of parent_bbox
                        height=1., # height : 1 inch
                        loc='lower left')
```

creates an inset axes whose width is 30% of the parent axes and whose height is fixed at 1 inch.

You may creates your inset whose size is determined so that the data scale of the inset axes to be that of the parent axes multiplied by some factor. For example,

```
inset_axes = zoomed_inset_axes(ax,
                               0.5, # zoom = 0.5
                               loc='upper right')
```

creates an inset axes whose data scale is half of the parent axes. Here is complete examples.

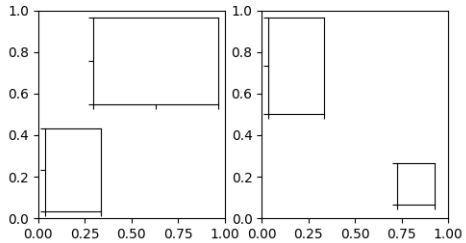


Fig. 57: Inset Locator Demo

For example, *zoomed_inset_axes()* can be used when you want the inset represents the zoom-up of the small portion in the parent axes. And *inset_locator* provides a helper function *mark_inset()* to mark the location of the area represented by the inset axes.

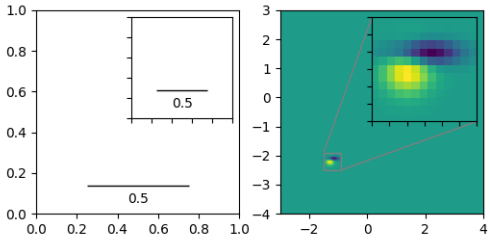
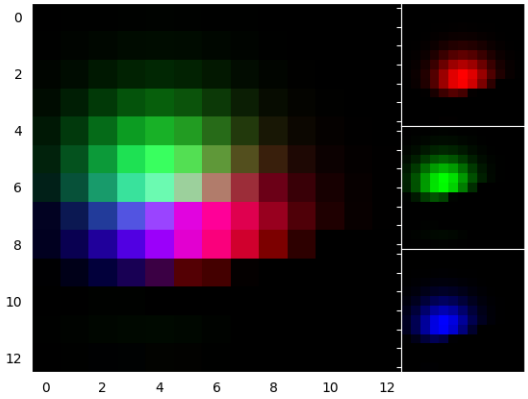


Fig. 58: Inset Locator Demo2

RGB Axes

RGBAxes is a helper class to conveniently show RGB composite images. Like ImageGrid, the location of axes are adjusted so that the area occupied by them fits in a given rectangle. Also, the xaxis and yaxis of each axes are shared.

```
from mpl_toolkits.axes_grid1.axes_rgb import RGBAxes

fig = plt.figure()
ax = RGBAxes(fig, [0.1, 0.1, 0.8, 0.8], pad=0.0)
r, g, b = get_rgb()  # r, g, b are 2D images.
ax.imshow_rgb(r, g, b)
```



AxesDivider

The *mpl_toolkits.axes_grid1.axes_divider* module provides helper classes to adjust the axes positions of a set of images at drawing time.

- *axes_size* provides a class of units that are used to determine the size of each axes. For example, you can specify a fixed size.
- *Divider* is the class that calculates the axes position. It divides the given rectangular area into several areas. The divider is initialized by setting the lists

of horizontal and vertical sizes on which the division will be based. Then use *new_locator()*, which returns a callable object that can be used to set the axes_locator of the axes.

Here, we demonstrate how to achieve the following layout: we want to position axes in a 3x4 grid (note that *Divider* makes row indices start from the *bottom*(!) of the grid):

```
+--------+--------+--------+--------+
| (2, 0) | (2, 1) | (2, 2) | (2, 3) |
+--------+--------+--------+--------+
| (1, 0) | (1, 1) | (1, 2) | (1, 3) |
+--------+--------+--------+--------+
| (0, 0) | (0, 1) | (0, 2) | (0, 3) |
+--------+--------+--------+--------+
```

such that the bottom row has a fixed height of 2 (inches) and the top two rows have a height ratio of 2 (middle) to 3 (top). (For example, if the grid has a size of 7 inches, the bottom row will be 2 inches, the middle row also 2 inches, and the top row 3 inches.)

These constraints are specified using classes from the *axes_size* module, namely:

```
from mpl_toolkits.axes_grid1.axes_size import Fixed, Scaled
vert = [Fixed(2), Scaled(2), Scaled(3)]
```

(More generally, *axes_size* classes define a `get_size(renderer)` method that returns a pair of floats -- a relative size, and an absolute size. `Fixed(2).get_size(renderer)` returns `(0, 2)`; `Scaled(2).get_size(renderer)` returns `(2, 0)`.)

We use these constraints to initialize a *Divider* object:

```
rect = [0.2, 0.2, 0.6, 0.6]  # Position of the grid in the figure.
vert = [Fixed(2), Scaled(2), Scaled(3)]  # As above.
horiz = [...]  # Some other horizontal constraints.
divider = Divider(fig, rect, horiz, vert)
```

then use *Divider.new_locator* to create an *AxesLocator* instance for a given grid entry:

```
locator = divider.new_locator(nx=0, ny=1)  # Grid entry (1, 0).
```

and make it responsible for locating the axes:

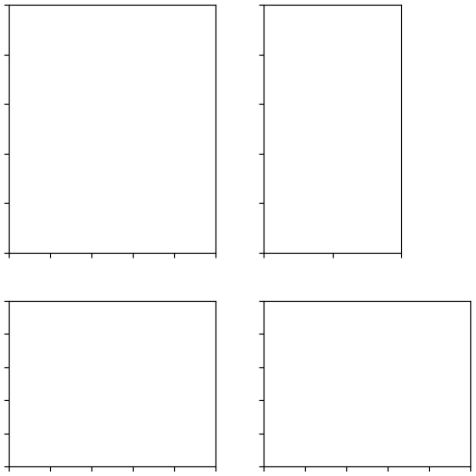
```
ax.set_axes_locator(locator)
```

The *AxesLocator* is a callable object that returns the location and size of the cell at the first column and the second row.

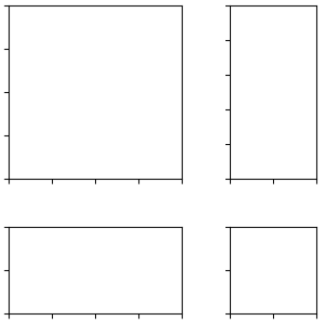
Locators that spans over multiple cells can be created with, e.g.:

```
# Columns #0 and #1 ("0-2 range"), row #1.
locator = divider.new_locator(nx=0, nx1=2, ny=1)
```

See the example,



You can also adjust the size of each axes according to its x or y data limits (AxesX and AxesY).



2.7.2 Overview of axisartist toolkit

The axisartist toolkit tutorial.

Warning: *axisartist* uses a custom Axes class (derived from the Matplotlib's original Axes class). As a side effect, some commands (mostly tick-related) do not work.

The *axisartist* contains a custom Axes class that is meant to support curvilinear grids (e.g., the world coordinate system in astronomy). Unlike Matplotlib's original Axes class which uses Axes.xaxis and Axes.yaxis to draw ticks, ticklines, etc., axisartist uses a special artist (AxisArtist) that can handle ticks, ticklines, etc. for curved coordinate systems.

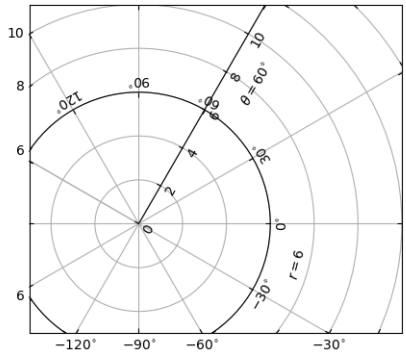


Fig. 59: Demo Floating Axis

Since it uses special artists, some Matplotlib commands that work on Axes.xaxis and Axes.yaxis may not work.

axisartist

The *axisartist* module provides a custom (and very experimental) Axes class, where each axis (left, right, top, and bottom) have a separate associated artist which is responsible for drawing the axis-line, ticks, ticklabels, and labels. You can also create your own axis, which can pass through a fixed position in the axes coordinate, or a fixed position in the data coordinate (i.e., the axis floats around when viewlimit changes).

The axes class, by default, has its xaxis and yaxis invisible, and has 4 additional artists which are responsible for drawing the 4 axis spines in "left", "right", "bottom", and "top". They are accessed as ax.axis["left"], ax.axis["right"], and so on, i.e., ax.axis is a dictionary that contains artists (note that ax.axis is still a callable method and it behaves as an original Axes.axis method in Matplotlib).

To create an axes,

```
import mpl_toolkits.axisartist as AA
fig = plt.figure()
ax = AA.Axes(fig, [0.1, 0.1, 0.8, 0.8])
fig.add_axes(ax)
```

or to create a subplot

```
ax = AA.Subplot(fig, 111)
fig.add_subplot(ax)
```

For example, you can hide the right and top spines using:

```
ax.axis["right"].set_visible(False)
ax.axis["top"].set_visible(False)
```

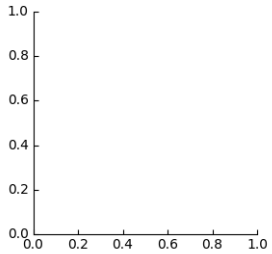


Fig. 60: Simple Axisline3

It is also possible to add a horizontal axis. For example, you may have an horizontal axis at y=0 (in data coordinate).

```
ax.axis["y=0"] = ax.new_floating_axis(nth_coord=0, value=0)
```

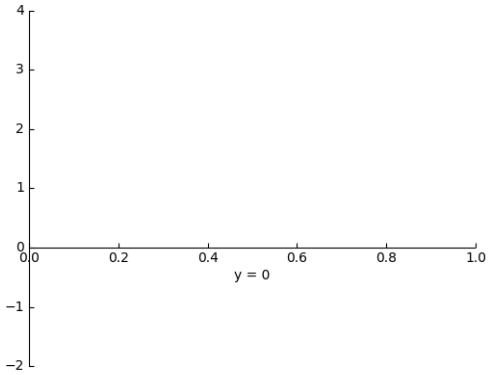


Fig. 61: Simple Axisartist1

Or a fixed axis with some offset

```
# make new (right-side) yaxis, but with some offset
ax.axis["right2"] = ax.new_fixed_axis(loc="right", offset=(20, 0))
```

axisartist with ParasiteAxes

Most commands in the axes_grid1 toolkit can take an axes_class keyword argument, and the commands create an axes of the given class. For example, to create a host subplot with axisartist.Axes,

```
import mpl_toolkits.axisartist as AA
from mpl_toolkits.axes_grid1 import host_subplot

host = host_subplot(111, axes_class=AA.Axes)
```

Here is an example that uses ParasiteAxes.

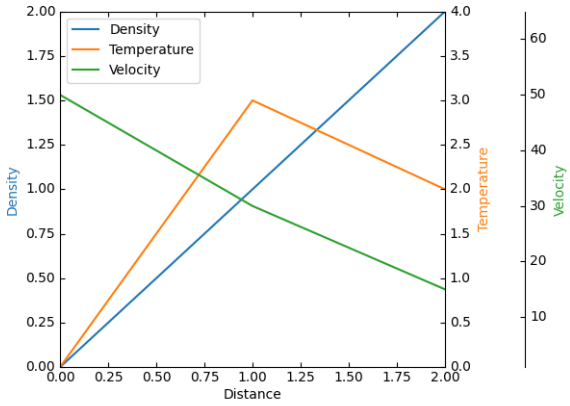


Fig. 62: Demo Parasite Axes2

Curvilinear Grid

The motivation behind the AxisArtist module is to support a curvilinear grid and ticks.

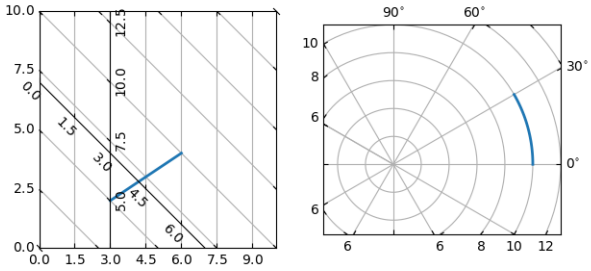


Fig. 63: Demo Curvelinear Grid

Floating Axes

AxisArtist also supports a Floating Axes whose outer axes are defined as floating axis.

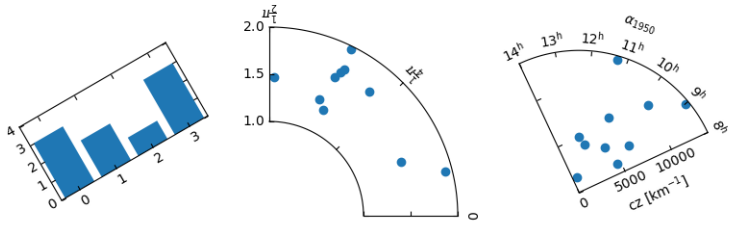


Fig. 64: Demo Floating Axes

axisartist namespace

The *axisartist* namespace includes a derived Axes implementation. The biggest difference is that the artists responsible to draw axis line, ticks, ticklabel and axis labels are separated out from the Matplotlib's Axis class, which are much more than artists in the original Matplotlib. This change was strongly motivated to support curvilinear grid. Here are a few things that mpl_toolkits.axisartist.Axes is different from original Axes from Matplotlib.

- Axis elements (axis line(spine), ticks, ticklabel and axis labels) are drawn by a AxisArtist instance. Unlike Axis, left, right, top and bottom axis are drawn by separate artists. And each of them may have different tick location and different tick labels.
- gridlines are drawn by a Gridlines instance. The change was motivated that in curvilinear coordinate, a gridline may not cross axis-lines (i.e., no associated ticks). In the original Axes class, gridlines are tied to ticks.
- ticklines can be rotated if necessary (i.e, along the gridlines)

In summary, all these changes was to support

- a curvilinear grid.
- a floating axis

mpl_toolkits.axisartist.Axes class defines a *axis* attribute, which is a dictionary of AxisArtist instances. By default, the dictionary has 4 AxisArtist instances, responsible for drawing of left, right, bottom and top axis.

xaxis and yaxis attributes are still available, however they are set to not visible. As separate artists are used for rendering axis, some axis-related method in Matplotlib may have no effect. In addition to AxisArtist instances, the

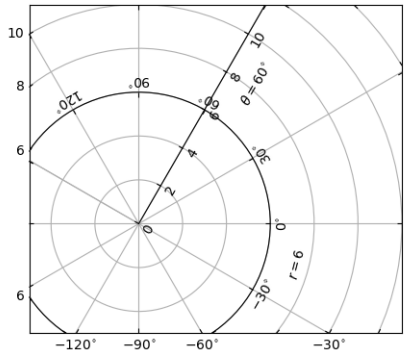


Fig. 65: Demo Floating Axis

mpl_toolkits.axisartist.Axes will have *gridlines* attribute (Gridlines), which obviously draws grid lines.

In both AxisArtist and Gridlines, the calculation of tick and grid location is delegated to an instance of GridHelper class. mpl_toolkits.axisartist.Axes class uses GridHelperRectlinear as a grid helper. The GridHelperRectlinear class is a wrapper around the *xaxis* and *yaxis* of Matplotlib's original Axes, and it was meant to work as the way how Matplotlib's original axes works. For example, tick location changes using set_ticks method and etc. should work as expected. But change in artist properties (e.g., color) will not work in general, although some effort has been made so that some often-change attributes (color, etc.) are respected.

AxisArtist

AxisArtist can be considered as a container artist with following attributes which will draw ticks, labels, etc.

- line
- major_ticks, major_ticklabels
- minor_ticks, minor_ticklabels
- offsetText
- label

line

Derived from Line2D class. Responsible for drawing a spinal(?) line.

major_ticks, minor_ticks

Derived from Line2D class. Note that ticks are markers.

major_ticklabels, minor_ticklabels

Derived from Text. Note that it is not a list of Text artist, but a single artist (similar to a collection).

axislabel

Derived from Text.

Default AxisArtists

By default, following for axis artists are defined.:

```
ax.axis["left"], ax.axis["bottom"], ax.axis["right"], ax.axis["top"]
```

The ticklabels and axislabel of the top and the right axis are set to not visible.

For example, if you want to change the color attributes of major_ticklabels of the bottom x-axis

```
ax.axis["bottom"].major_ticklabels.set_color("b")
```

Similarly, to make ticklabels invisible

```
ax.axis["bottom"].major_ticklabels.set_visible(False)
```

AxisArtist provides a helper method to control the visibility of ticks, ticklabels, and label. To make ticklabel invisible,

```
ax.axis["bottom"].toggle(ticklabels=False)
```

To make all of ticks, ticklabels, and (axis) label invisible

```
ax.axis["bottom"].toggle(all=False)
```

To turn all off but ticks on

```
ax.axis["bottom"].toggle(all=False, ticks=True)
```

To turn all on but (axis) label off

```
ax.axis["bottom"].toggle(all=True, label=False)
```

ax.axis's __getitem__ method can take multiple axis names. For example, to turn ticklabels of "top" and "right" axis on,

```
ax.axis["top", "right"].toggle(ticklabels=True)
```

Note that `ax.axis["top", "right"]` returns a simple proxy object that translate above code to something like below.

```
for n in ["top", "right"]:
    ax.axis[n].toggle(ticklabels=True)
```

So, any return values in the for loop are ignored. And you should not use it anything more than a simple method.

Like the list indexing ":" means all items, i.e.,

```
ax.axis[:].major_ticks.set_color("r")
```

changes tick color in all axis.

HowTo

1. Changing tick locations and label.

   Same as the original Matplotlib's axes:

   ```
   ax.set_xticks([1, 2, 3])
   ```

2. Changing axis properties like color, etc.

   Change the properties of appropriate artists. For example, to change the color of the ticklabels:

   ```
   ax.axis["left"].major_ticklabels.set_color("r")
   ```

3. To change the attributes of multiple axis:

   ```
   ax.axis["left", "bottom"].major_ticklabels.set_color("r")
   ```

   or to change the attributes of all axis:

   ```
   ax.axis[:].major_ticklabels.set_color("r")
   ```

4. To change the tick size (length), you need to use axis.major_ticks.set_ticksize method. To change the direction of the ticks (ticks are in opposite direction of ticklabels by default), use axis.major_ticks.set_tick_out method.

   To change the pad between ticks and ticklabels, use axis.major_ticklabels.set_pad method.

   To change the pad between ticklabels and axis label, axis.label.set_pad method.

Rotation and Alignment of TickLabels

This is also quite different from standard Matplotlib and can be confusing. When you want to rotate the ticklabels, first consider using "set_axis_direction" method.

```
ax1.axis["left"].major_ticklabels.set_axis_direction("top")
ax1.axis["right"].label.set_axis_direction("left")
```

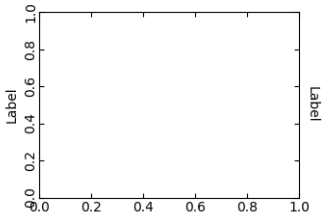


Fig. 66: Simple Axis Direction01

The parameter for set_axis_direction is one of ["left", "right", "bottom", "top"].

You must understand some underlying concept of directions.

1. There is a reference direction which is defined as the direction of the axis line with increasing coordinate. For example, the reference direction of the left x-axis is from bottom to top.

Fig. 67: Axis Direction Demo - Step 01

   The direction, text angle, and alignments of the ticks, ticklabels and axis-label is determined with respect to the reference direction

2. *ticklabel_direction* is either the right-hand side (+) of the reference direction or the left-hand side (-).

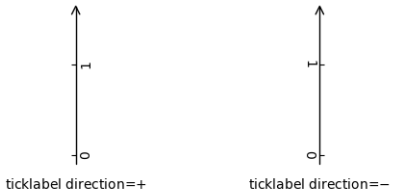


Fig. 68: Axis Direction Demo - Step 02

Fig. 69: Axis Direction Demo - Step 03

3. same for the *label_direction*
4. ticks are by default drawn toward the opposite direction of the ticklabels.
5. text rotation of ticklabels and label is determined in reference to the *ticklabel_direction* or *label_direction*, respectively. The rotation of ticklabels and label is anchored.

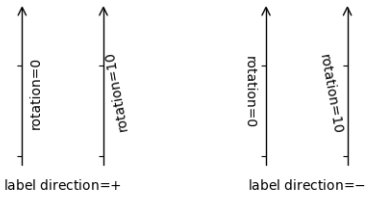


Fig. 70: Axis Direction Demo - Step 04

On the other hand, there is a concept of "axis_direction". This is a default setting of above properties for each, "bottom", "left", "top", and "right" axis.

?	?	left	bottom	right	top
axislabel	direction	'-'	'+'	'+'	'-'
axislabel	rotation	180	0	0	180
axislabel	va	center	top	center	bottom
axislabel	ha	right	center	right	center
ticklabel	direction	'-'	'+'	'+'	'-'
ticklabels	rotation	90	0	-90	180
ticklabel	ha	right	center	right	center
ticklabel	va	center	baseline	center	baseline

And, 'set_axis_direction("top")' means to adjust the text rotation etc, for settings suitable for "top" axis. The concept of axis direction can be more clear with curved axis.

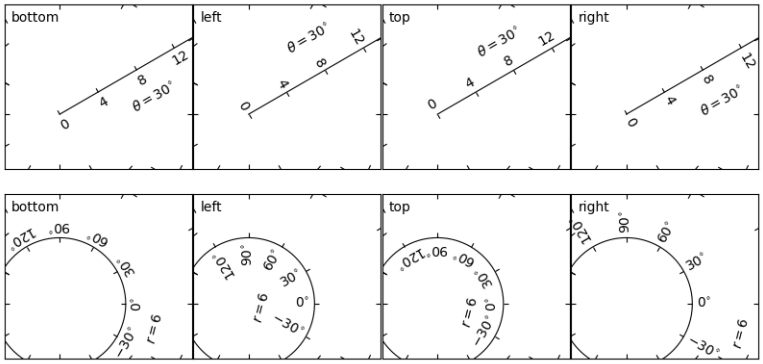


Fig. 71: Demo Axis Direction

The axis_direction can be adjusted in the AxisArtist level, or in the level of its child artists, i.e., ticks, ticklabels, and axis-label.

```
ax1.axis["left"].set_axis_direction("top")
```

changes axis_direction of all the associated artist with the "left" axis, while

```
ax1.axis["left"].major_ticklabels.set_axis_direction("top")
```

changes the axis_direction of only the major_ticklabels. Note that set_axis_direction in the AxisArtist level changes the ticklabel_direction and label_direction, while changing the axis_direction of ticks, ticklabels, and axis-label does not affect them.

If you want to make ticks outward and ticklabels inside the axes, use invert_ticklabel_direction method.

```
ax.axis[:].invert_ticklabel_direction()
```

A related method is "set_tick_out". It makes ticks outward (as a matter of fact, it makes ticks toward the opposite direction of the default direction).

```
ax.axis[:].major_ticks.set_tick_out(True)
```

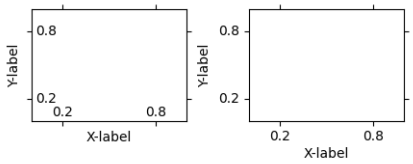


Fig. 72: Simple Axis Direction03

So, in summary,

- AxisArtist's methods
  - set_axis_direction: "left", "right", "bottom", or "top"
  - set_ticklabel_direction: "+" or "-"
  - set_axislabel_direction: "+" or "-"
  - invert_ticklabel_direction
- Ticks' methods (major_ticks and minor_ticks)
  - set_tick_out: True or False
  - set_ticksize: size in points
- TickLabels' methods (major_ticklabels and minor_ticklabels)
  - set_axis_direction: "left", "right", "bottom", or "top"
  - set_rotation: angle with respect to the reference direction
  - set_ha and set_va: see below
- AxisLabels' methods (label)
  - set_axis_direction: "left", "right", "bottom", or "top"
  - set_rotation: angle with respect to the reference direction
  - set_ha and set_va

Adjusting ticklabels alignment

Alignment of TickLabels are treated specially. See below

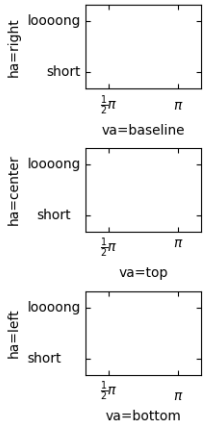


Fig. 73: Demo Ticklabel Alignment

Adjusting pad

To change the pad between ticks and ticklabels

```
ax.axis["left"].major_ticklabels.set_pad(10)
```

Or ticklabels and axis-label

```
ax.axis["left"].label.set_pad(10)
```

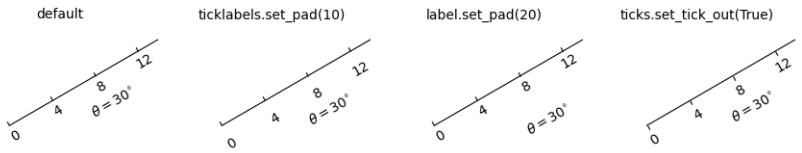


Fig. 74: Simple Axis Pad

GridHelper

To actually define a curvilinear coordinate, you have to use your own grid helper. A generalised version of grid helper class is supplied and this class should suffice in most of cases. A user may provide two functions which defines a transformation (and its inverse pair) from the curved coordinate to (rectilinear) image coordinate. Note that while ticks and grids are drawn for curved coordinate, the data transform of the axes itself (ax.transData) is still rectilinear (image) coordinate.

```
from mpl_toolkits.axisartist.grid_helper_curvelinear \
     import GridHelperCurveLinear
from mpl_toolkits.axisartist import Subplot

# from curved coordinate to rectlinear coordinate.
def tr(x, y):
    x, y = np.asarray(x), np.asarray(y)
    return x, y-x

# from rectlinear coordinate to curved coordinate.
def inv_tr(x, y):
    x, y = np.asarray(x), np.asarray(y)
    return x, y+x

grid_helper = GridHelperCurveLinear((tr, inv_tr))

ax1 = Subplot(fig, 1, 1, 1, grid_helper=grid_helper)

fig.add_subplot(ax1)
```

You may use Matplotlib's Transform instance instead (but a inverse transformation must be defined). Often, coordinate range in a curved coordinate system may have a limited range, or may have cycles. In those cases, a more customized version of grid helper is required.

```
import mpl_toolkits.axisartist.angle_helper as angle_helper

# PolarAxes.PolarTransform takes radian. However, we want our coordinate
# system in degree
tr = Affine2D().scale(np.pi/180., 1.) + PolarAxes.PolarTransform()

# extreme finder: find a range of coordinate.
# 20, 20: number of sampling points along x, y direction
# The first coordinate (longitude, but theta in polar)
#   has a cycle of 360 degree.
# The second coordinate (latitude, but radius in polar)  has a minimum of 0
extreme_finder = angle_helper.ExtremeFinderCycle(20, 20,
                                                 lon_cycle = 360,
                                                 lat_cycle = None,
                                                 lon_minmax = None,
                                                 lat_minmax = (0, np.inf),
                                                 )

# Find a grid values appropriate for the coordinate (degree,
# minute, second). The argument is a approximate number of grids.
grid_locator1 = angle_helper.LocatorDMS(12)

# And also uses an appropriate formatter.  Note that the acceptable Locator
# and Formatter classes are different than that of Matplotlib's, and you
# cannot directly use Matplotlib's Locator and Formatter here (but may be
# possible in the future).
tick_formatter1 = angle_helper.FormatterDMS()
```

(continues on next page)

(continued from previous page)

```
grid_helper = GridHelperCurveLinear(tr,
                                    extreme_finder=extreme_finder,
                                    grid_locator1=grid_locator1,
                                    tick_formatter1=tick_formatter1
                                    )
```

Again, the *transData* of the axes is still a rectilinear coordinate (image coordinate). You may manually do conversion between two coordinates, or you may use Parasite Axes for convenience.:

```
ax1 = SubplotHost(fig, 1, 2, 2, grid_helper=grid_helper)

# A parasite axes with given transform
ax2 = ParasiteAxesAuxTrans(ax1, tr, "equal")
# note that ax2.transData == tr + ax1.transData
# Anything you draw in ax2 will match the ticks and grids of ax1.
ax1.parasites.append(ax2)
```

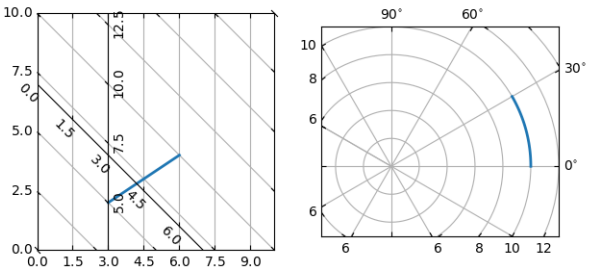


Fig. 75: Demo Curvelinear Grid

FloatingAxis

A floating axis is an axis one of whose data coordinate is fixed, i.e, its location is not fixed in Axes coordinate but changes as axes data limits changes. A floating axis can be created using *new_floating_axis* method. However, it is your responsibility that the resulting AxisArtist is properly added to the axes. A recommended way is to add it as an item of Axes's axis attribute.:

```
# floating axis whose first (index starts from 0) coordinate
# (theta) is fixed at 60

ax1.axis["lat"] = axis = ax1.new_floating_axis(0, 60)
axis.label.set_text(r"$\theta = 60^{\circ}$")
axis.label.set_visible(True)
```

See the first example of this page.

Current Limitations and TODO's

The code need more refinement. Here is a incomplete list of issues and TODO's

- No easy way to support a user customized tick location (for curvilinear grid). A new Locator class needs to be created.
- FloatingAxis may have coordinate limits, e.g., a floating axis of x = 0, but y only spans from 0 to 1.
- The location of axislabel of FloatingAxis needs to be optionally given as a coordinate value. ex, a floating axis of x=0 with label at y=1

2.7.3 The mplot3d Toolkit

Generating 3D plots using the mplot3d toolkit.

Contents

Getting started

3D Axes (of class *Axes3D*) are created by passing the `projection="3d"` keyword argument to *Figure.add_subplot*:

```
import matplotlib.pyplot as plt
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
```

Changed in version 1.0.0: Prior to Matplotlib 1.0.0, *Axes3D* needed to be directly instantiated with `from mpl_toolkits.mplot3d import Axes3D; ax = Axes3D(fig)`.

Changed in version 3.2.0: Prior to Matplotlib 3.2.0, it was necessary to explicitly import the *mpl_toolkits.mplot3d* module to make the '3d' projection to *Figure.add_subplot*.

See the *mplot3d FAQ* for more information about the mplot3d toolkit.

Line plots

`Axes3D.plot`(*self*, *xs*, *ys*, **args*, *zdir='z'*, ***kwargs*)

Plot 2D or 3D data.

Parameters

xs

[1D array-like] x coordinates of vertices.

ys

[1D array-like] y coordinates of vertices.

zs

[float or 1D array-like] z coordinates of vertices; either one for all points or one for each point.

zdir

[{'x', 'y', 'z'}, default: 'z'] When plotting 2D data, the direction to use as z ('x', 'y' or 'z').

****kwargs**

Other arguments are forwarded to *matplotlib.axes.Axes.plot*.

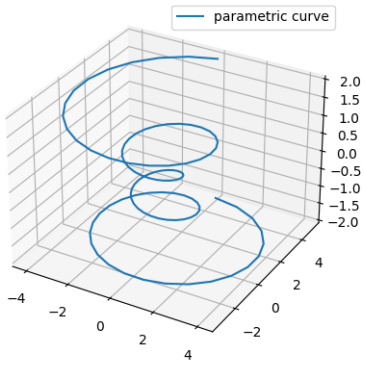


Fig. 76: Lines3d

Scatter plots

`Axes3D.scatter`(*self*, *xs*, *ys*, *zs=0*, *zdir='z'*, *s=20*, *c=None*, *depthshade=True*, **args*, ***kwargs*)

Create a scatter plot.

Parameters

xs, ys

[array-like] The data positions.

zs

[float or array-like, default: 0] The z-positions. Either an array of the same length as *xs* and *ys* or a single value to place all points in the same plane.

zdir

[{'x', 'y', 'z', '-x', '-y', '-z'}, default: 'z'] The axis direction for the *zs*. This is useful when plotting 2D data on a 3D Axes. The data must be passed as *xs*, *ys*. Setting *zdir* to 'y' then plots the data to the x-z-plane.

See also /gallery/mplot3d/2dcollections3d.

s

[float or array-like, default: 20] The marker size in points**2. Either an array of the same length as *xs* and *ys* or a single value to make all markers the same size.

c

[color, sequence, or sequence of colors, optional] The marker color. Possible values:

- A single color format string.
- A sequence of colors of length n.
- A sequence of n numbers to be mapped to colors using *cmap* and *norm*.
- A 2-D array in which the rows are RGB or RGBA.

For more details see the *c* argument of *scatter*.

depthshade

[bool, default: True] Whether to shade the scatter markers to give the appearance of depth. Each call to `scatter()` will perform its depthshading independently.

****kwargs**

All other arguments are passed on to *scatter*.

Returns

paths

[*PathCollection*]

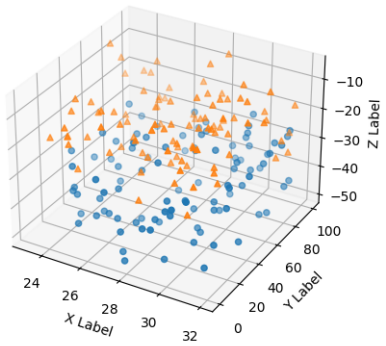


Fig. 77: Scatter3d

Wireframe plots

Axes3D.plot_wireframe(*self*, *X*, *Y*, *Z*, **args*, ***kwargs*)

Plot a 3D wireframe.

Note: The *rcount* and *ccount* kwargs, which both default to 50, determine the maximum number of samples used in each direction. If the input data is larger, it will be downsampled (by slicing) to these numbers of points.

Parameters

X, Y, Z

[2d arrays] Data values.

rcount, ccount

[int] Maximum number of samples used in each direction. If the input data is larger, it will be downsampled (by slicing) to these numbers of points. Setting a count to zero causes the data to be not sampled in the corresponding direction, producing a 3D line plot rather than a wireframe plot. Defaults to 50.

New in version 2.0.

rstride, cstride

[int] Downsampling stride in each direction. These arguments are mutually exclusive with *rcount* and *ccount*. If only one of *rstride* or *cstride* is set, the other defaults to 1. Setting a stride to zero causes the data to be not sampled in the corresponding direction, producing a 3D line plot rather than a wireframe plot.

'classic' mode uses a default of `rstride = cstride = 1` instead of the new default of `rcount = ccount = 50`.

****kwargs**

Other arguments are forwarded to *Line3DCollection*.

Surface plots

Axes3D.plot_surface(*self*, *X*, *Y*, *Z*, **args*, *norm=None*, *vmin=None*, *vmax=None*, *lightsource=None*, ***kwargs*)

Create a surface plot.

By default it will be colored in shades of a solid color, but it also supports color mapping by supplying the *cmap* argument.

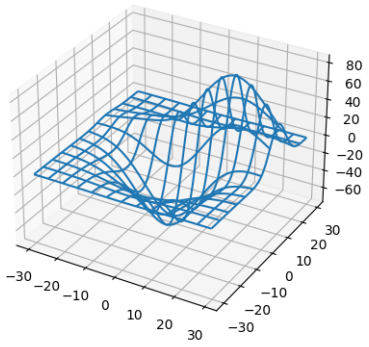


Fig. 78: Wire3d

Note: The *rcount* and *ccount* kwargs, which both default to 50, determine the maximum number of samples used in each direction. If the input data is larger, it will be downsampled (by slicing) to these numbers of points.

Note: To maximize rendering speed consider setting *rstride* and *cstride* to divisors of the number of rows minus 1 and columns minus 1 respectively. For example, given 51 rows rstride can be any of the divisors of 50.

Similarly, a setting of *rstride* and *cstride* equal to 1 (or *rcount* and *ccount* equal the number of rows and columns) can use the optimized path.

Parameters

X, Y, Z

[2d arrays] Data values.

rcount, ccount

[int] Maximum number of samples used in each direction. If the input data is larger, it will be downsampled (by slicing) to these numbers of points. Defaults to 50.

New in version 2.0.

rstride, cstride

[int] Downsampling stride in each direction. These arguments are mutually exclusive with *rcount* and *ccount*. If only one of *rstride* or *cstride* is set, the other defaults to 10.

'classic' mode uses a default of `rstride = cstride = 10` instead of the new default of `rcount = ccount = 50`.

color

[color-like] Color of the surface patches.

cmap

[Colormap] Colormap of the surface patches.

facecolors

[array-like of colors.] Colors of each individual patch.

norm

[Normalize] Normalization for the colormap.

vmin, vmax

[float] Bounds for the normalization.

shade

[bool, default: True] Whether to shade the facecolors. Shading is always disabled when *cmap* is specified.

lightsource

[*LightSource*] The lightsource to use when *shade* is True.

****kwargs**

Other arguments are forwarded to *Poly3DCollection*.

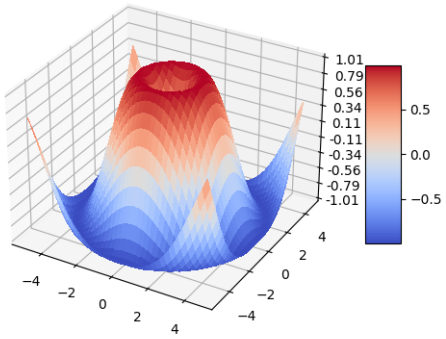


Fig. 79: Surface3d
Surface3d 2
Surface3d 3

Tri-Surface plots

Axes3D.**plot_trisurf**(*self*, **args*, *color=None*, *norm=None*, *vmin=None*, *vmax=None*, *lightsource=None*, ***kwargs*)

Plot a triangulated surface.

The (optional) triangulation can be specified in one of two ways; either:

```
plot_trisurf(triangulation, ...)
```

where triangulation is a *Triangulation* object, or:

```
plot_trisurf(X, Y, ...)
plot_trisurf(X, Y, triangles, ...)
plot_trisurf(X, Y, triangles=triangles, ...)
```

in which case a Triangulation object will be created. See *Triangulation* for a explanation of these possibilities.

The remaining arguments are:

```
plot_trisurf(..., Z)
```

where *Z* is the array of values to contour, one per point in the triangulation.

Parameters

X, Y, Z

[array-like] Data values as 1D arrays.

color

Color of the surface patches.

cmap

A colormap for the surface patches.

norm

[Normalize] An instance of Normalize to map values to colors.

vmin, vmax

[float, default: None] Minimum and maximum value to map.

shade

[bool, default: True] Whether to shade the facecolors. Shading is always disabled when *cmap* is specified.

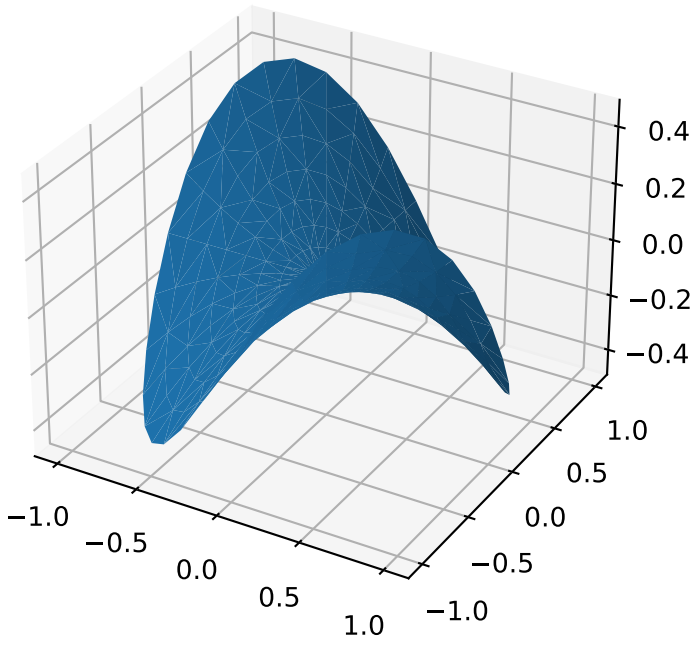
lightsource

[*LightSource*] The lightsource to use when *shade* is True.

****kwargs**

All other arguments are passed on to *Poly3DCollection*

Examples



New in version 1.2.0.

Contour plots

`Axes3D.contour`(*self*, *X*, *Y*, *Z*, **args*, *extend3d=False*, *stride=5*, *zdir='z'*, *offset=None*, ***kwargs*)

Create a 3D contour plot.

Parameters

X, Y, Z

[array-like] Input data.

extend3d

[bool, default: False] Whether to extend contour in 3D.

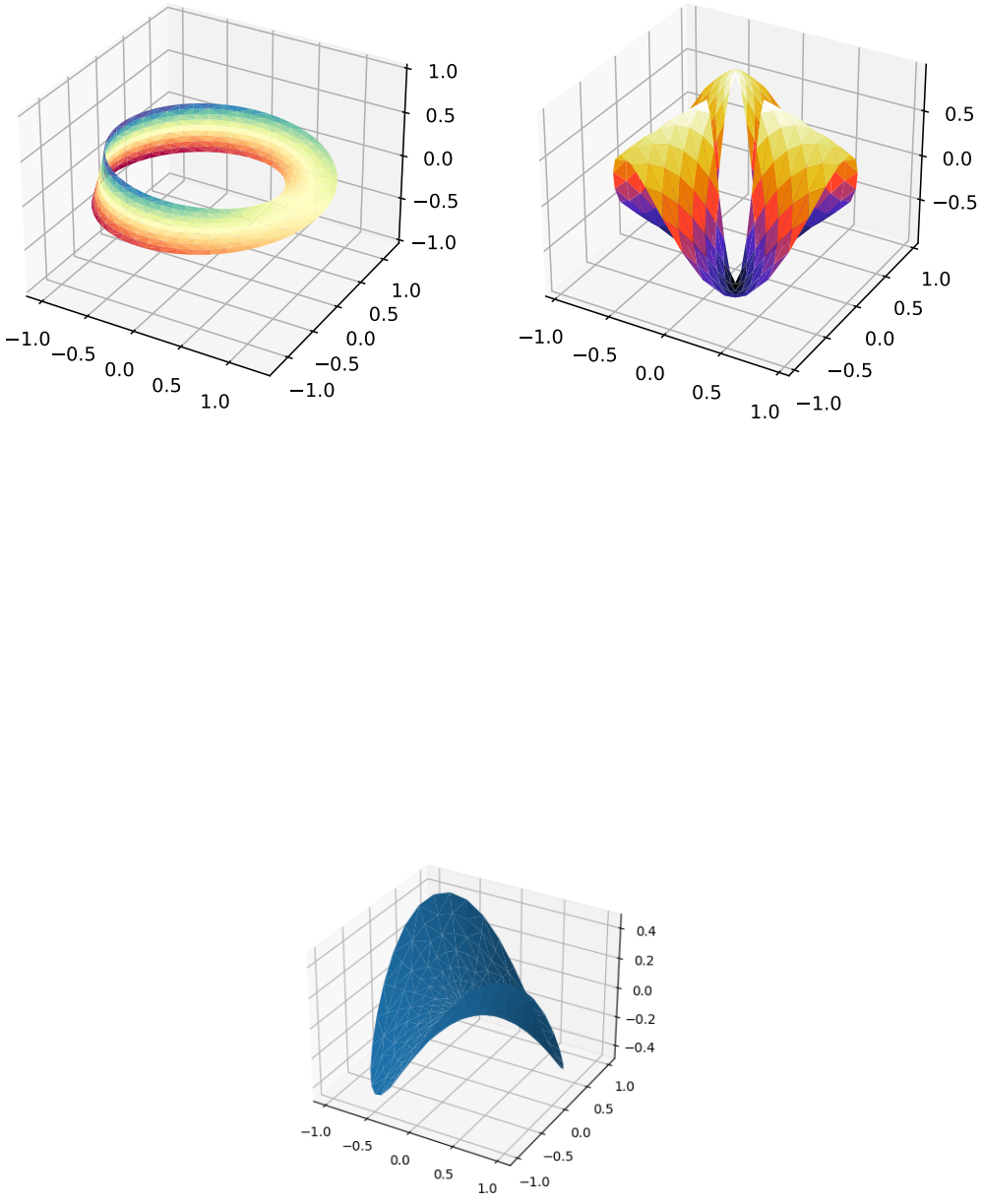


Fig. 80: Trisurf3d

stride

[int] Step size for extending contour.

zdir

[{'x', 'y', 'z'}, default: 'z'] The direction to use.

offset

[float, optional] If specified, plot a projection of the contour lines at this position in a plane normal to zdir.

***args, **kwargs**

Other arguments are forwarded to *matplotlib.axes.Axes.contour*.

Returns

matplotlib.contour.QuadContourSet

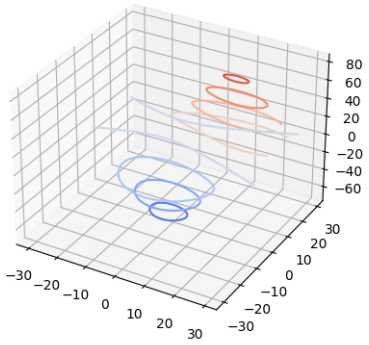


Fig. 81: Contour3d
Contour3d 2
Contour3d 3

Filled contour plots

`Axes3D.contourf`(*self*, *X*, *Y*, *Z*, **args*, *zdir='z'*, *offset=None*, ***kwargs*)

Create a 3D filled contour plot.

Parameters

X, Y, Z

[array-like] Input data.

zdir

[{'x', 'y', 'z'}, default: 'z'] The direction to use.

offset

[float, optional] If specified, plot a projection of the contour lines at this position in a plane normal to zdir.

***args, **kwargs**

Other arguments are forwarded to *matplotlib.axes.Axes.contourf*.

Returns

matplotlib.contour.QuadContourSet

Notes

New in version 1.1.0: The *zdir* and *offset* parameters.

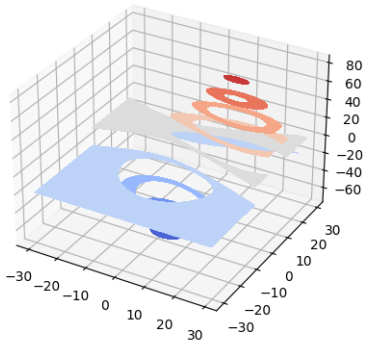


Fig. 82: Contourf3d
Contourf3d 2

New in version 1.1.0: The feature demoed in the second contourf3d example was enabled as a result of a bugfix for version 1.1.0.

Polygon plots

Axes3D.add_collection3d(*self*, *col*, *zs=0*, *zdir='z'*)

Add a 3D collection object to the plot.

2D collection types are converted to a 3D version by modifying the object and adding z coordinate information.

Supported are:

- PolyCollection
- LineCollection
- PatchCollection

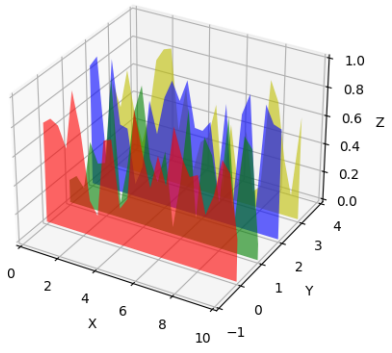


Fig. 83: Polys3d

Bar plots

Axes3D.bar(*self*, *left*, *height*, *zs=0*, *zdir='z'*, **args*, ***kwargs*)

Add 2D bar(s).

Parameters

left

[1D array-like] The x coordinates of the left sides of the bars.

height

[1D array-like] The height of the bars.

zs

[float or 1D array-like] Z coordinate of bars; if a single value is specified, it will be used for all bars.

zdir

[{'x', 'y', 'z'}, default: 'z'] When plotting 2D data, the direction to use as z ('x', 'y' or 'z').

****kwargs**

Other arguments are forwarded to *matplotlib.axes.Axes.bar*.

Returns

mpl_toolkits.mplot3d.art3d.Patch3DCollection

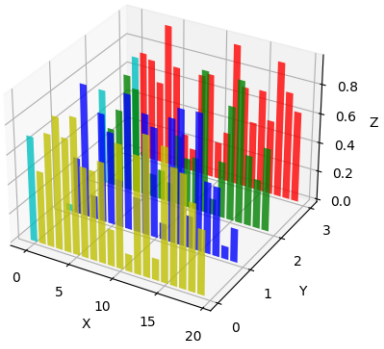


Fig. 84: Bars3d

Quiver

`Axes3D.quiver`(*X, Y, Z, U, V, W, /, length=1, arrow_length_ratio=0.3, pivot='tail', normalize=False, **kwargs*)

Plot a 3D field of arrows.

The arguments could be array-like or scalars, so long as they they can be broadcast together. The arguments can also be masked arrays. If an element in any of argument is masked, then that corresponding quiver element will not be plotted.

Parameters

X, Y, Z

[array-like] The x, y and z coordinates of the arrow locations (default is tail of arrow; see *pivot* kwarg).

U, V, W

[array-like] The x, y and z components of the arrow vectors.

length

[float, default: 1] The length of each quiver.

arrow_length_ratio

[float, default: 0.3] The ratio of the arrow head with respect to the quiver.

pivot

[{'tail', 'middle', 'tip'}, default: 'tail'] The part of the arrow that is at the grid point; the arrow rotates about this point, hence the name *pivot*.

normalize

[bool, default: False] Whether all arrows are normalized to have the same length, or keep the lengths defined by *u*, *v*, and *w*.

****kwargs**

Any additional keyword arguments are delegated to *LineCollection*

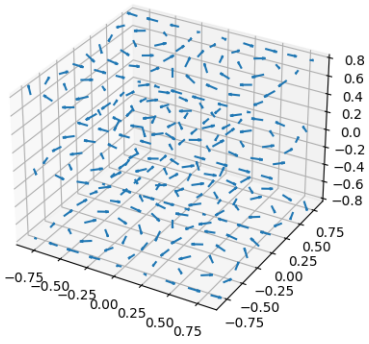


Fig. 85: Quiver3d

2D plots in 3D

Text

`Axes3D.text`(*self*, *x*, *y*, *z*, *s*, *zdir=None*, ***kwargs*)

Add text to the plot. kwargs will be passed on to Axes.text, except for the *zdir* keyword, which sets the direction to be used as the z direction.

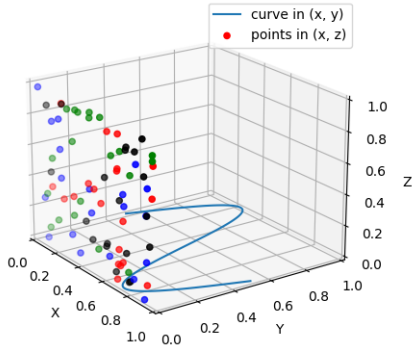


Fig. 86: 2dcollections3d

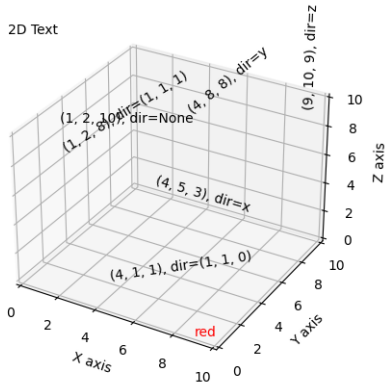


Fig. 87: Text3d

Subplotting

Having multiple 3D plots in a single figure is the same as it is for 2D plots. Also, you can have both 2D and 3D plots in the same figure.

New in version 1.0.0: Subplotting 3D plots was added in v1.0.0. Earlier version can not do this.

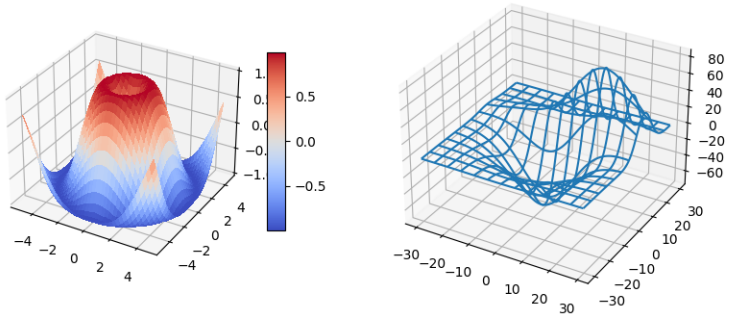


Fig. 88: Subplot3d
Mixed Subplots

CHAPTER

THREE

INTERACTIVE FIGURES

When working with data, interactivity can be invaluable. The pan/zoom and mouse-location tools built into the Matplotlib GUI windows are often sufficient, but you can also use the event system to build customized data exploration tools.

Matplotlib ships with *backends* binding to several GUI toolkits (Qt, Tk, Wx, GTK, macOS, JavaScript) and third party packages provide bindings to kivy and Jupyter Lab. For the figures to be responsive to mouse, keyboard, and paint events, the GUI event loop needs to be integrated with an interactive prompt. We recommend using IPython (see *below*).

The *pyplot* module provides functions for explicitly creating figures that include interactive tools, a toolbar, a tool-tip, and *key bindings*:

pyplot.figure

Creates a new empty *figure.Figure* or selects an existing figure

pyplot.subplots

Creates a new *figure.Figure* and fills it with a grid of *axes.Axes*

pyplot has a notion of "The Current Figure" which can be accessed through *pyplot.gcf* and a notion of "The Current Axes" accessed through *pyplot.gca*. Almost all of the functions in *pyplot* pass through the current *Figure* / *axes.Axes* (or create one) as appropriate.

Matplotlib keeps a reference to all of the open figures created via *pyplot.figure* or *pyplot.subplots* so that the figures will not be garbage collected. *Figure*s can be closed and deregistered from *pyplot* individually via *pyplot.close*; all open *Figure*s can be closed via `plt.close('all')`.

For more discussion of Matplotlib's event system and integrated event loops, please read:

3.1 Interactive Figures and Asynchronous Programming

Matplotlib supports rich interactive figures by embedding figures into a GUI window. The basic interactions of panning and zooming in an Axes to inspect your data is 'baked in' to Matplotlib. This is supported by a full mouse and keyboard event handling system that you can use to build sophisticated interactive graphs.

This guide is meant to be an introduction to the low-level details of how Matplotlib integration with a GUI event loop works. For a more practical introduction to the Matplotlib event API see *event handling system*, Interactive Tutorial, and Interactive Applications using Matplotlib.

3.1.1 Event Loops

Fundamentally, all user interaction (and networking) is implemented as an infinite loop waiting for events from the user (via the OS) and then doing something about it. For example, a minimal Read Evaluate Print Loop (REPL) is

```
exec_count = 0
while True:
    inp = input(f"[{exec_count}] > ")        # Read
    ret = eval(inp)                          # Evaluate
    print(ret)                               # Print
    exec_count += 1                          # Loop
```

This is missing many niceties (for example, it exits on the first exception!), but is representative of the event loops that underlie all terminals, GUIs, and servers[1]. In general the *Read* step is waiting on some sort of I/O -- be it user input or the network -- while the *Evaluate* and *Print* are responsible for interpreting the input and then **doing** something about it.

In practice we interact with a framework that provides a mechanism to register callbacks to be run in response to specific events rather than directly implement the I/O loop[2]. For example "when the user clicks on this button, please run this function" or "when the user hits the 'z' key, please run this other function". This allows users to write reactive, event-driven, programs without having to delve into the nitty-gritty[3] details of I/O. The core event loop is sometimes referred to as "the main loop" and is typically started, depending on the library, by methods with names like `_exec`, `run`, or `start`.

[1] A limitation of this design is that you can only wait for one input, if there is a need to multiplex between multiple sources then the loop would look something like

```
fds = [...]
while True:                     # Loop
    inp = select(fds).read()    # Read
    eval(inp)                   # Evaluate / Print
```

[2] Or you can write your own if you must.

[3] These examples are aggressively dropping many of the complexities that must be dealt with in the real world such as keyboard interrupts, timeouts, bad input, resource allocation and cleanup, etc.

All GUI frameworks (Qt, Wx, Gtk, tk, OSX, or web) have some method of capturing user interactions and passing them back to the application (for example `Signal` / `Slot` framework in Qt) but the exact details depend on the toolkit. Matplotlib has a *backend* for each GUI toolkit we support which uses the toolkit API to bridge the toolkit UI events into Matplotlib's *event handling system*. You can then use *FigureCanvasBase.mpl_connect* to connect your function to Matplotlib's event handling system. This allows you to directly interact with your data and write GUI toolkit agnostic user interfaces.

3.1.2 Command Prompt Integration

So far, so good. We have the REPL (like the IPython terminal) that lets us interactively send code to the interpreter and get results back. We also have the GUI toolkit that runs an event loop waiting for user input and lets us register functions to be run when that happens. However, if we want to do both we have a problem: the prompt and the GUI event loop are both infinite loops that each think *they* are in charge! In order for both the prompt and the GUI windows to be responsive we need a method to allow the loops to 'timeshare' :

1. let the GUI main loop block the python process when you want interactive windows
2. let the CLI main loop block the python process and intermittently run the GUI loop
3. fully embed python in the GUI (but this is basically writing a full application)

Blocking the Prompt

pyplot.show	Display all open figures.
pyplot.pause	Run the GUI event loop for *interval* seconds.
backend_bases.FigureCanvasBase.start_event_loop	Start a blocking event loop.
backend_bases.FigureCanvasBase.stop_event_loop	Stop the current blocking event loop.

The simplest "integration" is to start the GUI event loop in 'blocking' mode and take over the CLI. While the GUI event loop is running you can not enter new commands into the prompt (your terminal may echo the characters typed into the terminal, but they will not be sent to the Python interpreter because it is busy running the GUI event loop), but the figure windows will be responsive. Once the event loop is stopped (leaving any still open figure windows non-responsive) you will be able to use the prompt again. Re-starting the event loop will make any open figure responsive again (and will process any queued up user interaction).

To start the event loop until all open figures are closed use *pyplot.show* as

```
pyplot.show(block=True)
```

To start the event loop for a fixed amount of time (in seconds) use *pyplot.pause*.

If you are not using *pyplot* you can start and stop the event loops via *FigureCanvasBase.start_event_loop* and *FigureCanvasBase.stop_event_loop*. However, in most contexts where you would not be using *pyplot* you are embedding Matplotlib in a large GUI application and the GUI event loop should already be running for the application.

Away from the prompt, this technique can be very useful if you want to write a script that pauses for user interaction, or displays a figure between polling for additional data. See *Scripts and functions* for more details.

Input Hook integration

While running the GUI event loop in a blocking mode or explicitly handling UI events is useful, we can do better! We really want to be able to have a usable prompt **and** interactive figure windows.

We can do this using the 'input hook' feature of the interactive prompt. This hook is called by the prompt as it waits for the user to type (even for a fast typist the prompt is mostly waiting for the human to think and move their fingers). Although the details vary between prompts the logic is roughly

1. start to wait for keyboard input
2. start the GUI event loop
3. as soon as the user hits a key, exit the GUI event loop and handle the key
4. repeat

This gives us the illusion of simultaneously having interactive GUI windows and an interactive prompt. Most of the time the GUI event loop is running, but as soon as the user starts typing the prompt takes over again.

This time-share technique only allows the event loop to run while python is otherwise idle and waiting for user input. If you want the GUI to be responsive during long running code it is necessary to periodically flush the GUI event queue as described *above*. In this case it is your code, not the REPL, which is blocking the process so you need to handle the "time-share" manually. Conversely, a very slow figure draw will block the prompt until it finishes drawing.

3.1.3 Full embedding

It is also possible to go the other direction and fully embed figures (and a Python interpreter) in a rich native application. Matplotlib provides classes for each toolkit which can be directly embedded in GUI applications (this is how the built-in windows are implemented!). See user_interfaces for more details.

3.1.4 Scripts and functions

backend_bases.FigureCanvasBase.flush_events	Flush the GUI events for the figure.
backend_bases.FigureCanvasBase.draw_idle	Request a widget redraw once control returns to the GUI event loop.
figure.Figure.ginput	Blocking call to interact with a figure.
pyplot.ginput	Blocking call to interact with a figure.
pyplot.show	Display all open figures.
pyplot.pause	Run the GUI event loop for *interval* seconds.

There are several use-cases for using interactive figures in scripts:

- capture user input to steer the script
- progress updates as a long running script progresses
- streaming updates from a data source

Blocking functions

If you only need to collect points in an Axes you can use *figure.Figure.ginput* or more generally the tools from *blocking_input* the tools will take care of starting and stopping the event loop for you. However if you have written some custom event handling or are using *widgets* you will need to manually run the GUI event loop using the methods described *above*.

You can also use the methods described in *Blocking the Prompt* to suspend run the GUI event loop. Once the loop exits your code will resume. In general, any place you would use `time.sleep` you can use *pyplot.pause* instead with the added benefit of interactive figures.

For example, if you want to poll for data you could use something like

```
fig, ax = plt.subplots()
ln, = ax.plot([], [])

while True:
    x, y = get_new_data()
```

(continues on next page)

(continued from previous page)

```
    ln.set_data(x, y)
    plt.pause(1)
```

which would poll for new data and update the figure at 1Hz.

Explicitly spinning the Event Loop

backend_bases.FigureCanvasBase.flush_events	Flush the GUI events for the figure.
backend_bases.FigureCanvasBase.draw_idle	Request a widget redraw once control returns to the GUI event loop.

If you have open windows that have pending UI events (mouse clicks, button presses, or draws) you can explicitly process those events by calling *FigureCanvasBase.flush_events*. This will run the GUI event loop until all UI events currently waiting have been processed. The exact behavior is backend-dependent but typically events on all figure are processed and only events waiting to be processed (not those added during processing) will be handled.

For example

```
import time
import matplotlib.pyplot as plt
import numpy as np
plt.ion()

fig, ax = plt.subplots()
th = np.linspace(0, 2*np.pi, 512)
ax.set_ylim(-1.5, 1.5)

ln, = ax.plot(th, np.sin(th))

def slow_loop(N, ln):
    for j in range(N):
        time.sleep(.1)  # to simulate some work
        ln.figure.canvas.flush_events()

slow_loop(100, ln)
```

While this will feel a bit laggy (as we are only processing user input every 100ms whereas 20-30ms is what feels "responsive") it will respond.

If you make changes to the plot and want it re-rendered you will need to call *draw_idle* to request that the canvas be re-drawn. This method can be thought of *draw_soon* in analogy to `asyncio.loop.call_soon`.

We can add this our example above as

```
def slow_loop(N, ln):
    for j in range(N):
        time.sleep(.1)  # to simulate some work
        if j % 10:
            ln.set_ydata(np.sin(((j // 10) % 5 * th)))
            ln.figure.canvas.draw_idle()

        ln.figure.canvas.flush_events()

slow_loop(100, ln)
```

The more frequently you call *FigureCanvasBase.flush_events* the more responsive your figure will feel but at the cost of spending more resources on the visualization and less on your computation.

3.1.5 Stale Artists

Artists (as of Matplotlib 1.5) have a **stale** attribute which is `True` if the internal state of the artist has changed since the last time it was rendered. By default the stale state is propagated up to the Artists parents in the draw tree, e.g., if the color of a *Line2D* instance is changed, the *axes.Axes* and *figure.Figure* that contain it will also be marked as "stale". Thus, `fig.stale` will report if any artist in the figure has been modified and is out of sync with what is displayed on the screen. This is intended to be used to determine if `draw_idle` should be called to schedule a re-rendering of the figure.

Each artist has a `Artist.stale_callback` attribute which holds a callback with the signature

```
def callback(self: Artist, val: bool) -> None:
   ...
```

which by default is set to a function that forwards the stale state to the artist's parent. If you wish to suppress a given artist from propagating set this attribute to None.

figure.Figure instances do not have a containing artist and their default callback is `None`. If you call *pyplot.ion* and are not in `IPython` we will install a callback to invoke *draw_idle* whenever the *figure.Figure* becomes stale. In `IPython` we use the `'post_execute'` hook to invoke *draw_idle* on any stale figures after having executed the user's input, but before returning the prompt to the user. If you are not using *pyplot* you can use the callback `Figure.stale_callback` attribute to be notified when a figure has become stale.

3.1.6 Draw Idle

backend_bases.FigureCanvasBase.draw	Render the *Figure*.
backend_bases.FigureCanvasBase.draw_idle	Request a widget redraw once control returns to the GUI event loop.
backend_bases.FigureCanvasBase.flush_events	Flush the GUI events for the figure.

In almost all cases, we recommend using *backend_bases.FigureCanvasBase.draw_idle* over *backend_bases.FigureCanvasBase.draw*. `draw` forces a rendering of the figure whereas `draw_idle` schedules a rendering the next time the GUI window is going to re-paint the screen. This improves performance by only rendering pixels that will be shown on the screen. If you want to be sure that the screen is updated as soon as possible do

```
fig.canvas.draw_idle()
fig.canvas.flush_events()
```

3.1.7 Threading

Most GUI frameworks require that all updates to the screen, and hence their main event loop, run on the main thread. This makes pushing periodic updates of a plot to a background thread impossible. Although it seems backwards, it is typically easier to push your computations to a background thread and periodically update the figure on the main thread.

In general Matplotlib is not thread safe. If you are going to update *Artist* objects in one thread and draw from another you should make sure that you are locking in the critical sections.

3.1.8 Eventloop integration mechanism

CPython / readline

The Python C API provides a hook, `PyOS_InputHook`, to register a function to be run "The function will be called when Python's interpreter prompt is about to become idle and wait for user input from the terminal.". This hook can be used to integrate a second event loop (the GUI event loop) with the python input prompt loop. The hook functions typically exhaust all pending events on the GUI event queue, run the main loop for a short fixed amount of time, or run the event loop until a key is pressed on stdin.

Matplotlib does not currently do any management of `PyOS_InputHook` due to the wide range of ways that Matplotlib is used. This management is left to downstream libraries -- either user code or the shell. Interactive figures, even with matplotlib

in 'interactive mode', may not work in the vanilla python repl if an appropriate `PyOS_InputHook` is not registered.

Input hooks, and helpers to install them, are usually included with the python bindings for GUI toolkits and may be registered on import. IPython also ships input hook functions for all of the GUI frameworks Matplotlib supports which can be installed via `%matplotlib`. This is the recommended method of integrating Matplotlib and a prompt.

IPython / prompt toolkit

With IPython >= 5.0 IPython has changed from using cpython's readline based prompt to a `prompt_toolkit` based prompt. `prompt_toolkit` has the same conceptual input hook, which is fed into `prompt_toolkit` via the `IPython.terminal.interactiveshell.TerminalInteractiveShell.inputhook()` method. The source for the `prompt_toolkit` input hooks lives at `IPython.terminal.pt_inputhooks`

3.2 Event handling and picking

Matplotlib works with a number of user interface toolkits (wxpython, tkinter, qt4, gtk, and macosx) and in order to support features like interactive panning and zooming of figures, it is helpful to the developers to have an API for interacting with the figure via key presses and mouse movements that is "GUI neutral" so we don't have to repeat a lot of code across the different user interfaces. Although the event handling API is GUI neutral, it is based on the GTK model, which was the first user interface matplotlib supported. The events that are triggered are also a bit richer vis-a-vis matplotlib than standard GUI events, including information like which *matplotlib.axes.Axes* the event occurred in. The events also understand the matplotlib coordinate system, and report event locations in both pixel and data coordinates.

3.2.1 Event connections

To receive events, you need to write a callback function and then connect your function to the event manager, which is part of the *FigureCanvasBase*. Here is a simple example that prints the location of the mouse click and which button was pressed:

```
fig, ax = plt.subplots()
ax.plot(np.random.rand(10))

def onclick(event):
    print('%s click: button=%d, x=%d, y=%d, xdata=%f, ydata=%f' %
          ('double' if event.dblclick else 'single', event.button,
           event.x, event.y, event.xdata, event.ydata))

cid = fig.canvas.mpl_connect('button_press_event', onclick)
```

The `FigureCanvas` method *mpl_connect()* returns a connection id which is simply an integer. When you want to disconnect the callback, just call:

```
fig.canvas.mpl_disconnect(cid)
```

Note: The canvas retains only weak references to instance methods used as callbacks. Therefore, you need to retain a reference to instances owning such methods. Otherwise the instance will be garbage-collected and the callback will vanish.

This does not affect free functions used as callbacks.

Here are the events that you can connect to, the class instances that are sent back to you when the event occurs, and the event descriptions:

Event name	Class	Description
'button_press_event'	*MouseEvent*	mouse button is pressed
'button_release_event'	*MouseEvent*	mouse button is released
'close_event'	*CloseEvent*	figure is closed
'draw_event'	*DrawEvent*	canvas has been drawn (but screen widget not updated yet)
'key_press_event'	*KeyEvent*	key is pressed
'key_release_event'	*KeyEvent*	key is released
'motion_notify_event'	*MouseEvent*	mouse moves
'pick_event'	*PickEvent*	artist in the canvas is selected
'resize_event'	*ResizeEvent*	figure canvas is resized
'scroll_event'	*MouseEvent*	mouse scroll wheel is rolled
'figure_enter_event'	*LocationEvent*	mouse enters a new figure
'figure_leave_event'	*LocationEvent*	mouse leaves a figure
'axes_enter_event'	*LocationEvent*	mouse enters a new axes
'axes_leave_event'	*LocationEvent*	mouse leaves an axes

3.2.2 Event attributes

All matplotlib events inherit from the base class *matplotlib.backend_bases.Event*, which store the attributes:

name

the event name

canvas

the FigureCanvas instance generating the event

guiEvent

the GUI event that triggered the matplotlib event

The most common events that are the bread and butter of event handling are key press/release events and mouse press/release and movement events. The *KeyEvent* and *MouseEvent* classes that handle these events are both derived from the LocationEvent, which has the following attributes

x

x position - pixels from left of canvas

y

y position - pixels from bottom of canvas

inaxes

the *Axes* instance if mouse is over axes

xdata

x coord of mouse in data coords

ydata

y coord of mouse in data coords

Let's look a simple example of a canvas, where a simple line segment is created every time a mouse is pressed:

```
from matplotlib import pyplot as plt

class LineBuilder:
    def __init__(self, line):
        self.line = line
        self.xs = list(line.get_xdata())
        self.ys = list(line.get_ydata())
        self.cid = line.figure.canvas.mpl_connect('button_press_event', self)

    def __call__(self, event):
        print('click', event)
        if event.inaxes!=self.line.axes: return
        self.xs.append(event.xdata)
        self.ys.append(event.ydata)
        self.line.set_data(self.xs, self.ys)
        self.line.figure.canvas.draw()

fig = plt.figure()
ax = fig.add_subplot(111)
ax.set_title('click to build line segments')
line, = ax.plot([0], [0])  # empty line
linebuilder = LineBuilder(line)
```

(continues on next page)

(continued from previous page)

```
plt.show()
```

The *MouseEvent* that we just used is a *LocationEvent*, so we have access to the data and pixel coordinates in event.x and event.xdata. In addition to the `LocationEvent` attributes, it has

button

button pressed None, 1, 2, 3, 'up', 'down' (up and down are used for scroll events)

key

the key pressed: None, any character, 'shift', 'win', or 'control'

Draggable rectangle exercise

Write draggable rectangle class that is initialized with a *Rectangle* instance but will move its x,y location when dragged. Hint: you will need to store the original `xy` location of the rectangle which is stored as rect.xy and connect to the press, motion and release mouse events. When the mouse is pressed, check to see if the click occurs over your rectangle (see `matplotlib.patches.Rectangle.contains()`) and if it does, store the rectangle xy and the location of the mouse click in data coords. In the motion event callback, compute the deltax and deltay of the mouse movement, and add those deltas to the origin of the rectangle you stored. The redraw the figure. On the button release event, just reset all the button press data you stored as None.

Here is the solution:

```
import numpy as np
import matplotlib.pyplot as plt

class DraggableRectangle:
    def __init__(self, rect):
        self.rect = rect
        self.press = None

    def connect(self):
        'connect to all the events we need'
        self.cidpress = self.rect.figure.canvas.mpl_connect(
            'button_press_event', self.on_press)
        self.cidrelease = self.rect.figure.canvas.mpl_connect(
            'button_release_event', self.on_release)
        self.cidmotion = self.rect.figure.canvas.mpl_connect(
            'motion_notify_event', self.on_motion)

    def on_press(self, event):
        'on button press we will see if the mouse is over us and store some data'
        if event.inaxes != self.rect.axes: return
```

(continues on next page)

(continued from previous page)

```
        contains, attrd = self.rect.contains(event)
        if not contains: return
        print('event contains', self.rect.xy)
        x0, y0 = self.rect.xy
        self.press = x0, y0, event.xdata, event.ydata

    def on_motion(self, event):
        'on motion we will move the rect if the mouse is over us'
        if self.press is None: return
        if event.inaxes != self.rect.axes: return
        x0, y0, xpress, ypress = self.press
        dx = event.xdata - xpress
        dy = event.ydata - ypress
        #print('x0=%f, xpress=%f, event.xdata=%f, dx=%f, x0+dx=%f' %
        #      (x0, xpress, event.xdata, dx, x0+dx))
        self.rect.set_x(x0+dx)
        self.rect.set_y(y0+dy)

        self.rect.figure.canvas.draw()


    def on_release(self, event):
        'on release we reset the press data'
        self.press = None
        self.rect.figure.canvas.draw()

    def disconnect(self):
        'disconnect all the stored connection ids'
        self.rect.figure.canvas.mpl_disconnect(self.cidpress)
        self.rect.figure.canvas.mpl_disconnect(self.cidrelease)
        self.rect.figure.canvas.mpl_disconnect(self.cidmotion)

fig = plt.figure()
ax = fig.add_subplot(111)
rects = ax.bar(range(10), 20*np.random.rand(10))
drs = []
for rect in rects:
    dr = DraggableRectangle(rect)
    dr.connect()
    drs.append(dr)

plt.show()
```

Extra credit: Use blitting to make the animated drawing faster and smoother.

Extra credit solution:

```
# Draggable rectangle with blitting.
import numpy as np
import matplotlib.pyplot as plt
```

(continues on next page)

(continued from previous page)

```
class DraggableRectangle:
    lock = None  # only one can be animated at a time
    def __init__(self, rect):
        self.rect = rect
        self.press = None
        self.background = None

    def connect(self):
        'connect to all the events we need'
        self.cidpress = self.rect.figure.canvas.mpl_connect(
            'button_press_event', self.on_press)
        self.cidrelease = self.rect.figure.canvas.mpl_connect(
            'button_release_event', self.on_release)
        self.cidmotion = self.rect.figure.canvas.mpl_connect(
            'motion_notify_event', self.on_motion)

    def on_press(self, event):
        'on button press we will see if the mouse is over us and store some data'
        if event.inaxes != self.rect.axes: return
        if DraggableRectangle.lock is not None: return
        contains, attrd = self.rect.contains(event)
        if not contains: return
        print('event contains', self.rect.xy)
        x0, y0 = self.rect.xy
        self.press = x0, y0, event.xdata, event.ydata
        DraggableRectangle.lock = self

        # draw everything but the selected rectangle and store the pixel buffer
        canvas = self.rect.figure.canvas
        axes = self.rect.axes
        self.rect.set_animated(True)
        canvas.draw()
        self.background = canvas.copy_from_bbox(self.rect.axes.bbox)

        # now redraw just the rectangle
        axes.draw_artist(self.rect)

        # and blit just the redrawn area
        canvas.blit(axes.bbox)

    def on_motion(self, event):
        'on motion we will move the rect if the mouse is over us'
        if DraggableRectangle.lock is not self:
            return
        if event.inaxes != self.rect.axes: return
        x0, y0, xpress, ypress = self.press
        dx = event.xdata - xpress
        dy = event.ydata - ypress
        self.rect.set_x(x0+dx)
        self.rect.set_y(y0+dy)
```

(continues on next page)

(continued from previous page)

```
        canvas = self.rect.figure.canvas
        axes = self.rect.axes
        # restore the background region
        canvas.restore_region(self.background)

        # redraw just the current rectangle
        axes.draw_artist(self.rect)

        # blit just the redrawn area
        canvas.blit(axes.bbox)

    def on_release(self, event):
        'on release we reset the press data'
        if DraggableRectangle.lock is not self:
            return

        self.press = None
        DraggableRectangle.lock = None

        # turn off the rect animation property and reset the background
        self.rect.set_animated(False)
        self.background = None

        # redraw the full figure
        self.rect.figure.canvas.draw()

    def disconnect(self):
        'disconnect all the stored connection ids'
        self.rect.figure.canvas.mpl_disconnect(self.cidpress)
        self.rect.figure.canvas.mpl_disconnect(self.cidrelease)
        self.rect.figure.canvas.mpl_disconnect(self.cidmotion)

fig = plt.figure()
ax = fig.add_subplot(111)
rects = ax.bar(range(10), 20*np.random.rand(10))
drs = []
for rect in rects:
    dr = DraggableRectangle(rect)
    dr.connect()
    drs.append(dr)

plt.show()
```

3.2.3 Mouse enter and leave

If you want to be notified when the mouse enters or leaves a figure or axes, you can connect to the figure/axes enter/leave events. Here is a simple example that changes the colors of the axes and figure background that the mouse is over:

```
"""
Illustrate the figure and axes enter and leave events by changing the
frame colors on enter and leave
"""
import matplotlib.pyplot as plt

def enter_axes(event):
    print('enter_axes', event.inaxes)
    event.inaxes.patch.set_facecolor('yellow')
    event.canvas.draw()

def leave_axes(event):
    print('leave_axes', event.inaxes)
    event.inaxes.patch.set_facecolor('white')
    event.canvas.draw()

def enter_figure(event):
    print('enter_figure', event.canvas.figure)
    event.canvas.figure.patch.set_facecolor('red')
    event.canvas.draw()

def leave_figure(event):
    print('leave_figure', event.canvas.figure)
    event.canvas.figure.patch.set_facecolor('grey')
    event.canvas.draw()

fig1 = plt.figure()
fig1.suptitle('mouse hover over figure or axes to trigger events')
ax1 = fig1.add_subplot(211)
ax2 = fig1.add_subplot(212)

fig1.canvas.mpl_connect('figure_enter_event', enter_figure)
fig1.canvas.mpl_connect('figure_leave_event', leave_figure)
fig1.canvas.mpl_connect('axes_enter_event', enter_axes)
fig1.canvas.mpl_connect('axes_leave_event', leave_axes)

fig2 = plt.figure()
fig2.suptitle('mouse hover over figure or axes to trigger events')
ax1 = fig2.add_subplot(211)
ax2 = fig2.add_subplot(212)

fig2.canvas.mpl_connect('figure_enter_event', enter_figure)
fig2.canvas.mpl_connect('figure_leave_event', leave_figure)
fig2.canvas.mpl_connect('axes_enter_event', enter_axes)
fig2.canvas.mpl_connect('axes_leave_event', leave_axes)

plt.show()
```

3.2.4 Object picking

You can enable picking by setting the `picker` property of an *Artist* (e.g., a matplotlib *Line2D*, *Text*, *Patch*, *Polygon*, *AxesImage*, etc...)

There are a variety of meanings of the `picker` property:

None

picking is disabled for this artist (default)

boolean

if True then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist

float

if picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if its data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event.

function

if picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event. The signature is `hit, props = picker(artist, mouseevent)` to determine the hit test. If the mouse event is over the artist, return `hit=True` and props is a dictionary of properties you want added to the *PickEvent* attributes

After you have enabled an artist for picking by setting the `picker` property, you need to connect to the figure canvas pick_event to get pick callbacks on mouse press events. e.g.:

```
def pick_handler(event):
    mouseevent = event.mouseevent
    artist = event.artist
    # now do something with this...
```

The *PickEvent* which is passed to your callback is always fired with two attributes:

mouseevent **the mouse event that generate the pick event. The**

mouse event in turn has attributes like `x` and `y` (the coords in display space, e.g., pixels from left, bottom) and xdata, ydata (the coords in data space). Additionally, you can get information about which buttons were pressed, which keys were pressed, which *Axes* the mouse is over, etc. See *matplotlib.backend_bases.MouseEvent* for details.

artist

the *Artist* that generated the pick event.

Additionally, certain artists like *Line2D* and *PatchCollection* may attach additional meta data like the indices into the data that meet the picker criteria (e.g., all the points in the line that are within the specified epsilon tolerance)

Simple picking example

In the example below, we set the line picker property to a scalar, so it represents a tolerance in points (72 points per inch). The onpick callback function will be called when the pick event it within the tolerance distance from the line, and has the indices of the data vertices that are within the pick distance tolerance. Our onpick callback function simply prints the data that are under the pick location. Different matplotlib Artists can attach different data to the PickEvent. For example, `Line2D` attaches the ind property, which are the indices into the line data under the pick point. See `pick()` for details on the `PickEvent` properties of the line. Here is the code:

```
import numpy as np
import matplotlib.pyplot as plt

fig = plt.figure()
ax = fig.add_subplot(111)
ax.set_title('click on points')

line, = ax.plot(np.random.rand(100), 'o', picker=5)  # 5 points tolerance

def onpick(event):
    thisline = event.artist
    xdata = thisline.get_xdata()
    ydata = thisline.get_ydata()
    ind = event.ind
    points = tuple(zip(xdata[ind], ydata[ind]))
    print('onpick points:', points)

fig.canvas.mpl_connect('pick_event', onpick)

plt.show()
```

Picking exercise

Create a data set of 100 arrays of 1000 Gaussian random numbers and compute the sample mean and standard deviation of each of them (hint: NumPy arrays have a mean and std method) and make a xy marker plot of the 100 means vs. the 100 standard deviations. Connect the line created by the plot command to the pick event, and plot the original time series of the data that generated the clicked on points. If more than one point is within the tolerance of the clicked on point, you can use multiple subplots to plot the multiple time series.

Exercise solution:

```
"""
compute the mean and stddev of 100 data sets and plot mean vs. stddev.
When you click on one of the mu, sigma points, plot the raw data from
the dataset that generated the mean and stddev
"""
import numpy as np
import matplotlib.pyplot as plt

X = np.random.rand(100, 1000)
xs = np.mean(X, axis=1)
ys = np.std(X, axis=1)

fig = plt.figure()
ax = fig.add_subplot(111)
ax.set_title('click on point to plot time series')
line, = ax.plot(xs, ys, 'o', picker=5)  # 5 points tolerance


def onpick(event):

    if event.artist!=line: return True

    N = len(event.ind)
    if not N: return True


    figi = plt.figure()
    for subplotnum, dataind in enumerate(event.ind):
        ax = figi.add_subplot(N,1,subplotnum+1)
        ax.plot(X[dataind])
        ax.text(0.05, 0.9, 'mu=%1.3f\nsigma=%1.3f'%(xs[dataind], ys[dataind]),
                transform=ax.transAxes, va='top')
        ax.set_ylim(-0.5, 1.5)
    figi.show()
    return True

fig.canvas.mpl_connect('pick_event', onpick)

plt.show()
```

3.3 IPython integration

We recommend using IPython for an interactive shell. In addition to all of its features (improved tab-completion, magics, multiline editing, etc), it also ensures that the GUI toolkit event loop is properly integrated with the command line (see *Command Prompt Integration*).

In this example, we create and modify a figure via an IPython prompt. The figure displays in a Qt5Agg GUI window. To configure the integration and enable *interactive mode* use the `%matplotlib` magic:

::

    In [1]: %matplotlib Using matplotlib backend: Qt5Agg

    In [2]: import matplotlib.pyplot as plt

Create a new figure window:

```
In [3]: fig, ax = plt.subplots()
```

Add a line plot of the data to the window:

```
In [4]: ln, = ax.plot(range(5))
```

Change the color of the line from blue to orange:

```
In [5]: ln.set_color('orange')
```

If you wish to disable automatic redrawing of the plot:

```
In [6]: plt.ioff()
```

If you wish to re-enable automatic redrawing of the plot:

```
In [7]: plt.ion()
```

In recent versions of `Matplotlib` and `IPython`, it is sufficient to import *matplotlib.pyplot* and call *pyplot.ion*. Using the `%` magic is guaranteed to work in all versions of Matplotlib and IPython.

3.4 Interactive mode

pyplot.ion	Turn the interactive mode on.
pyplot.ioff	Turn the interactive mode off.
pyplot.isinteractive	Return if pyplot is in "interactive mode" or not.

pyplot.show	Display all open figures.
pyplot.pause	Run the GUI event loop for *interval* seconds.

Interactive mode controls:

- whether created figures are automatically shown
- whether changes to artists automatically trigger re-drawing existing figures
- when *pyplot.show()* returns if given no arguments: immediately, or after all of

the figures have been closed

If in interactive mode:

- newly created figures will be displayed immediately
- figures will automatically redraw when elements are changed
- *pyplot.show()* displays the figures and immediately returns

If not in interactive mode:

- **newly created figures and changes to figures are not displayed until**
  - *pyplot.show()* is called
  - *pyplot.pause()* is called
  - *FigureCanvasBase.flush_events()* is called
- *pyplot.show()* runs the GUI event loop and does not return until all the plot windows are closed

If you are in non-interactive mode (or created figures while in non-interactive mode) you may need to explicitly call *pyplot.show* to display the windows on your screen. If you only want to run the GUI event loop for a fixed amount of time, you can use *pyplot.pause*. This will block the progress of your code as if you had called `time.sleep`, ensure the current window is shown and re-drawn if needed, and run the GUI event loop for the specified period of time.

The GUI event loop being integrated with your command prompt and the figures being in interactive mode are independent of each other. If you use *pyplot.ion* but have not arranged for the event loop integration, your figures will appear but will not be interactive while the prompt is waiting for input. You will not be able to pan/zoom and the figure may not even render (the window might appear black, transparent, or as a snapshot of the desktop under it). Conversely, if you configure the event loop integration, displayed figures will be responsive while waiting for input at the prompt, regardless of pyplot's "interactive mode".

No matter what combination of interactive mode setting and event loop integration, figures will be responsive if you use `pyplot.show(block=True)`, *pyplot.pause*, or run the GUI main loop in some other way.

Warning: Using *figure.Figure.show* it is possible to display a figure on the screen without starting the event loop and without being in interactive mode. This may work (depending on the GUI toolkit) but will likely result in a non-responsive figure.

3.5 Default UI

The windows created by *pyplot* have an interactive toolbar with navigation buttons and a readout of the data values the cursor is pointing at. A number of helpful keybindings are registered by default.

3.5.1 Navigation Keyboard Shortcuts

The following table holds all the default keys, which can be overwritten by use of your *matplotlibrc*.

Command	Default key binding and rcParam
Home/Reset	`rcParams["keymap.home"]` (default: `['h', 'r', 'home']`)
Back	`rcParams["keymap.back"]` (default: `['left', 'c', 'backspace', 'MouseButton.BACK']`)
Forward	`rcParams["keymap.forward"]` (default: `['right', 'v', 'MouseButton.FORWARD']`)
Pan/Zoom	`rcParams["keymap.pan"]` (default: `['p']`)
Zoom-to-rect	`rcParams["keymap.zoom"]` (default: `['o']`)
Save	`rcParams["keymap.save"]` (default: `['s', 'ctrl+s']`)
Toggle fullscreen	`rcParams["keymap.fullscreen"]` (default: `['f', 'ctrl+f']`)
Toggle major grids	`rcParams["keymap.grid"]` (default: `['g']`)
Toggle minor grids	`rcParams["keymap.grid_minor"]` (default: `['G']`)
Toggle x axis scale (log/linear)	`rcParams["keymap.xscale"]` (default: `['k', 'L']`)
Toggle y axis scale (log/linear)	`rcParams["keymap.yscale"]` (default: `['l']`)
Close Figure	`rcParams["keymap.quit"]` (default: `['ctrl+w', 'cmd+w', 'q']`)
Constrain pan/zoom to x axis	hold **x** when panning/zooming with mouse
Constrain pan/zoom to y axis	hold **y** when panning/zooming with mouse
Preserve aspect ratio	hold **CONTROL** when panning/zooming with mouse

3.6 Other Python prompts

Interactive mode works in the default Python prompt:

```
>>> import matplotlib.pyplot as plt
>>> plt.ion()
>>>
```

however this does not ensure that the event hook is properly installed and your figures may not be responsive. Please consult the documentation of your GUI toolkit for details.

3.6.1 Jupyter Notebooks / Lab

Note: To get the interactive functionality described here, you must be using an interactive backend. The default backend in notebooks, the inline backend, is not. `backend_inline` renders the figure once and inserts a static image into the notebook when the cell is executed. Because the images are static, they can not be panned / zoomed, take user input, or be updated from other cells.

To get interactive figures in the 'classic' notebook or Jupyter lab, use the ipympl backend (must be installed separately) which uses the **ipywidget** framework. If `ipympl` is installed use the magic:

```
%matplotlib widget
```

to select and enable it.

If you only need to use the classic notebook, you can use

```
%matplotlib notebook
```

which uses the *backend_nbagg* backend provided by Matplotlib; however, nbagg does not work in Jupyter Lab.

GUIs + Jupyter

You can also use one of the non-`ipympl` GUI backends in a Jupyter Notebook. If you are running your Jupyter kernel locally, the GUI window will spawn on your desktop adjacent to your web browser. If you run your notebook on a remote server, the kernel will try to open the GUI window on the remote computer. Unless you have arranged to forward the xserver back to your desktop, you will not be able to see or interact with the window. It may also raise an exception.

3.6.2 PyCharm, Spyder, and VSCode

Many IDEs have built-in integration with Matplotlib, please consult their documentation for configuration details.

CHAPTER

FOUR

WHAT'S NEW?

CHAPTER

FIVE

WHAT'S NEW IN MATPLOTLIB 3.3.0

For a list of all of the issues and pull requests since the last revision, see the *GitHub Stats*.

Table of Contents

- *Other changes*
  - *New `Axes.axline` method*
  - *`imshow` now coerces 3D arrays with depth 1 to 2D*
  - *Better control of `Axes.pie` normalization*
  - *Dates use a modern epoch*
  - *Lines now accept `MarkerStyle` instances as input*
- *Fonts*
  - *Simple syntax to select fonts by absolute path*
  - *Improved font weight detection*
- *rcParams improvements*
  - *`matplotlib.rc_context` can be used as a decorator*
  - *rcParams for controlling default "raise window" behavior*
  - *Add generalized `mathtext.fallback` to rcParams*
  - *Add `contour.linewidth` to rcParams*
- *3D Axes improvements*
  - *`Axes3D` no longer distorts the 3D plot to match the 2D aspect ratio*
  - *3D axes now support minor ticks*
  - *Home/Forward/Backward buttons now work with 3D axes*
- *Interactive tool improvements*
  - *More consistent toolbar behavior across backends*
  - *Toolbar icons are now styled for dark themes*
  - *Cursor text now uses a number of significant digits matching pointing precision*
  - *GTK / Qt zoom rectangle now black and white*
  - *Event handler simplifications*
- *Functions to compute a Path's size*
  - *Better interface for Path segment iteration*
  - *Fixed bug that computed a Path's Bbox incorrectly*
- *Backend-specific improvements*
  - *`savefig()` gained a* backend *keyword argument*
  - *The SVG backend can now render hatches with transparency*
  - *SVG supports URLs on more artists*

* *Images in SVG will no longer be blurred in some viewers*
* *Saving SVG now supports adding metadata*
* *Saving PDF metadata via PGF now consistent with PDF backend*
* *NbAgg and WebAgg no longer use jQuery & jQuery UI*

5.1 Figure and Axes creation / management

5.1.1 Provisional API for composing semantic axes layouts from text or nested lists

The *Figure* class has a provisional method to generate complex grids of named *axes.Axes* based on nested list input or ASCII art:

```
axd = plt.figure(constrained_layout=True).subplot_mosaic(
    [['.', 'histx'],
     ['histy', 'scat']]
)
for k, ax in axd.items():
    ax.text(0.5, 0.5, k,
            ha='center', va='center', fontsize=36,
            color='darkgrey')
```

or as a string (with single-character Axes labels):

```
axd = plt.figure(constrained_layout=True).subplot_mosaic(
    """
    TTE
    L.E
    """)
for k, ax in axd.items():
    ax.text(0.5, 0.5, k,
            ha='center', va='center', fontsize=36,
            color='darkgrey')
```

See *Complex and semantic figure composition* for more details and examples.

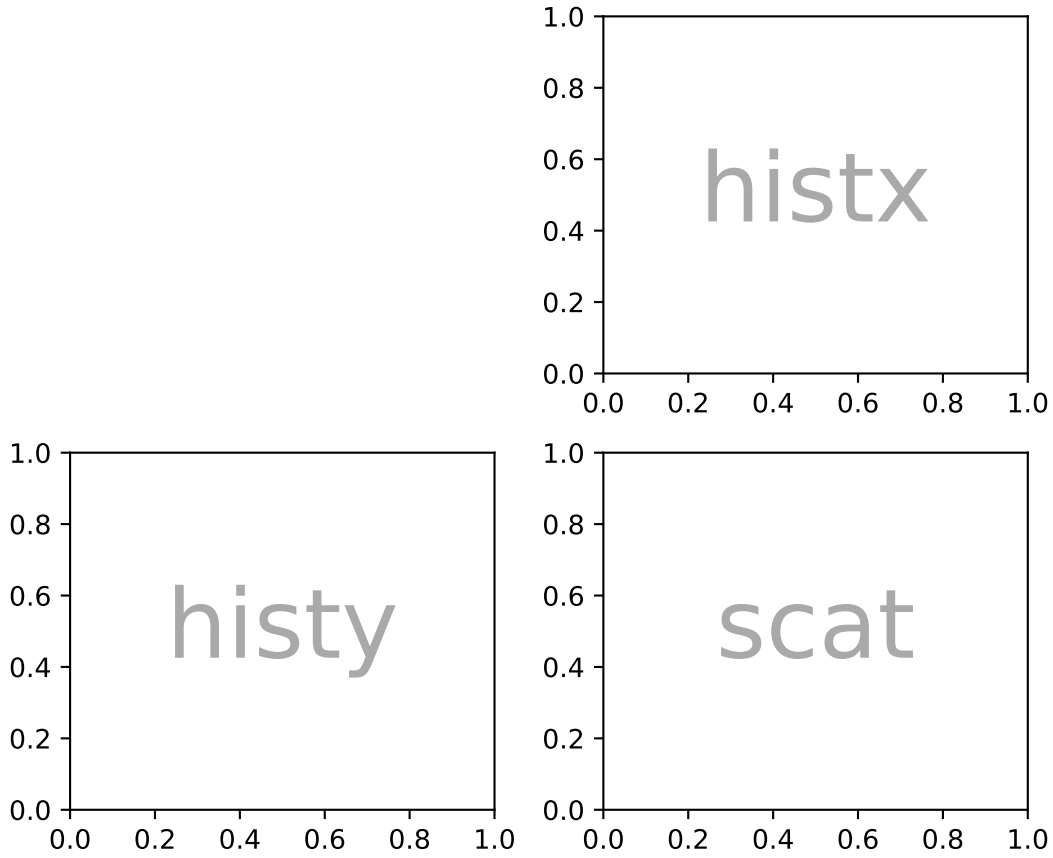
5.1.2 GridSpec.subplots()

The *GridSpec* class gained a *subplots* method, so that one can write

```
fig.add_gridspec(2, 2, height_ratios=[3, 1]).subplots()
```

as an alternative to

```
fig.subplots(2, 2, gridspec_kw={"height_ratios": [3, 1]})
```

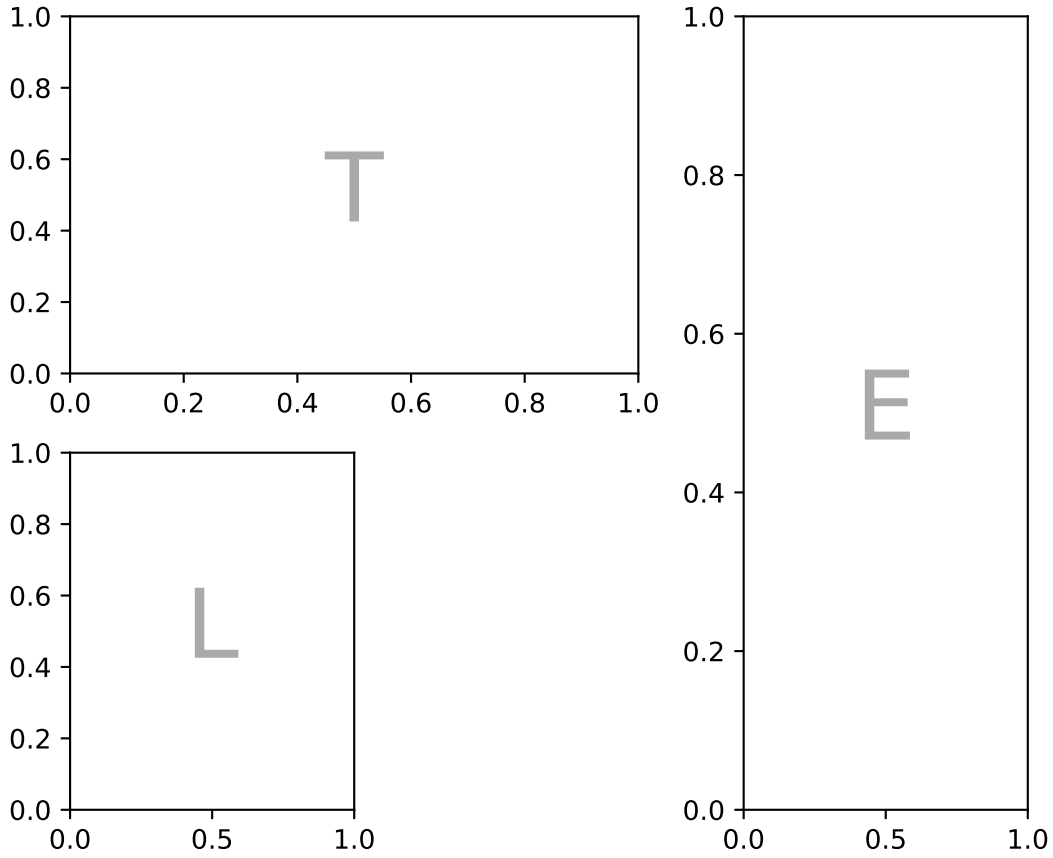


5.1.3 New `Axes.sharex`, `Axes.sharey` methods

These new methods allow sharing axes *immediately* after creating them. Note that behavior is indeterminate if axes are not shared immediately after creation.

For example, they can be used to selectively link some axes created all together using *`subplot_mosaic`*:

```
fig = plt.figure(constrained_layout=True)
axd = fig.subplot_mosaic([['.', 'histx'], ['histy', 'scat']],
                         gridspec_kw={'width_ratios': [1, 7],
                                      'height_ratios': [2, 7]})

axd['histx'].sharex(axd['scat'])
axd['histy'].sharey(axd['scat'])
```



5.1.4 tight_layout now supports suptitle

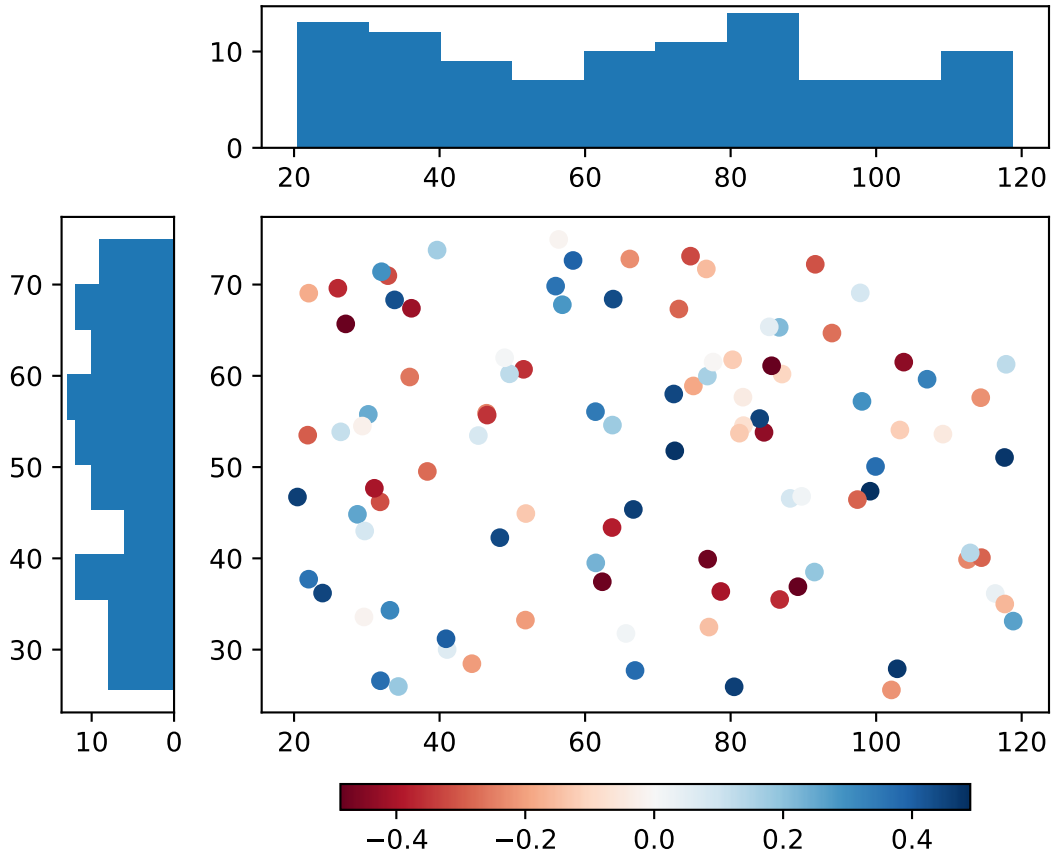
Previous versions did not consider *Figure.suptitle*, so it may overlap with other artists after calling *tight_layout*:

From now on, the `suptitle` will be considered:

5.1.5 Setting axes box aspect

It is now possible to set the aspect of an axes box directly via *set_box_aspect*. The box aspect is the ratio between axes height and axes width in physical units, independent of the data limits. This is useful to, e.g., produce a square plot, independent of the data it contains, or to have a non-image plot with the same axes dimensions next to an image plot with fixed (data-)aspect.

For use cases check out the Axes box aspect example.



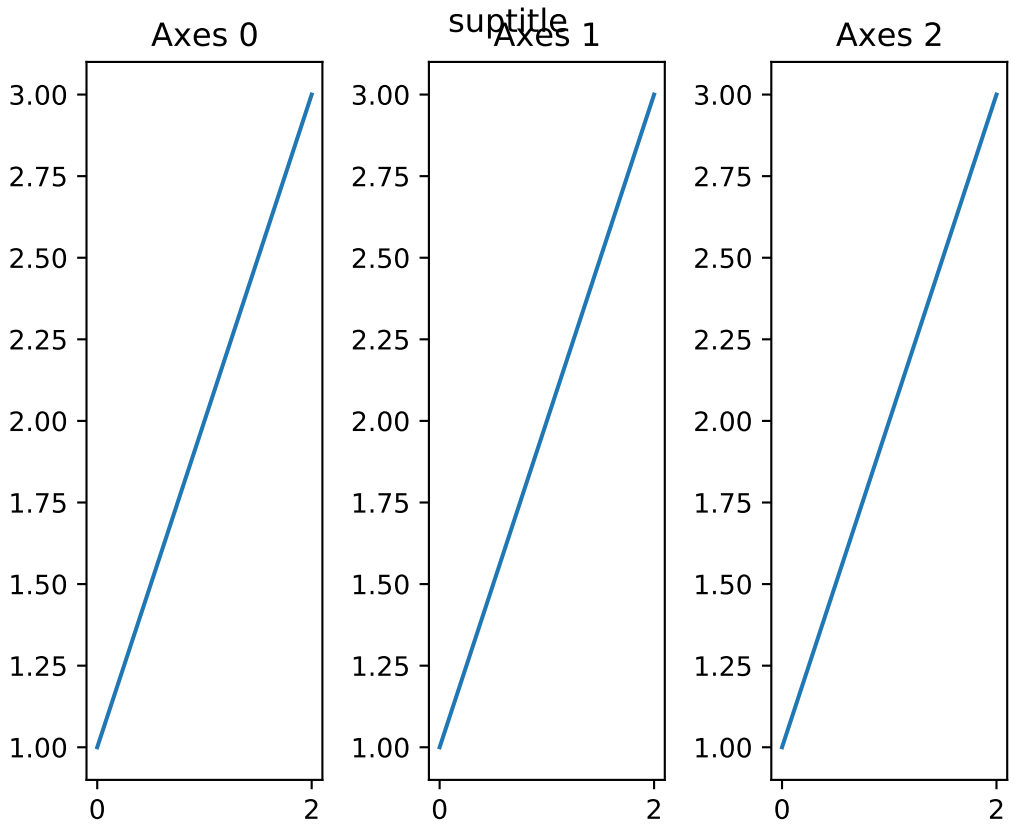
5.2 Colors and colormaps

5.2.1 Turbo colormap

Turbo is an improved rainbow colormap for visualization, created by the Google AI team for computer vision and machine learning. Its purpose is to display depth and disparity data. Please see the Google AI Blog for further details.

5.2.2 `colors.BoundaryNorm` supports *extend* keyword argument

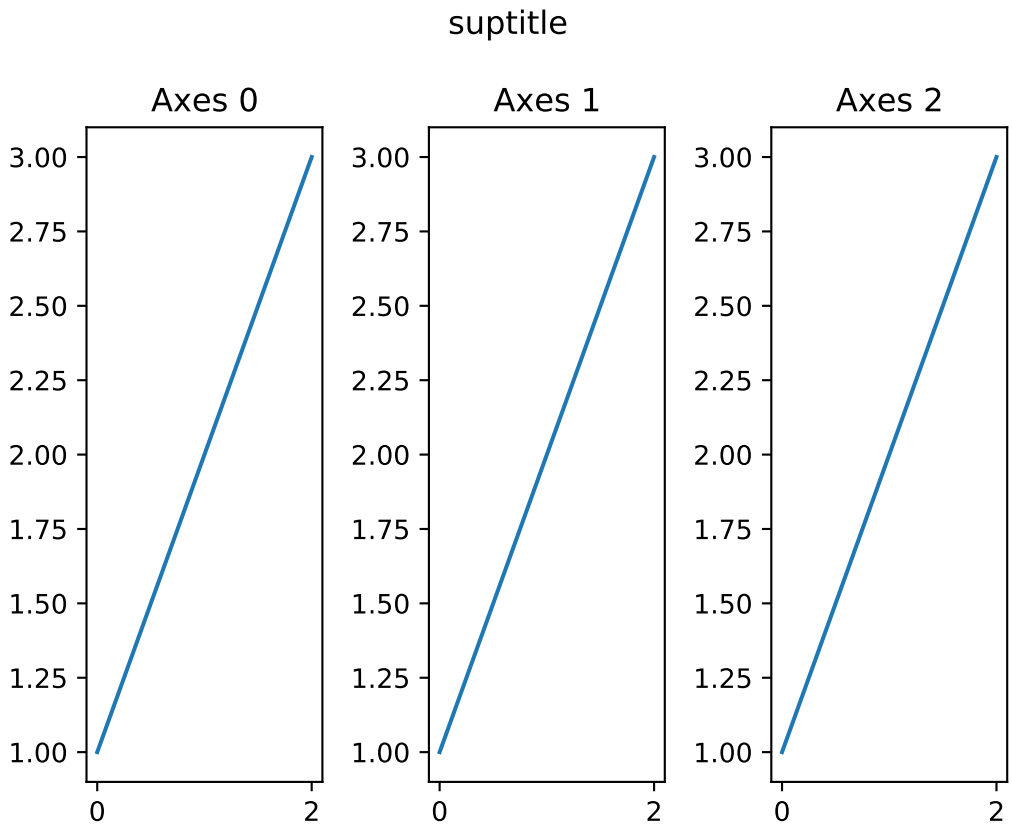
BoundaryNorm now has an *extend* keyword argument, analogous to *extend* in *contourf*. When set to 'both', 'min', or 'max', it maps the corresponding out-of-range values to *Colormap* lookup-table indices near the appropriate ends of their range so that the colors for out-of range values are adjacent to, but distinct from, their in-range neighbors. The colorbar inherits the *extend* argument from the norm, so with `extend='both'`, for example, the colorbar will have triangular extensions for out-of-range values with colors that differ from adjacent in-range colors.



5.2.3 Text color for legend labels

The text color of legend labels can now be set by passing a parameter `labelcolor` to *`legend`*. The `labelcolor` keyword can be:

- A single color (either a string or RGBA tuple), which adjusts the text color of all the labels.
- A list or tuple, allowing the text color of each label to be set individually.
- `linecolor`, which sets the text color of each label to match the corresponding line color.
- `markerfacecolor`, which sets the text color of each label to match the corresponding marker face color.
- `markeredgecolor`, which sets the text color of each label to match the corresponding marker edge color.



5.2.4 Pcolor and Pcolormesh now accept `shading='nearest'` and `'auto'`

Previously *axes.Axes.pcolor* and *axes.Axes.pcolormesh* handled the situation where x and y have the same (respective) size as C by dropping the last row and column of C, and x and y are regarded as the edges of the remaining rows and columns in C. However, many users want x and y centered on the rows and columns of C.

To accommodate this, `shading='nearest'` and `shading='auto'` are new allowed strings for the *shading* keyword argument. `'nearest'` will center the color on x and y if x and y have the same dimensions as C (otherwise an error will be thrown). `shading='auto'` will choose 'flat' or 'nearest' based on the size of X, Y, C.

If `shading='flat'` then X, and Y should have dimensions one larger than C. If X and Y have the same dimensions as C, then the previous behavior is used and the last row and column of C are dropped, and a DeprecationWarning is emitted.

Users can also specify this by the new `rcParams["pcolor.shading"]` (default: `'flat'`) in their `.matplotlibrc` or via *rcParams*.

See pcolormesh for examples.

5.3 Titles, ticks, and labels

5.3.1 Align labels to Axes edges

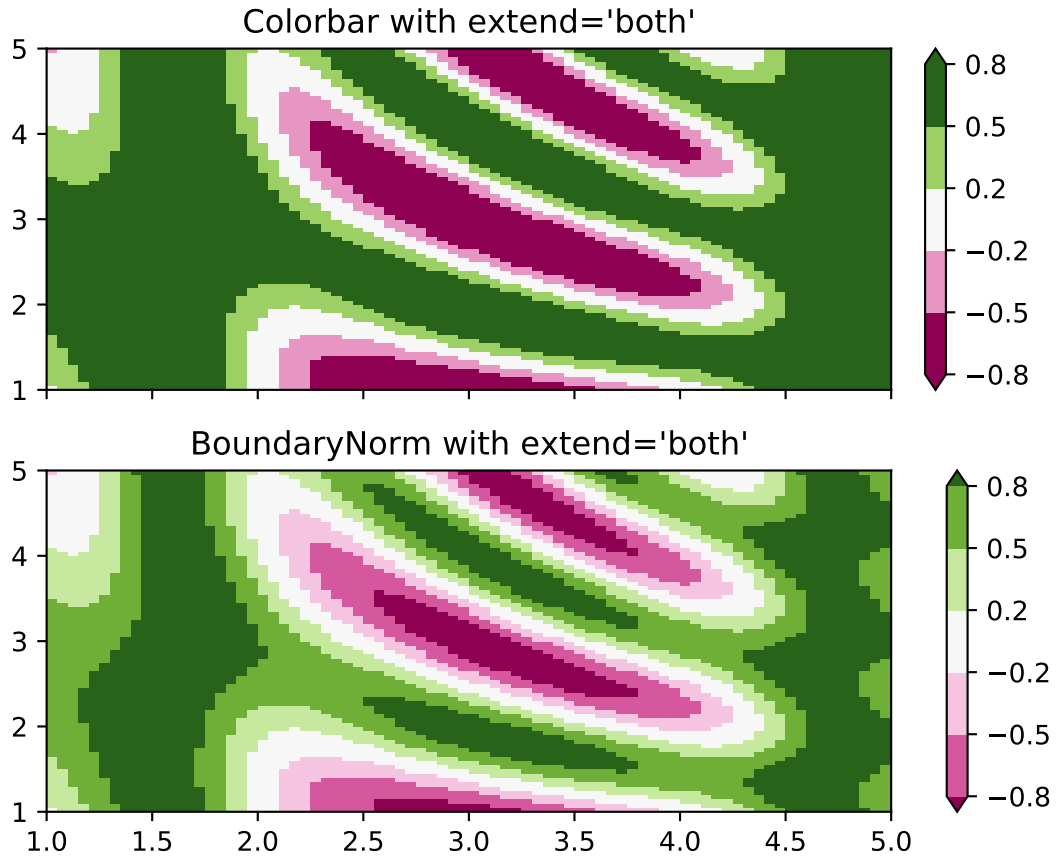
set_xlabel, *set_ylabel* and *ColorbarBase.set_label* support a parameter `loc` for simplified positioning. For the xlabel, the supported values are 'left', 'center', or 'right'. For the ylabel, the supported values are 'bottom', 'center', or 'top'.

The default is controlled via `rcParams["xaxis.labelposition"]` and `rcParams["yaxis.labelposition"]`; the Colorbar label takes the rcParam based on its orientation.

5.3.2 Allow tick formatters to be set with str or function inputs

set_major_formatter and *set_minor_formatter* now accept `str` or function inputs in addition to *Formatter* instances. For a `str` a *StrMethodFormatter* is automatically generated and used. For a function a *FuncFormatter* is automatically generated and used. In other words,

```
ax.xaxis.set_major_formatter('{x} km')
ax.xaxis.set_minor_formatter(lambda x, pos: str(x-5))
```

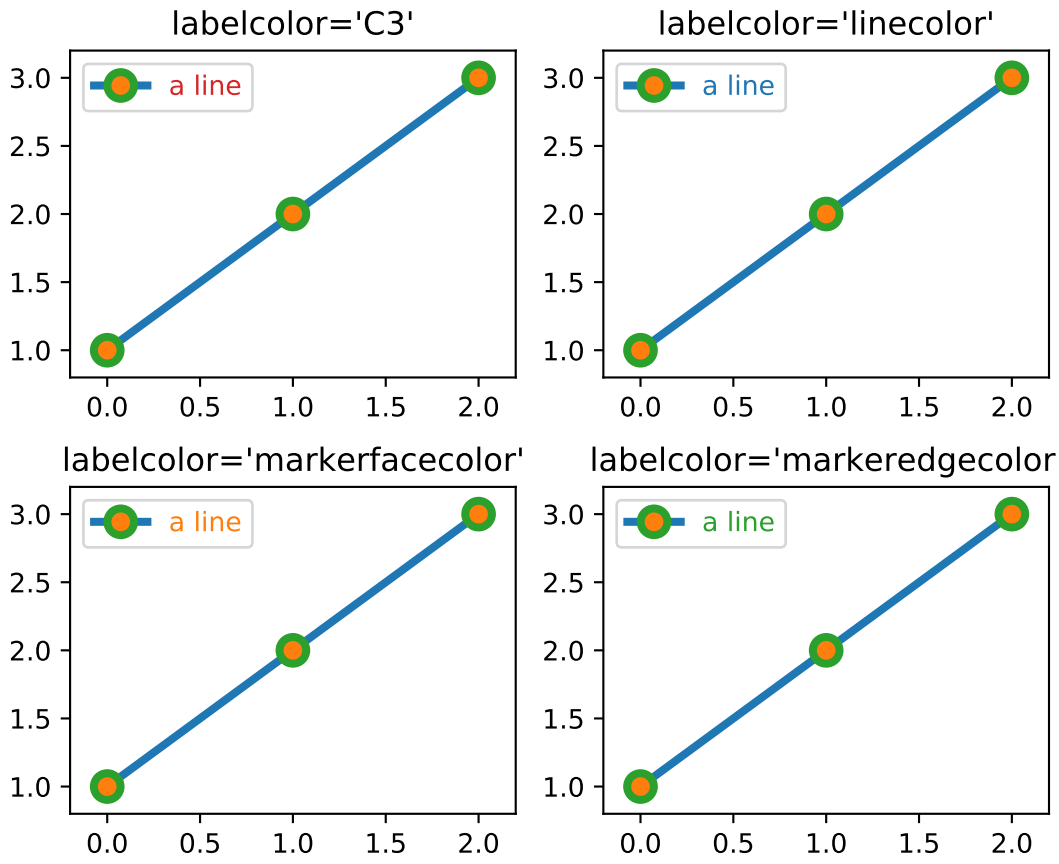


are shortcuts for:

```
import matplotlib.ticker as mticker

ax.xaxis.set_major_formatter(mticker.StrMethodFormatter('{x} km'))
ax.xaxis.set_minor_formatter(
    mticker.FuncFormatter(lambda x, pos: str(x-5))
```

5.3.3 `Axes.set_title` gains a *y* keyword argument to control auto positioning

set_title tries to auto-position the title to avoid any decorators on the top x-axis. This is not always desirable so now *y* is an explicit keyword argument of *set_title*. It defaults to *None* which means to use auto-positioning. If a value is supplied (i.e. the pre-3.0 default was `y=1.0`) then auto-positioning is turned off. This can also be set with the new rcParameter `rcParams["axes.titley"]` (default: `None`).

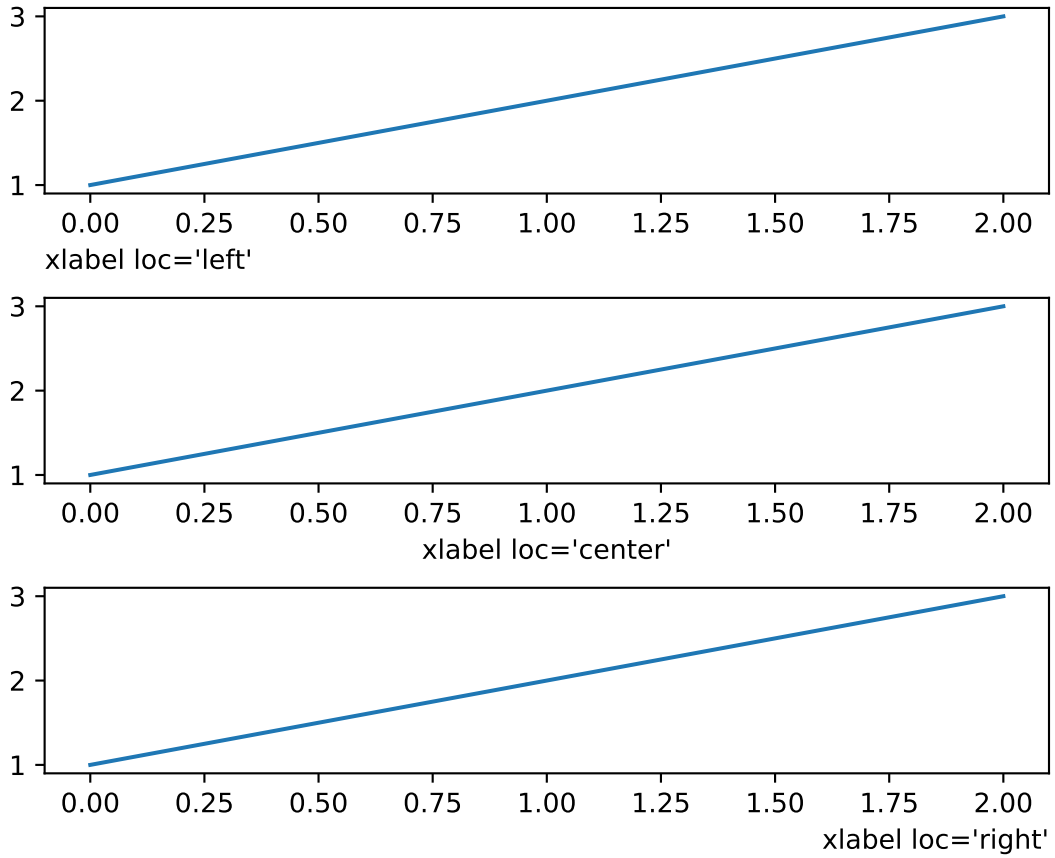


5.3.4 Offset text is now set to the top when using `axis.tick_top()`

Solves the issue that the power indicator (e.g., 1e4) stayed on the bottom, even if the ticks were on the top.

5.3.5 Set zorder of contour labels

clabel now accepts a *zorder* keyword argument making it easier to set the *zorder* of contour labels. If not specified, the default *zorder* of clabels used to always be 3 (i.e. the default *zorder* of *Text*) irrespective of the *zorder* passed to *contour*/*contourf*. The new default *zorder* for clabels has been changed to (`2 + zorder` passed to *contour* / *contourf*).

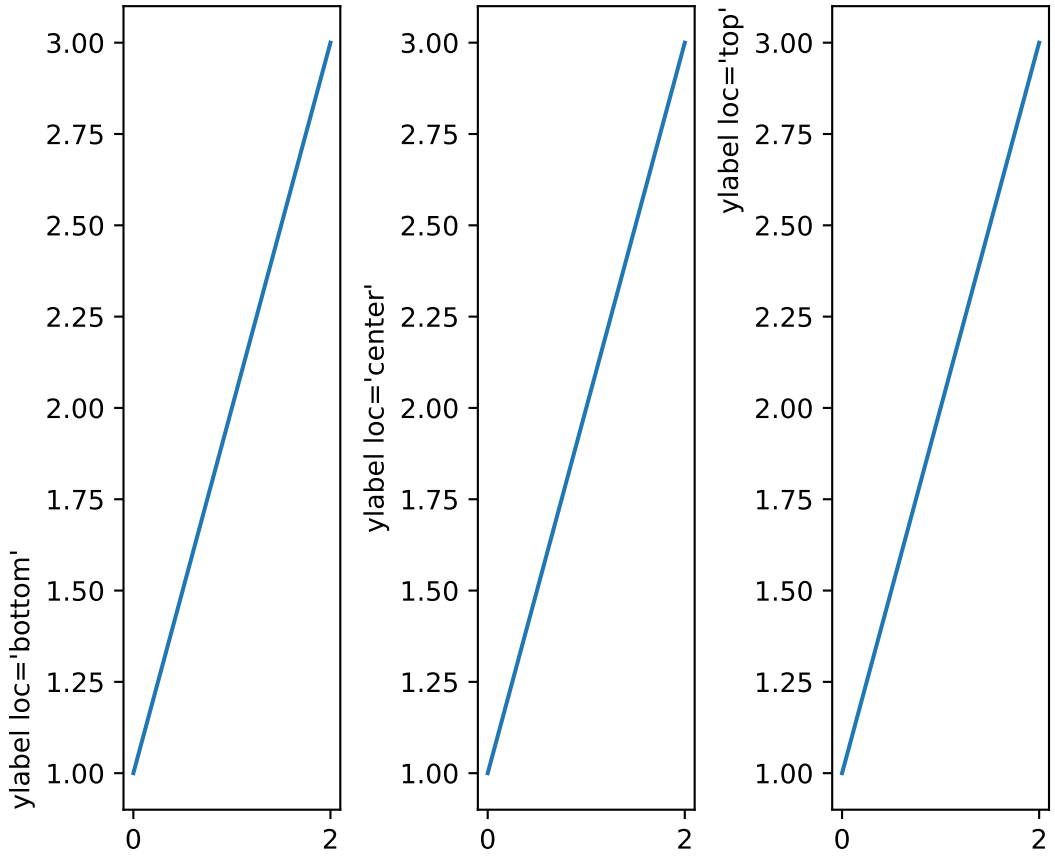


5.4 Other changes

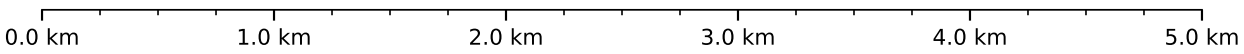
5.4.1 New `Axes.axline` method

A new *axline* method has been added to draw infinitely long lines that pass through two points.

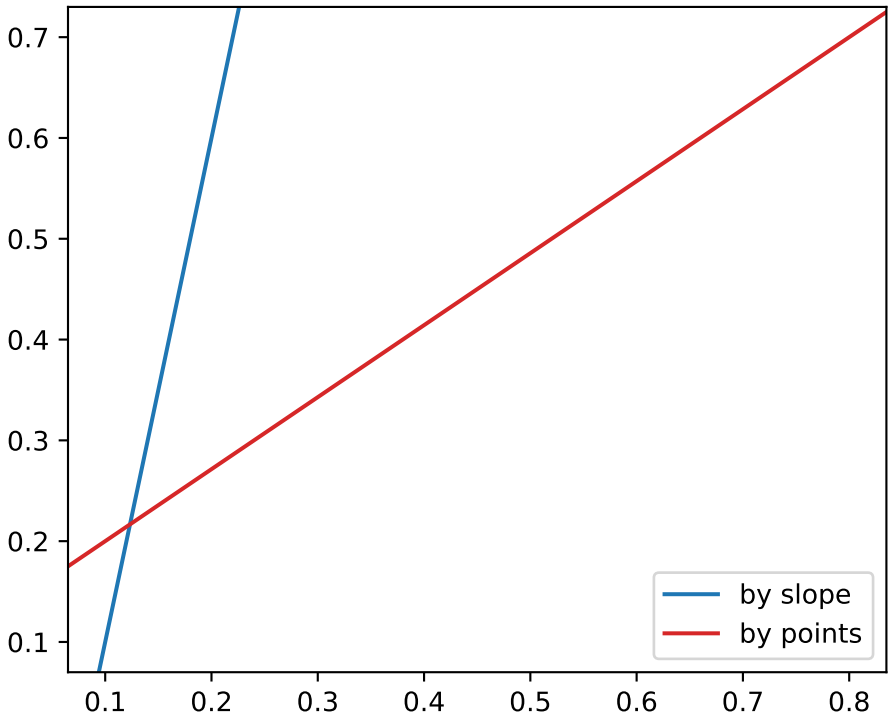
```
fig, ax = plt.subplots()

ax.axline((.1, .1), slope=5, color='C0', label='by slope')
ax.axline((.1, .2), (.8, .7), color='C3', label='by points')

ax.legend()
```




```
ax.xaxis.set_major_formatter('{x} km')
```




```
ax.xaxis.set_major_formatter(lambda x, pos: str(x-5))
```



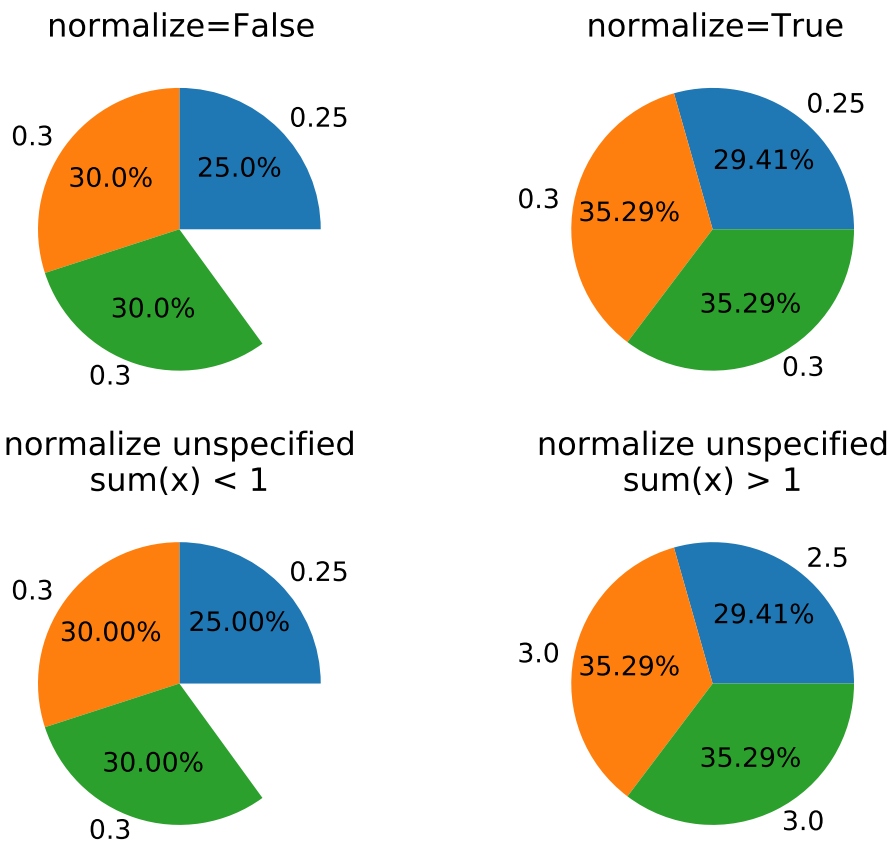
5.4.2 `imshow` now coerces 3D arrays with depth 1 to 2D

Starting from this version arrays of size MxNx1 will be coerced into MxN for displaying. This means commands like `plt.imshow(np.random.rand(3, 3, 1))` will no longer return an error message that the image shape is invalid.

5.4.3 Better control of `Axes.pie` normalization

Previously, *`Axes.pie`* would normalize its input *x* if `sum(x) > 1`, but would do nothing if the sum were less than 1. This can be confusing, so an explicit keyword argument *normalize* has been added. By default, the old behavior is preserved.

By passing *normalize*, one can explicitly control whether any rescaling takes place or whether partial pies should be created. If normalization is disabled, and `sum(x) > 1`, then an error is raised.



5.4.4 Dates use a modern epoch

Matplotlib converts dates to days since an epoch using *dates.date2num* (via *matplotlib.units*). Previously, an epoch of `0000-12-31T00:00:00` was used so that `0001-01-01` was converted to 1.0. An epoch so distant in the past meant that a modern date was not able to preserve microseconds because 2000 years times the 2^(-52) resolution of a 64-bit float gives 14 microseconds.

Here we change the default epoch to the more reasonable UNIX default of `1970-01-01T00:00:00` which for a modern date has 0.35 microsecond resolution. (Finer resolution is not possible because we rely on `datetime.datetime` for the date locators). Access to the epoch is provided by *get_epoch*, and there is a new `rcParams["date.epoch"]` (default: `'1970-01-01T00:00:00'`) rcParam. The user may also call *set_epoch*, but it must be set *before* any date conversion or plotting is used.

If you have data stored as ordinal floats in the old epoch, you can convert them to the new ordinal using the following formula:

```
new_ordinal = old_ordinal + mdates.date2num(np.datetime64('0000-12-31'))
```

5.4.5 Lines now accept `MarkerStyle` instances as input

Similar to *scatter*, *plot* and *Line2D* now accept *MarkerStyle* instances as input for the *marker* parameter:

```
plt.plot(..., marker=matplotlib.markers.MarkerStyle("D"))
```

5.5 Fonts

5.5.1 Simple syntax to select fonts by absolute path

Fonts can now be selected by passing an absolute `pathlib.Path` to the *font* keyword argument of *Text*.

5.5.2 Improved font weight detection

Matplotlib is now better able to determine the weight of fonts from their metadata, allowing to differentiate between fonts within the same family more accurately.

5.6 rcParams improvements

5.6.1 `matplotlib.rc_context` can be used as a decorator

matplotlib.rc_context can now be used as a decorator (technically, it is now implemented as a `contextlib.contextmanager`), e.g.,

```
@rc_context({"lines.linewidth": 2})
def some_function(...):
    ...
```

5.6.2 rcParams for controlling default "raise window" behavior

The new config option `rcParams["figure.raise_window"]` (default: `True`) allows disabling of the raising of the plot window when calling *show* or *pause*. The `MacOSX` backend is currently not supported.

5.6.3 Add generalized `mathtext.fallback` to rcParams

New `rcParams["mathtext.fallback"]` (default: `'cm'`) rcParam. Takes "cm", "stix", "stixsans" or "none" to turn fallback off. The rcParam *mathtext.fallback_to_cm* is deprecated, but if used, will override new fallback.

5.6.4 Add `contour.linewidth` to rcParams

The new config option `rcParams["contour.linewidth"]` (default: `None`) allows to control the default line width of contours as a float. When set to `None`, the line widths fall back to `rcParams["lines.linewidth"]` (default: `1.5`). The config value is overridden as usual by the *linewidths* argument passed to *contour* when it is not set to `None`.

5.7 3D Axes improvements

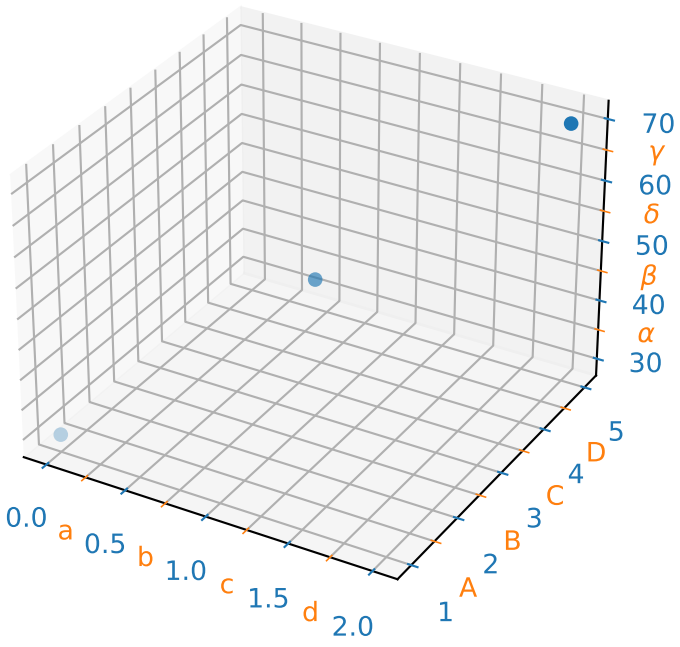
5.7.1 `Axes3D` no longer distorts the 3D plot to match the 2D aspect ratio

Plots made with *Axes3D* were previously stretched to fit a square bounding box. As this stretching was done after the projection from 3D to 2D, it resulted in distorted images if non-square bounding boxes were used. As of 3.3, this no longer occurs.

Currently, modes of setting the aspect (via *set_aspect*) in data space are not supported for Axes3D but may be in the future. If you want to simulate having equal aspect in data space, set the ratio of your data limits to match the value of *get_box_aspect*. To control these ratios use the *set_box_aspect* method which accepts the ratios as a 3-tuple of X:Y:Z. The default aspect ratio is 4:4:3.

5.7.2 3D axes now support minor ticks

```
ax = plt.figure().add_subplot(projection='3d')

ax.scatter([0, 1, 2], [1, 3, 5], [30, 50, 70])

ax.set_xticks([0.25, 0.75, 1.25, 1.75], minor=True)
ax.set_xticklabels(['a', 'b', 'c', 'd'], minor=True)

ax.set_yticks([1.5, 2.5, 3.5, 4.5], minor=True)
ax.set_yticklabels(['A', 'B', 'C', 'D'], minor=True)

ax.set_zticks([35, 45, 55, 65], minor=True)
ax.set_zticklabels([r'$\alpha$', r'$\beta$', r'$\delta$', r'$\gamma$'],
                   minor=True)

ax.tick_params(which='major', color='C0', labelcolor='C0', width=5)
ax.tick_params(which='minor', color='C1', labelcolor='C1', width=3)
```



5.7.3 Home/Forward/Backward buttons now work with 3D axes

5.8 Interactive tool improvements

5.8.1 More consistent toolbar behavior across backends

Toolbar features are now more consistent across backends. The history buttons will auto-disable when there is no further action in a direction. The pan and zoom buttons will be marked active when they are in use.

In NbAgg and WebAgg, the toolbar buttons are now grouped similarly to other backends. The WebAgg toolbar now uses the same icons as other backends.

5.8.2 Toolbar icons are now styled for dark themes

On dark themes, toolbar icons will now be inverted. When using the GTK3Agg backend, toolbar icons are now symbolic, and both foreground and background colors will follow the theme. Tooltips should also behave correctly.

5.8.3 Cursor text now uses a number of significant digits matching pointing precision

Previously, the x/y position displayed by the cursor text would usually include far more significant digits than the mouse pointing precision (typically one pixel). This is now fixed for linear scales.

5.8.4 GTK / Qt zoom rectangle now black and white

This makes it visible even over a dark background.

5.8.5 Event handler simplifications

The *backend_bases.key_press_handler* and *backend_bases.button_press_handler* event handlers can now be directly connected to a canvas with `canvas.mpl_connect("key_press_event", key_press_handler)` and `canvas.mpl_connect("button_press_event", button_press_handler)`, rather than having to write wrapper functions that fill in the (now optional) *canvas* and *toolbar* parameters.

5.9 Functions to compute a Path's size

Various functions were added to *BezierSegment* and *Path* to allow computation of the shape/size of a *Path* and its composite Bezier curves.

In addition to the fixes below, *BezierSegment* has gained more documentation and usability improvements, including properties that contain its dimension, degree, control_points, and more.

5.9.1 Better interface for Path segment iteration

iter_bezier iterates through the *BezierSegment*'s that make up the Path. This is much more useful typically than the existing *iter_segments* function, which returns the absolute minimum amount of information possible to reconstruct the Path.

5.9.2 Fixed bug that computed a Path's Bbox incorrectly

Historically, *get_extents* has always simply returned the Bbox of a curve's control points, instead of the Bbox of the curve itself. While this is a correct upper bound for the path's extents, it can differ dramatically from the Path's actual extents for non-linear Bezier curves.

5.10 Backend-specific improvements

5.10.1 `savefig()` gained a *backend* keyword argument

The *backend* keyword argument to `savefig` can now be used to pick the rendering backend without having to globally set the backend; e.g., one can save PDFs using the pgf backend with `savefig("file.pdf", backend="pgf")`.

5.10.2 The SVG backend can now render hatches with transparency

The SVG backend now respects the hatch stroke alpha. Useful applications are, among others, semi-transparent hatches as a subtle way to differentiate columns in bar plots.

5.10.3 SVG supports URLs on more artists

URLs on more artists (i.e., from *Artist.set_url*) will now be saved in SVG files, namely, `Ticks` and `Line2Ds` are now supported.

5.10.4 Images in SVG will no longer be blurred in some viewers

A style is now supplied to images without interpolation (`imshow(..., interpolation='none')`) so that SVG image viewers will no longer perform interpolation when rendering themselves.

5.10.5 Saving SVG now supports adding metadata

When saving SVG files, metadata can now be passed which will be saved in the file using Dublin Core and RDF. A list of valid metadata can be found in the documentation for *FigureCanvasSVG.print_svg*.

5.10.6 Saving PDF metadata via PGF now consistent with PDF backend

When saving PDF files using the PGF backend, passed metadata will be interpreted in the same way as with the PDF backend. Previously, this metadata was only accepted by the PGF backend when saving a multi-page PDF with *backend_pgf.PdfPages*, but is now allowed when saving a single figure, as well.

5.10.7 NbAgg and WebAgg no longer use jQuery & jQuery UI

Instead, they are implemented using vanilla JavaScript. Please report any issues with browsers.

CHAPTER

SIX

HISTORY

Note: The following introductory text was written in 2008 by John D. Hunter (1968-2012), the original author of Matplotlib.

Matplotlib is a library for making 2D plots of arrays in Python. Although it has its origins in emulating the MATLAB graphics commands, it is independent of MATLAB, and can be used in a Pythonic, object oriented way. Although Matplotlib is written primarily in pure Python, it makes heavy use of NumPy and other extension code to provide good performance even for large arrays.

Matplotlib is designed with the philosophy that you should be able to create simple plots with just a few commands, or just one! If you want to see a histogram of your data, you shouldn't need to instantiate objects, call methods, set properties, and so on; it should just work.

For years, I used to use MATLAB exclusively for data analysis and visualization. MATLAB excels at making nice looking plots easy. When I began working with EEG data, I found that I needed to write applications to interact with my data, and developed an EEG analysis application in MATLAB. As the application grew in complexity, interacting with databases, http servers, manipulating complex data structures, I began to strain against the limitations of MATLAB as a programming language, and decided to start over in Python. Python more than makes up for all of MATLAB's deficiencies as a programming language, but I was having difficulty finding a 2D plotting package (for 3D VTK more than exceeds all of my needs).

When I went searching for a Python plotting package, I had several requirements:

- Plots should look great - publication quality. One important requirement for me is that the text looks good (antialiased, etc.)
- Postscript output for inclusion with TeX documents
- Embeddable in a graphical user interface for application development
- Code should be easy enough that I can understand it and extend it
- Making plots should be easy

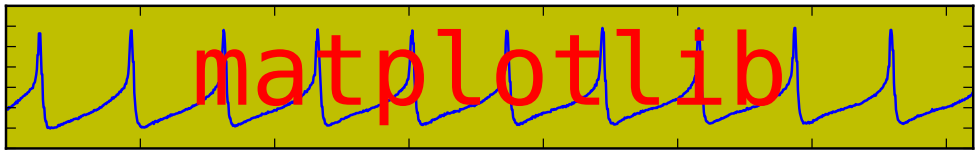
Finding no package that suited me just right, I did what any self-respecting Python programmer would do: rolled up my sleeves and dived in. Not having any real expe-

rience with computer graphics, I decided to emulate MATLAB's plotting capabilities because that is something MATLAB does very well. This had the added advantage that many people have a lot of MATLAB experience, and thus they can quickly get up to steam plotting in python. From a developer's perspective, having a fixed user interface (the pylab interface) has been very useful, because the guts of the code base can be redesigned without affecting user code.

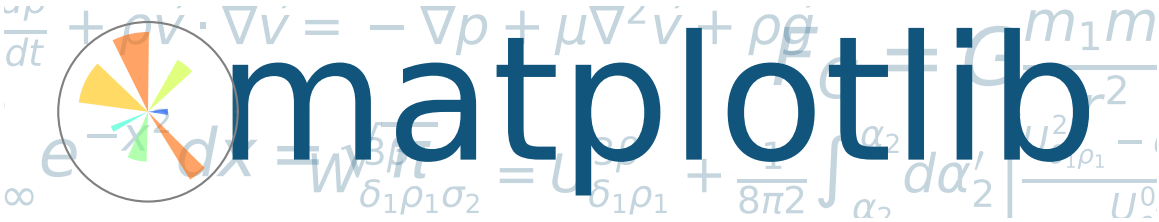
The Matplotlib code is conceptually divided into three parts: the *pylab interface* is the set of functions provided by `pylab` which allow the user to create plots with code quite similar to MATLAB figure generating code (*Pyplot tutorial*). The *Matplotlib frontend* or *Matplotlib API* is the set of classes that do the heavy lifting, creating and managing figures, text, lines, plots and so on (*Artist tutorial*). This is an abstract interface that knows nothing about output. The *backends* are device-dependent drawing devices, aka renderers, that transform the frontend representation to hardcopy or a display device (*What is a backend?*). Example backends: PS creates PostScript® hardcopy, SVG creates Scalable Vector Graphics hardcopy, Agg creates PNG output using the high quality Anti-Grain Geometry library that ships with Matplotlib, GTK embeds Matplotlib in a Gtk+ application, GTKAgg uses the Anti-Grain renderer to create a figure and embed it in a Gtk+ application, and so on for PDF, WxWidgets, Tkinter, etc.

Matplotlib is used by many people in many different contexts. Some people want to automatically generate PostScript files to send to a printer or publishers. Others deploy Matplotlib on a web application server to generate PNG output for inclusion in dynamically-generated web pages. Some use Matplotlib interactively from the Python shell in Tkinter on Windows. My primary use is to embed Matplotlib in a Gtk+ EEG application that runs on Windows, Linux and Macintosh OS X.

Matplotlib's original logo (2003 -- 2008).



Matplotlib logo (2008 - 2015).



CHAPTER

SEVEN

GITHUB STATS

GitHub stats for 2020/09/15 - 2020/11/11 (tag: v3.3.2)

These lists are automatically generated, and may be incomplete or contain duplicates.

We closed 14 issues and merged 46 pull requests. The full list can be seen on GitHub

The following 11 authors contributed 73 commits.

- Antony Lee
- David Stansby
- Elliott Sales de Andrade
- Eric Larson
- Jody Klymak
- Jouni K. Seppänen
- Ryan May
- shevawen
- Stephen Sinclair
- Thomas A Caswell
- Tim Hoffmann

GitHub issues and pull requests:

Pull Requests (46):

- PR #18936: Backport PR #18929 on branch v3.3.x
- PR #18929: FIX: make sure scalarmappable updates are handled correctly in 3D
- PR #18928: Backport PR #18842 on branch v3.3.x (Add CPython 3.9 wheels.)
- PR #18842: Add CPython 3.9 wheels.
- PR #18921: Backport PR #18732 on branch v3.3.x (Add a ponyfill for ResizeObserver on older browsers.)
- PR #18732: Add a ponyfill for ResizeObserver on older browsers.

- PR #18886: Backport #18860 on branch v3.3.x
- PR #18860: FIX: stop deprecation message colorbar
- PR #18845: Backport PR #18839 on branch v3.3.x
- PR #18843: Backport PR #18756 on branch v3.3.x (FIX: improve date performance regression)
- PR #18850: Backport CI fixes to v3.3.x
- PR #18839: MNT: make sure we do not mutate input in Text.update
- PR #18838: Fix ax.set_xticklabels(fontproperties=fp)
- PR #18756: FIX: improve date performance regression
- PR #18787: Backport PR #18769 on branch v3.3.x
- PR #18786: Backport PR #18754 on branch v3.3.x (FIX: make sure we have more than 1 tick with small log ranges)
- PR #18754: FIX: make sure we have more than 1 tick with small log ranges
- PR #18769: Support `ax.grid(visible=<bool>)`.
- PR #18778: Backport PR #18773 on branch v3.3.x (Update to latest cibuildwheel release.)
- PR #18773: Update to latest cibuildwheel release.
- PR #18755: Backport PR #18734 on branch v3.3.x (Fix deprecation warning in GitHub Actions.)
- PR #18734: Fix deprecation warning in GitHub Actions.
- PR #18725: Backport PR #18533 on branch v3.3.x
- PR #18723: Backport PR #18584 on branch v3.3.x (Fix setting 0-timeout timer with Tornado.)
- PR #18676: Backport PR #18670 on branch v3.3.x (MNT: make certifi actually optional)
- PR #18670: MNT: make certifi actually optional
- PR #18665: Backport PR #18639 on branch v3.3.x (nbagg: Don't close figures for bubbled events.)
- PR #18639: nbagg: Don't close figures for bubbled events.
- PR #18640: Backport PR #18636 on branch v3.3.x (BLD: certifi is not a run-time dependency)
- PR #18636: BLD: certifi is not a run-time dependency
- PR #18629: Backport PR #18621 on branch v3.3.x (Fix singleshot timers in wx.)
- PR #18621: Fix singleshot timers in wx.

- PR #18607: Backport PR #18604 on branch v3.3.x (Update test image to fix Ghostscript 9.53.)
- PR #18604: Update test image to fix Ghostscript 9.53.
- PR #18584: Fix setting 0-timeout timer with Tornado.
- PR #18550: backport pr 18549
- PR #18545: Backport PR #18540 on branch v3.3.x (Call to ExitStack.push should have been ExitStack.callback.)
- PR #18549: FIX: unit-convert pcolorargs before interpolating
- PR #18540: Call to ExitStack.push should have been ExitStack.callback.
- PR #18533: Correctly remove support for stackrel.
- PR #18509: Backport PR #18505 on branch v3.3.x (Fix depth shading when edge/facecolor is none.)
- PR #18505: Fix depth shading when edge/facecolor is none.
- PR #18504: Backport PR #18500 on branch v3.3.x (BUG: Fix all-masked imshow)
- PR #18500: BUG: Fix all-masked imshow
- PR #18476: CI: skip qt, cairo, pygobject related installs on OSX on travis
- PR #18134: Build on xcode9

Issues (14):

- #18885: 3D Scatter Plot with Colorbar is not saved correctly with savefig
- #18922: pyplot.xticks(): Font property specification is not effective except 1st tick label.
- #18481: "%matplotlib notebook" not working in firefox with matplotlib 3.3.1
- #18595: Getting internal "MatplotlibDeprecationWarning: shading='flat' ..."
- #18743: from mpl 3.2.2 to 3.3.0 enormous increase in creation time
- #18317: pcolormesh: shading='nearest' and non-monotonic coordinates
- #18758: Using Axis.grid(visible=True) results in TypeError for multiple values for keyword argument
- #18638: `matplotlib>=3.3.2` breaks `ipywidgets.interact`
- #18337: Error installing matplotlib-3.3.1 using pip due to old version of certifi on conda environment
- #18620: wx backend assertion error with fig.canvas.timer.start()
- #18551: test_transparent_markers[pdf] is broken on v3.3.x Travis macOS
- #18580: Animation freezes in Jupyter notebook

- #18547: pcolormesh x-axis with datetime broken for nearest shading
- #18539: Error in Axes.redraw_in_frame in use of ExitStack: push() takes 2 positional arguments but 3 were given

7.1 Previous GitHub Stats

7.1.1 GitHub Stats for Matplotlib 3.3.2

GitHub stats for 2020/08/14 - 2020/09/15 (tag: v3.3.1)

These lists are automatically generated, and may be incomplete or contain duplicates.

We closed 15 issues and merged 39 pull requests. The full list can be seen on GitHub

The following 13 authors contributed 61 commits.

- Antony Lee
- Bruno Beltran
- David Stansby
- David Young
- Elliott Sales de Andrade
- Greg Lucas
- Jody Klymak
- johnthagen
- Jouni K. Seppänen
- Richard Sheridan
- Ryan May
- Thomas A Caswell
- Tim Hoffmann

GitHub issues and pull requests:

Pull Requests (39):

- PR #18488: Backport PR #18483 on branch v3.3.x (DOC: reword non-monotonic cell center warning)
- PR #18483: DOC: reword non-monotonic cell center warning
- PR #18485: Backport PR #18475 on branch v3.3.x (BF: ensure exception caught if no kpeswitch)
- PR #18482: Backport PR #18398 on branch v3.3.x (Warn on non-increasing/decreasing pcolor coords)

- PR #18484: Backport PR #18458: Fix huge imshow range
- PR #18475: BF: ensure exception caught if no kpeswitch
- PR #18458: Fix huge imshow range
- PR #18398: Warn on non-increasing/decreasing pcolor coords
- PR #18479: Nbagg backports
- PR #18454: nbagg: Use OutputArea event to trigger figure close.
- PR #18469: Backport PR #18464 on branch v3.3.x (Remove extra stickies in barstacked histogram.)
- PR #18464: Remove extra stickies in barstacked histogram.
- PR #18459: Backport PR #18393 on branch v3.3.x (Fix Axis scale on twinned Axes.)
- PR #18393: Fix Axis scale on twinned Axes.
- PR #18441: Backport PR #18395: TkAgg bugfix: deselect buttons that are not the current _Mode
- PR #18395: TkAgg bugfix: deselect buttons that are not the current _Mode
- PR #18380: Backport PR #18374 on branch v3.3.x (FIX: make _reshape_2D accept pandas df with string indices)
- PR #18374: FIX: make _reshape_2D accept pandas df with string indices
- PR #18376: Backport PR #18298 on branch v3.3.x (Include license files in built distribution)
- PR #18375: Backport PR #18293 on branch v3.3.x (Fix scatter3d color/linewidth re-projection)
- PR #18298: Include license files in built distribution
- PR #18293: Fix scatter3d color/linewidth re-projection
- PR #18361: nbagg: Store DPI ratio on figure instead of window.
- PR #18354: Backport PR #18352 on branch v3.3.x (Avoid triggering backend resolution during qt initial import.)
- PR #18352: Avoid triggering backend resolution during qt initial import.
- PR #18335: Backport PR #18322 on branch v3.3.x (Disable FH4 so that we don't require VCRUNTIME140_1.dll.)
- PR #18322: Disable FH4 so that we don't require VCRUNTIME140_1.dll.
- PR #18333: Backport PR #18328 on branch v3.3.x (Add missing check for None in Qt toolmanager.)
- PR #18328: Add missing check for None in Qt toolmanager.

- PR #18309: Backport PR #18304 on branch v3.3.x (Fix canvas redraws during motion in figures with a Button or TextBox)
- PR #18304: Fix canvas redraws during motion in figures with a Button or TextBox
- PR #18297: Backport PR #18288 on branch v3.3.x (FIX: check if axes is off page before repositioning title)
- PR #18288: FIX: check if axes is off page before repositioning title
- PR #18269: Backport PR #18266 on branch v3.3.x (Fix Path.get_extents for empty paths.)
- PR #18266: Fix Path.get_extents for empty paths.
- PR #18263: Backport PR #18260 on branch v3.3.x (Add parent widget to IntVar)
- PR #18260: Add parent widget to IntVar
- PR #18253: Backport PR #18245 on branch v3.3.x
- PR #18245: MNT: do a better job guessing the GUI framework in use

Issues (15):

- #18415: imshow with LogNorm crashes with certain inputs
- #18447: nbagg: Closing a figure from the notebook does not close the python figure
- #18470: interactive plots slow with matplotlib 3.3.1
- #18457: Incorrect log y-scale for histogram with partitioned and barstacked data
- #18385: twinx not respecting log-scale
- #18371: Plotting a pandas DataFrame with string MultiIndex
- #18296: LICENSE file(s) not included in published PyPI package
- #18287: scatter3D assigns wrong color to points for some plot orientations
- #18292: ImportError: DLL load failed with Matplotlib 3.3.1 on Windows
- #18327: Tool Manager: adding buttons to toolbar fails with matplotlib version 3.3.1 using Qt backend
- #18324: Poor UI responsiveness of 3.3.1 compared with 3.2.2 for interactive mode UI using widgets
- #18303: Canvas redraws during any motion when Button is present
- #18283: Automatic title placement wrong if parent axes is off the page
- #18254: scatter(..., marker='') raises on drawing with mpl3.3.1
- #18259: New IntVar needs a parent widget

7.1.2 GitHub Stats for Matplotlib 3.3.1

GitHub stats for 2020/07/16 - 2020/08/13 (tag: v3.3.0)

These lists are automatically generated, and may be incomplete or contain duplicates.

We closed 25 issues and merged 73 pull requests. The full list can be seen on GitHub

The following 17 authors contributed 131 commits.

- Antony Lee
- Ben Root
- Bruno Beltran
- David Stansby
- Elliott Sales de Andrade
- Isuru Fernando
- jbhopkins
- Jody Klymak
- Jouni K. Seppänen
- Lee Johnston
- linchiwei123
- Neilzon Viloria
- Ryan May
- Thomas A Caswell
- Tim Hoffmann
- Tom Neep
- Yichao Yu

GitHub issues and pull requests:

Pull Requests (73):

- PR #18243: Fix reshape list of strings
- PR #18240: Backport PR #18235 on branch v3.3.x
- PR #18239: Backport PR #18233 on branch v3.3.x (Fix cibuildwheel trigger condition.)
- PR #18235: FIX: check we have a back button in tk toolbar before we touch it
- PR #18233: Fix cibuildwheel trigger condition.
- PR #18231: Backport PR #18224 on branch v3.3.x (Try out cibuildwheel.)
- PR #18224: Try out cibuildwheel.

- PR #18230: Backport PR #18225 on branch v3.3.x (Use certifi when downloading bundled build requirements.)
- PR #18225: Use certifi when downloading bundled build requirements.
- PR #18229: Backport PR #18219 on branch v3.3.x (Fixes an issue where WxAgg NavigationToolbar2 broke custom toolbars)
- PR #18219: Fixes an issue where WxAgg NavigationToolbar2 broke custom toolbars
- PR #18228: Backport PR #18227 on branch v3.3.x (Set pipefail when running flake8 linter.)
- PR #18227: Set pipefail when running flake8 linter.
- PR #18215: Backport PR #18185 on branch v3.3.x (FIX: fix reading from http/https urls via imread)
- PR #18214: Backport PR #18184 on branch v3.3.x (Go back to checking figures for their manager in destroy.)
- PR #18185: FIX: fix reading from http/https urls via imread
- PR #18184: Go back to checking figures for their manager in destroy.
- PR #18183: Backport PR #17995 on branch v3.3.x (Avoid using Bbox machinery in Path.get_extents; special case polylines.)
- PR #18182: Backport PR #17994 on branch v3.3.x (Special case degree-1 Bezier curves.)
- PR #18179: Backport PR #18175 on branch v3.3.x (Downgrade symbol substitution log to info level.)
- PR #18177: Backport PR #18092 on branch v3.3.x (Use same Make as FreeType's configure to build it.)
- PR #18174: Backport PR #18167 on branch v3.3.x (Catch Pandas AssertionError on deprecated multidimensional indexing. Closes #18158)
- PR #18176: Backport PR #18173 on branch v3.3.x (Fix the return value of Axes.get_navigate_mode.)
- PR #18175: Downgrade symbol substitution log to info level.
- PR #18092: Use same Make as FreeType's configure to build it.
- PR #18173: Fix the return value of Axes.get_navigate_mode.
- PR #18167: Catch Pandas AssertionError on deprecated multidimensional indexing. Closes #18158
- PR #18162: Backport PR #18156 on branch v3.3.x (Fix IndexError when using scatter3d and depthshade=False)
- PR #18156: Fix IndexError when using scatter3d and depthshade=False
- PR #18153: Backport PR #18142 on branch v3.3.x (Fix nbagg in Chrome 84)

- PR #18146: Backport PR #17989 on branch v3.3.x (gtk/tk: Ensure no flicker when hovering over images.)
- PR #18142: Fix nbagg in Chrome 84
- PR #18147: Backport PR #18136 on branch v3.3.x (Sort 3d sizes along with other properties)
- PR #18136: Sort 3d sizes along with other properties
- PR #17989: gtk/tk: Ensure no flicker when hovering over images.
- PR #18102: Fix linting on v3.3.x
- PR #18111: Backport PR #18089 on branch v3.3.x
- PR #18109: Backport PR #18093 on branch v3.3.x (Improve saving animated GIF with ffmpeg)
- PR #18089: Revert "Convert adjust_bbox to use ExitStack."
- PR #18093: Improve saving animated GIF with ffmpeg
- PR #18104: Backport PR #18101 on branch v3.3.x (FIX: catch all multi-dim warnings pandas)
- PR #18101: FIX: catch all multi-dim warnings pandas
- PR #18091: ci: Fix linting being ignored by reviewdog
- PR #18083: Backport PR #18079 on branch v3.3.x (Set shading='auto' if invalid value passed to pcolormesh)
- PR #18079: Set shading='auto' if invalid value passed to pcolormesh
- PR #18067: Backport PR #17956 on branch v3.3.x (ENH: Add version check for mac sdk version)
- PR #17956: ENH: Add version check for mac sdk version
- PR #18053: Backport PR #18021: FIX: update num2julian and julian2num
- PR #18021: FIX: update num2julian and julian2num
- PR #18041: Backport PR #18038 on branch v3.3.x (FIX: use internal _set_postion, not external)
- PR #18038: FIX: use internal _set_postion, not external
- PR #18036: Backport PR #18030 on branch v3.3.x (Fix PolyCollection.set_verts optimization.)
- PR #18030: Fix PolyCollection.set_verts optimization.
- PR #18032: Backport PR #18026 on branch v3.3.x (FIX: Be sure matplotlib.backends is imported before we use it)
- PR #18026: FIX: Be sure matplotlib.backends is imported before we use it

- PR #18027: Backport PR #17981 on branch v3.3.x (gtk: Fix `draw` on unmapped windows.)
- PR #17981: gtk: Fix `draw` on unmapped windows.
- PR #18024: Backport PR #17963 on branch v3.3.x (TST: Ignore deprecations when switching backends.)
- PR #18023: Backport PR #18014 on branch v3.3.x (Fix flipped paths in non-writable config dir warning.)
- PR #17963: TST: Ignore deprecations when switching backends.
- PR #18014: Fix flipped paths in non-writable config dir warning.
- PR #18008: Backport PR #17969 on branch v3.3.x (Honor `'Date': None` in metadata)
- PR #18009: Backport PR #17982 on branch v3.3.x (BF: for degenerate polygons, add CLOSEPOLY vertex)
- PR #17982: BF: for degenerate polygons, add CLOSEPOLY vertex
- PR #17969: Honor `'Date': None` in metadata
- PR #17995: Avoid using Bbox machinery in Path.get_extents; special case polylines.
- PR #17994: Special case degree-1 Bezier curves.
- PR #17990: Manual backport of pr 17983 on v3.3.x
- PR #17984: Backport PR #17972 on branch v3.3.x (Fix PyPy compatiblity issue)
- PR #17985: Backport PR #17976 on branch v3.3.x (Fixed #17970 - Docstrings should not accessed with -OO)
- PR #17983: FIX: undeprecate and update num2epoch/epoch2num
- PR #17976: Fixed #17970 - Docstrings should not accessed with -OO
- PR #17972: Fix PyPy compatiblity issue

Issues (25):

- #18234: _reshape_2D function behavior changed, breaks hist for some cases in 3.3.0
- #18232: different behaviour between 3.3.0 and 3.2.2 (and earlier) for ploting in a Tk canvas
- #18212: Updated WxAgg NavigationToolbar2 breaks custom toolbars
- #18129: Error reading png image from URL with imread in matplotlib 3.3
- #18163: Figure can not be closed if it has associated Agg canvas
- #17974: Major speed regression introduced in "plt.bar" definition clipping between 3.0.3 and 3.3.0.

- #17998: New warning: Substituting symbol perp from STIXGeneral
- #18057: Fails to install in FreeBSD
- #18150: Regression in get_navigate_mode() return value
- #18158: X-axis that is Pandas Series time zone aware timestamps raises AssertionError
- #18037: Scatter3D: depthshade=False causes IndexError for Tkinter when plotting more than one point.
- #18169: When running python with -OO option, an empty matplotlib docstring causes an exception.
- #18165: fig.colorbar() and using bbox='tight' in PDF export mess up figure dimensions
- #18132: A simple 3D scatter plot with %matplotlib notebook is not working
- #18135: Point size array in the Axes3D scatter() does not follow the same order as in the data points
- #18061: 3.3.0 regression in png backend with colorbar()
- #18076: pcolormesh + gourand shading + polar axes is broken
- #18010: 3.3.0: possible regression/bug with DateFormatter?
- #18033: v. 3.3.0: horizontal colorbar broken
- #18017: Optimisation in set_verts causes error if `verts` have irregular sizes
- #18022: AttributeError: module 'matplotlib' has no attribute 'backends'
- #18011: Confusing error message when home config directory not writable
- #17975: Computing the bounding box of a degenerate polygon throws an error
- #17968: Setting `Date` metadata to `None` does not remove the date metadata from the SVG file
- #17970: AttributeError when using PYTHONOPTIMIZE (due to stripped docstring)

7.1.3 GitHub Stats for Matplotlib 3.3.0

GitHub stats for 2020/03/03 - 2020/07/16 (tag: v3.2.0)

These lists are automatically generated, and may be incomplete or contain duplicates.

We closed 198 issues and merged 1066 pull requests. The full list can be seen on GitHub

The following 144 authors contributed 3829 commits.

- Adam

- Adam Paszke
- Adam Ruszkowski
- Alex Henrie
- Alexander Rudy
- Amy Roberts
- andrzejnovak
- Antony Lee
- Ardie Orden
- Asaf Maman
- Avni Sharma
- Ben Root
- Bruno Beltran
- Bruno Pagani
- chaoyi1
- Cho Yin Yong
- Chris
- Christoph Pohl
- Cimarron Mittelsteadt
- Clemens Brunner
- Dan Hickstein
- Dan Stromberg
- David Chudzicki
- David Stansby
- Dennis Tismenko
- Dominik Schmidt
- donchanee
- Dora Fraeman Caswell
- Edoardo Pizzigoni
- Elan Ernest
- Elliott Sales de Andrade
- Emlyn Price
- Eric Firing

- Eric Larson
- Eric Relson
- Eric Wieser
- Fabien Maussion
- Frank Sauerburger
- Gal Avineri
- Generated images
- Georg Raiser
- Gina
- Greg Lucas
- hannah
- Hanno Rein
- Harshal Prakash Patankar
- henryhu123
- Hugo van Kemenade
- Ian Hincks
- ImportanceOfBeingErnest
- Inception95
- Ingo Fründ
- Jake Lee
- Javad
- jbhopkins
- Jeroonk
- jess
- Jess Tiu
- jfbu
- Jiahao Chen
- Jody Klymak
- Jon Haitz Legarreta Gorroño
- Jose Manuel Martí
- Joshua Taillon
- Juanjo Bazán

- Julian Mehne
- Kacper Kowalik (Xarthisius)
- Kevin Mader
- kolibril13
- kopytjuk
- ksafran
- Kyle Sunden
- Larry Bradley
- Laurent Thomas
- Lawrence D'Anna
- Leo Singer
- lepuchi
- Luke Davis
- Manan Kevadiya
- Manuel Nuno Melo
- Maoz Gelbart
- Marat K
- Marco Gorelli
- Matt Newville
- Matthias Bussonnier
- Max
- Max Chen
- Max Humber
- Maximilian Nöthe
- Michaël Defferrard
- Michele Mastropietro
- mikhailov
- MuhammadFarooq1234
- Mykola Dvornik
- Nelle Varoquaux
- Nelson Darkwah Oppong
- Nick Pope

- Nico Schlömer
- Nikita Kniazev
- Olivier Castany
- Omar Chehab
- Paul Gierz
- Paul Hobson
- Paul Ivanov
- Pavel Fedin
- Peter Würtz
- Philippe Pinard
- pibion
- Po
- Pradeep Reddy Raamana
- Ram Rachum
- ranjanm
- Raphael
- Ricardo Mendes
- Riccardo Di Maio
- Ryan May
- Sadie Louise Bartholomew
- Sairam Pillai
- Samesh Lakhotia
- SamSchott
- Sandro Tosi
- Siddhesh Poyarekar
- Sidharth Bansal
- Snowhite
- SojiroFukuda
- Spencer McCoubrey
- Stefan Mitic
- Stephane Raynaud
- Steven G. Johnson

- Steven Munn
- Ted Drain
- Terence Honles
- Thomas A Caswell
- Thomas Robitaille
- Till Stensitzki
- Tim Hoffmann
- Todd Jennings
- Tyrone Xiong
- Umar Javed
- Venkada
- vishalBindal
- Vitaly Buka
- Yue Zhihan
- Zulko

GitHub issues and pull requests:

Pull Requests (1066):

- PR #17943: Backport PR #17942 on branch v3.3.x (Increase heading level for 3.3 What's New)
- PR #17942: Increase heading level for 3.3 What's New
- PR #17941: Backport PR #17938 on branch v3.3.x (Don't allow 1D lists as subplot_moasic layout.)
- PR #17940: Backport PR #17885 on branch v3.3.x (BF: ignore CLOSEPOLY after NaN in PathNanRemover)
- PR #17937: Backport PR #17877 on branch v3.3.x (Fix drawing zoom rubberband on GTK backends.)
- PR #17938: Don't allow 1D lists as subplot_moasic layout.
- PR #17885: BF: ignore CLOSEPOLY after NaN in PathNanRemover
- PR #17877: Fix drawing zoom rubberband on GTK backends.
- PR #17933: Backport PR #17858 on branch v3.3.x (Refresh what's new page for 3.3.0)
- PR #17858: Refresh what's new page for 3.3.0
- PR #17919: Backport PR #17913 on branch v3.3.x (Revert using SVG inheritance diagrams)

- PR #17913: Revert using SVG inheritance diagrams
- PR #17911: Backport PR #17907 on branch v3.3.x (Fix release() method name in macosx backend)
- PR #17907: Fix release() method name in macosx backend
- PR #17903: Backport PR #17859 on branch v3.3.x (API: resolve unset vmin / vmax in all ScalarMapple based methods)
- PR #17859: API: resolve unset vmin / vmax in all ScalarMapple based methods
- PR #17898: Backport PR #17882 on branch v3.3.x (Fix FFMpegBase.isAvailable with detached terminals.)
- PR #17882: Fix FFMpegBase.isAvailable with detached terminals.
- PR #17881: Backport PR #17871 on branch v3.3.x (Mention single char colors shading in more places)
- PR #17871: Mention single char colors shading in more places
- PR #17872: Backport PR #17800 on branch v3.3.x (Increase tolerance for alternate architectures)
- PR #17800: Increase tolerance for alternate architectures
- PR #17861: Revert "Fix linewidths and colors for scatter() with unfilled markers"
- PR #17864: Backport PR #17862 on branch v3.3.x (CI: Install, or upgrade, Python 3 on homebrew.)
- PR #17846: Backport PR #17844 on branch v3.3.x (Explain why Qt4 backends are deprecated)
- PR #17844: Explain why Qt4 backends are deprecated
- PR #17833: Backport PR #17831 on branch v3.3.x (BLD: default to system freetype on AIX)
- PR #17831: BLD: default to system freetype on AIX
- PR #17823: Backport PR #17821 on branch v3.3.x (FIX: Keep lists of lists of one scalar each 2D in _reshape_2D)
- PR #17821: FIX: Keep lists of lists of one scalar each 2D in _reshape_2D
- PR #17811: Backport PR #17797 on branch v3.3.x (Fix running contour's test_internal_cpp_api directly.)
- PR #17812: Backport PR #17772 on branch v3.3.x (Partially fix rubberbanding in GTK3.)
- PR #17815: Backport PR #17814 on branch v3.3.x (Don't duplicate deprecated parameter addendum.)
- PR #17814: Don't duplicate deprecated parameter addendum.
- PR #17772: Partially fix rubberbanding in GTK3.

- PR #17797: Fix running contour's test_internal_cpp_api directly.
- PR #17809: Backport PR #17801 on branch v3.3.x (BUG: Fix implementation of _is_closed_polygon)
- PR #17801: BUG: Fix implementation of _is_closed_polygon
- PR #17796: Backport PR #17764 on branch v3.3.x (FIX: be more careful about not importing pyplot early)
- PR #17795: Backport PR #17781 on branch v3.3.x (Fix limit setting after plotting empty data)
- PR #17764: FIX: be more careful about not importing pyplot early
- PR #17781: Fix limit setting after plotting empty data
- PR #17787: Backport PR #17784 on branch v3.3.x (Allow passing emtpy list of ticks to FixedLocator)
- PR #17784: Allow passing empty list of ticks to FixedLocator
- PR #17766: Backport PR #17752 on branch v3.3.x (Numpydoc-ify various functions)
- PR #17752: Numpydoc-ify various functions
- PR #17762: Backport PR #17742 on branch v3.3.x (Update tricontour[f] docs)
- PR #17742: Update tricontour[f] docs
- PR #17760: Backport PR #17756 on branch v3.3.x (Fix tk tooltips for dark themes.)
- PR #17756: Fix tk tooltips for dark themes.
- PR #17747: Backport PR #17731 on branch v3.3.x ("Fix" tight_layout for template backend.)
- PR #17731: "Fix" tight_layout for template backend.
- PR #17739: Backport PR #17734 on branch v3.3.x (Oversample thumbnail x2)
- PR #17734: Oversample thumbnail x2
- PR #17738: Backport PR #17729 on branch v3.3.x (Fix type doc for scroll event "step" attribute.)
- PR #17729: Fix type doc for scroll event "step" attribute.
- PR #17724: Backport PR #17720 on branch v3.3.x (Fix check for manager = None.)
- PR #17720: Fix check for manager = None.
- PR #17719: Backport PR #17693 on branch v3.3.x (DOC: Add svg2pdf converter for generating PDF docs.)
- PR #17693: DOC: Add svg2pdf converter for generating PDF docs.

- PR #17718: Backport PR #17715 on branch v3.3.x (Clarify gridspec error message for non-integer inputs.)
- PR #17717: Backport PR #17705 on branch v3.3.x (Keep cachedRenderer as None when pickling Figure.)
- PR #17715: Clarify gridspec error message for non-integer inputs.
- PR #17705: Keep cachedRenderer as None when pickling Figure.
- PR #17701: Backport PR #17687 on branch v3.3.x (Mention keyboard modifiers in toolbar tooltip texts.)
- PR #17687: Mention keyboard modifiers in toolbar tooltip texts.
- PR #17698: Backport PR #17686 on branch v3.3.x (Fix tooltip for wx toolbar.)
- PR #17686: Fix tooltip for wx toolbar.
- PR #17692: Backport PR #17680 on branch v3.3.x (MNT: migrate away from deprecated c-api)
- PR #17680: MNT: migrate away from deprecated c-api
- PR #17688: Backport PR #17676 on branch v3.3.x (FIX: correctly process the tick label size)
- PR #17676: FIX: correctly process the tick label size
- PR #17677: Backport PR #17664 on branch v3.3.x (Clarify docs of AutoDateLocator.intervald)
- PR #17678: Backport PR #17665 on branch v3.3.x (Document that some single char colors are shaded)
- PR #17679: Backport PR #17675 on branch v3.3.x (DOC: specify that the LaTeX installation needs to include cm-super)
- PR #17675: DOC: specify that the LaTeX installation needs to include cm-super
- PR #17665: Document that some single char colors are shaded
- PR #17664: Clarify docs of AutoDateLocator.intervald
- PR #17672: Backport PR #17668 on branch v3.3.x (Don't pass "wrong" `indent=False` in SVG generation.)
- PR #17671: Backport PR #17667 on branch v3.3.x (Don't linewrap css in svg header.)
- PR #17668: Don't pass "wrong" `indent=False` in SVG generation.
- PR #17667: Don't linewrap css in svg header.
- PR #17666: Prepare for 3.3.0 rc1
- PR #17663: DOC: update the gh stats for v3.3.0
- PR #17656: Fix default colouring of Shadows

- PR #17657: V3.2.x mergeup
- PR #17623: Add a flag for disabling LTO.
- PR #17569: Delay usepackage{textcomp} until after the custom tex preamble.
- PR #17416: Reorder NavigationToolbar2 methods.
- PR #17604: DOC: Clarify offset notation and scientific notation
- PR #17617: Rewrite pdf test to use check_figures_equal.
- PR #17654: Small fixes to recent What's New
- PR #17649: MNT: make _setattr_cm more forgiving
- PR #17644: Doc 33 whats new consolidation
- PR #17647: Fix example in docstring of cbook._unfold.
- PR #10187: DOC: add a blitting tutorial
- PR #17471: Removed idiomatic constructs from interactive figures docs
- PR #17639: DOC: Update colormap deprecation warning to use Python's copy function.
- PR #17223: Warn on invalid savefig keyword arguments
- PR #17625: Give _DummyAxis instances a __name__
- PR #17636: Fix image vlim clipping again
- PR #17635: Fix autoscaling with tiny sticky values.
- PR #17620: MNT: make _setattr_cm more conservative
- PR #17621: FIX: restore ability to pass a tuple to axes_class in axes_grid
- PR #16603: axes collage
- PR #17622: Fix typo in description of savefig.bbox.
- PR #17619: Skip test_tmpconfigdir_warning when running as root.
- PR #17610: MNT: allow 0 sized figures
- PR #17163: Fix clipping of markers in PDF backend.
- PR #17556: DOC: Update contributor listing in credits
- PR #17221: Add metadata saving support to SVG.
- PR #17603: Replace image comparison in test_axes_grid1 by geometry checks.
- PR #17428: Doc start 33 merges
- PR #17607: Convert adjust_bbox to use ExitStack.
- PR #17575: DOCS: update collections.py docstrings to current doc conventions
- PR #15826: Fix bar3d bug with matching color string and array x lengths

- PR #14507: Simplify handling of Qt modifier keys.
- PR #17589: Fix doc build with Sphinx < 3.
- PR #17590: Clarify docs of set_powerlimits()
- PR #17597: MNT: cleanup minor style issues
- PR #17183: Update configuration of CircleCI builds
- PR #17592: Improve docstrings of ScalarFormatter
- PR #17456: Improve stackplot example
- PR #17545: Improve docs of markers
- PR #17233: Improve PDF metadata support in PGF
- PR #17086: Remove jQuery & jQuery UI
- PR #17580: Fix same_color() for 'none' color
- PR #17582: Fix link in doc
- PR #17491: DOC: Only link to overall Zenodo DOI.
- PR #17515: FIX: add set_box_aspect, improve tight bounding box for Axes3D + fix bbox_inches support with fixed box_aspect
- PR #17581: DOC: Remove duplicate Returns in subplot2grid.
- PR #17550: Update subplot2grid doc to use Figure.add_gridspec, not GridSpec.
- PR #17544: markerfacecolor should not override fillstyle='none' in plot()
- PR #15672: Remove mention that tkagg was derived from PIL.
- PR #17573: Examples: fix formatting issue in 'Errorbar limit selection'
- PR #17543: Fix linewidths and colors for scatter() with unfilled markers
- PR #17448: Add example for drawing an error band around a curve
- PR #17572: Examples: clarity for 'set and get' example page
- PR #17276: Allow numpy arrays in markevery
- PR #17536: Consolidate some tests and fix a couple typos
- PR #17558: Simplify plot_date()
- PR #17534: Fmaussion extended boundary norm
- PR #17540: Fix help window on GTK.
- PR #17535: Update docs on subplot2grid / SubplotBase
- PR #17510: Fix exception handling in FT2Font init.
- PR #16953: Changed 'colors' paramater in PyPlot vlines/hlines and Axes vlines/hlines to default to configured rcParams 'lines.color' option
- PR #17459: Use light icons on dark themes for wx and gtk, too.

- PR #17539: Use symbolic icons for buttons in GTK toolbar.
- PR #15435: Reuse png metadata handling of imsave() in FigureCanvasAgg.print_png().
- PR #5034: New "extend" keyword to colors.BoundaryNorm
- PR #17532: DOC: correct legend.title_fontsize docstring
- PR #17531: Remove unneeded check/comment re: multiprocessing in setup.py.
- PR #17522: Privatize ttconv module.
- PR #17517: Make sure _parent is in sync with Qt parent in NavigationToolbar2QT
- PR #17525: DOC/API: set __qualname__ when using class factory
- PR #17511: Fix offset legend tightbbox
- PR #16203: Port fontconfig's font weight detection to font_manager.
- PR #17485: Support marking a single artist as not-usetex.
- PR #17338: Support url on more Artists in svg
- PR #17519: Prefer demo'ing rcParams rather than rc in examples.
- PR #13457: Give `AnnotationBbox` an opinion about its extent
- PR #15037: Simplifications to errorbar().
- PR #17493: Update SVGs that use interpolation='none'.
- PR #15221: Don't fallback to agg in tight_layout.get_renderer.
- PR #17512: DOC: remove inkscape restriction in doc
- PR #17484: Deprecate ismath parameter to draw_tex and ismath="TeX!".
- PR #17492: Correctly set default linewidth for unfilled markers.
- PR #16908: Adding 2d support to quadmesh set_array
- PR #17506: Fix dicts unpacking for `.plot`
- PR #17496: Fix some incorrect image clipping
- PR #17340: convert some sample plots to use plt.subplots() instead of other methods
- PR #17504: Undocument parameter orientation of bar()
- PR #13884: Add some documentation for axisartist's ExtremeFinder, plus some cleanups.
- PR #17495: Fix Pillow import in testing.
- PR #17462: Inline FigureCanvasGtkFoo._render_figure.
- PR #17474: Numpydocify RectangleSelector docstring.

- PR #17003: Optimize extensions with LTO and hidden visibility
- PR #17489: BUG: Picking vertical line broken
- PR #17486: Simplify handling of fontproperties=None.
- PR #17478: Add support for blitting in qt5cairo.
- PR #15641: Make get_sample_data autoload npy/npz files.
- PR #17481: Fix LightSource.shade on fully unmasked array.
- PR #17289: Prepare for ragged array warnings in NumPy 1.19
- PR #17358: Fix masked CubicTriInterpolator
- PR #17477: DOC: Use Sphinx-gallery animation capture
- PR #17482: Shorten RectangleSelector._release.
- PR #17475: Cleanup RectangleSelector example.
- PR #17461: Deprecate the private FigureCanvasGTK3._renderer_init.
- PR #17464: Fold _make_nseq_validator into _listify_validator.
- PR #17469: Use qVersion, not QT_VERSION_STR -- the latter doesn't exist in PySide2.
- PR #4779: DOC: Start to document interactive figures
- PR #17458: Cleanup C++ code
- PR #17466: DOC: clarify that milestones are intentions not approvals
- PR #17062: Fix to "exported SVG files blurred in viewers"
- PR #17443: Fix rcParams validator for dashes.
- PR #17350: Move integerness checks to SubplotSpec._from_subplot_args.
- PR #17444: Support odd-length dash patterns in Agg.
- PR #17405: Show the failing line in bad-rcparams warnings.
- PR #17452: Make validate_date throw ValueError, not RuntimeError.
- PR #17439: Remove comment re: validation of datetime format strings.
- PR #17438: Discourage use of proprietary Matplotlib names for freetype hinting
- PR #16990: update testing helpers
- PR #16340: Make set_x/ymargin() update axes limits, just like margins().
- PR #15029: Get default params from matplotlibrc.template.
- PR #17363: Fix toolbar separators in wx+toolmanager.
- PR #17348: Avoid creating a Tick in Axis.get_tick_space.
- PR #15725: Changed line color of boxplot for dark_background

- PR #17362: Remove status bars in toolmanager mode as well.
- PR #16551: DOC: be more opinionated about flags passed to pip
- PR #17328: Fixes icon clipping issue with WxAgg NavigationToolbar2 for wxpython 4.1.0
- PR #17425: fix typo in stem doc
- PR #17415: Cygwin fixes
- PR #17401: FIX: Fix for FFmpeg + GIF
- PR #16569: MNT: improve the error message in Path init
- PR #17404: Don't forget to dlclose() main_program in tkagg init.
- PR #17414: Keep validate_date private.
- PR #17413: Revert "DOC: drop the experimental tag constrained_layout and tight_layout"
- PR #17394: Deprecate passing keys to update_keymap as single comma-separated string
- PR #17395: TexManager fixes.
- PR #17399: Remove qt4 backends from backend fallback candidates.
- PR #17392: Clarify deprecation message re: tex/pgf preambles as list-of-strings.
- PR #17400: Cleanup wx examples.
- PR #17378: Fix marker overlap
- PR #17351: Fix running the test suite with inkscape>=1.
- PR #17382: FIX: properly check figure on gridspec
- PR #17390: Small updates to troubleshooting guide.
- PR #15104: Simplify file handling in ft2font.
- PR #17380: Support standard names for freetype hinting flags.
- PR #15594: Fix marker overlap
- PR #17372: Auto-set artist.mouseover based on if get_cursor_data is overridden.
- PR #17377: Remove code for sphinx < 1.8
- PR #17266: Keep explicit ticklabels in sync with ticks from FixedLocator
- PR #17359: Fix running test_internal_cpp_api directly.
- PR #17355: Change subprocess for inkscape version detection
- PR #17369: CI: Add eslint for JS linting
- PR #17226: Replace backend_driver by new example runner.
- PR #17365: Also use light color tool buttons in qt+toolmanager+dark theme.

- PR #17366: Restrict Qt toolbars to top/bottom of canvas.
- PR #17361: Remove randomness from test_colorbar_get_ticks_2.
- PR #17151: Cleanup colors.py docstrings.
- PR #17287: Make API of get_tightbbox more consistent between Axes and Axis.
- PR #17092: Don't create a statusbar in Qt, wx backends.
- PR #17220: Simplify Annotation and Text bbox drawing.
- PR #17353: Make zooming work in qt-embedding example.
- PR #16727: Update xtick.alignment parameter in rcsetup to validate against correct values
- PR #17236: Add the "contour.linewidths" configuration option
- PR #16328: Make Artist.set() apply properties in the order in which they are given.
- PR #9696: FIX: set_url() without effect in the plot for instances of Tick
- PR #17002: Fix AnnotationBbox picking and a bit of cleanup
- PR #17256: Improve ps handling of individual usetex strings.
- PR #17267: Improve image comparison decorator
- PR #17332: Cleanup docstring of subplots().
- PR #16843: Deprecate is_pyqt5.
- PR #15898: New textcolor kwarg for legend
- PR #17333: Make sharex, etc. args of subplots() keyword-only.
- PR #17329: Improve docs of eventplot()
- PR #17330: Remove pnpoly license.
- PR #13656: For single datasets, don't wrap artist added by Axes.hist in silent_list
- PR #16247: DOC added kwargs and tight_layout description in plt.figure
- PR #16992: Implement FigureManager.resize for macosx backend
- PR #17324: DOC: add offset axes to secondary_axes
- PR #17311: Make pyplot signatures of rgrids() and thetagrids() explicit
- PR #17302: Fix alignment of offset text on top axis.
- PR #14421: Add GridSpec.subplots()
- PR #15111: By default, don't change the figure face/edgecolor on savefig().
- PR #17318: both x and y should multiply the radius
- PR #17309: Cleanup parameter types in docstrings
- PR #17308: Improve docs of bar() and barh()

- PR #17312: changed axis to axes in lifecycle tutorial
- PR #16715: Automatically create tick formatters for str and callable inputs.
- PR #16959: Simplify and robustify ConnectionPatch coordinates conversion.
- PR #17306: FIX: CL more stable
- PR #17301: Use deprecate_privatize_attribute more.
- PR #16985: Adds normalize kwarg to pie function
- PR #5243: Enhancement of tick label offset text positioning
- PR #17292: Deprecate various wx Toolbar attributes.
- PR #17297: Simplify pickling support.
- PR #17298: Fix rubberband in tk.
- PR #17299: Avoid "dash motion" in qt zoom box.
- PR #17200: Implement set_history_buttons for Tk toolbar.
- PR #16798: Make the Qt interactive zoom rectangle black & white.
- PR #17296: Fix doc wording
- PR #17282: Don't divide by zero in Line2D.segment_hits.
- PR #17293: Fix incorrect deprecation.
- PR #17285: V32 mergeup
- PR #15933: Warn if a temporary config/cache dir must be created.
- PR #15911: Use os.getpid() in configdir, to avoid multiprocess concurrency issues
- PR #17277: Move slow FontManager warning to FontManager constructor.
- PR #17222: FIX: long titles x/ylabel layout
- PR #14960: Don't generate individual doc entries for inherited Axes/Axis/Tick methods
- PR #17175: Further sync axes_grid colorbars with standard colorbars.
- PR #17030: Move widget functions into matplotlib.testing.widgets.
- PR #16975: Fix "out of bounds" undefined behavior
- PR #17111: Deprecate NavigationToolbar2._init_toolbar.
- PR #15275: adds turbo colormap
- PR #17174: Inline RGBAxes._config_axes to its only call site.
- PR #17156: Deprecate text.latex.preview rcParam.
- PR #17242: Make deprecations versions explicit
- PR #17165: Small optimizations to scale and translate of Affine2D

- PR #17181: Inline some private helper methods in ColorbarBase + small refactors.
- PR #17264: Don't trigger save when gtk save dialog is closed by escape.
- PR #17262: fix typo in set_clip_on doc
- PR #17234: Shorten and privatize qt's UiSubplotTool.
- PR #17137: Deprecate Toolbar.press/release; add helper to find overridden methods.
- PR #17245: Improve error handling in _parse_scatter_color_args
- PR #15008: ENH: add variable epoch
- PR #17260: Text Rotation Example: Correct roation_mode typo
- PR #17258: Improve info logged by tex subsystem.
- PR #17211: Deprecate support for running svg converter from path contaning newline.
- PR #17078: Improve nbAgg & WebAgg toolbars
- PR #17191: Inline unsampled-image path; remove renderer kwarg from _check_unsampled_image.
- PR #17213: Replace use of Bbox.bounds by appropriate properties.
- PR #17219: Add support for suptitle() in tight_layout().
- PR #17235: More axisartist cleanups
- PR #17239: Remove deprecations that expire in 3.3
- PR #13696: Deprecate offset_position="data".
- PR #16991: Begin warning on modifying global state of colormaps
- PR #17053: Replace most jQuery with vanilla JavaScript
- PR #17228: Make params to pyplot.tight_layout keyword-only.
- PR #17225: Remove Patch visibility tracking by Legend & OffsetBox.
- PR #17027: Fix saving nbAgg figure after a partial blit
- PR #16847: Ticks are not markers
- PR #17229: Autogenerate subplots_adjust with boilerplate.py.
- PR #17209: Simplify some axisartist code.
- PR #17204: Draw unfilled hist()s with the zorder of lines.
- PR #17205: Shorten tight_layout code.
- PR #17218: Document `Transform.__add__` and `.__sub__`.
- PR #17215: Small cleanups.

- PR #17212: Cleanup text.py.
- PR #17196: Move polar tests to their own module.
- PR #14747: Deprecate AxisArtist.dpi_transform.
- PR #13144: Deprecate NavigationToolbar2GTK3.ctx.
- PR #17202: DOC: Remove extra word
- PR #17194: Small cleanups/simplifications/fixes to pie().
- PR #17102: Switch tk pan/zoom to use togglable buttons.
- PR #16832: Correctly compute path extents
- PR #17193: Document docstring quote convention
- PR #17195: Fix polar tests.
- PR #17189: Make all parameters of ColorbarBase, except `ax`, keyword-only.
- PR #16717: Bugfix for issue 16501 raised ValueError polar subplot with (thetamax - thetamin) > 2pi
- PR #17180: Doc: spines arrows example
- PR #17184: Fix various small typos.
- PR #17143: Move linting to GitHub Actions with reviewdog.
- PR #17160: Correctly go through property setter when init'ing Timer interval.
- PR #17166: Deprecate ScalarMappable.check_update and associated machinery.
- PR #17177: Manually linewrap PS hexlines. Fixes #17176
- PR #17162: Update docs of rc_context()
- PR #17170: Convert SubplotZero example into centered-spines-with-arrows recipe.
- PR #17164: Fix Figure.add_axes(rect=...).
- PR #17154: DOC: Fix some warning and unreproducibility
- PR #17169: Clarify that draw_event occurs after the canvas draw.
- PR #17089: Cleanup some imports in tests
- PR #17040: Improve docs on automated tests
- PR #17145: CI: run pydocstyle with our custom options
- PR #16864: Check parameter type for legend(labels)
- PR #17146: FigureManager/NavigationToolbar2 cleanups.
- PR #16933: Add tests for toolmanager.
- PR #17127: ENH: allow title autopositioning to be turned off

- PR #17150: Many docstring cleanups.
- PR #17148: Fix most instances of D404 ("docstring should not start with 'this'").
- PR #17142: BUGFIX: conditional for add_axes arg deprecation
- PR #17032: Fold table.CustomCell into Cell.
- PR #17117: TextBox improvements.
- PR #17108: Make widgets.TextBox work also when embedding.
- PR #17135: Simplify pan/zoom toggling.
- PR #17134: Don't override update() in NavigationToolbar2Tk.
- PR #17129: In docs remove 'optional' if 'default' can be given
- PR #16963: Deprecate Locator.refresh and associated helpers.
- PR #17133: Fix Button widget motion callback.
- PR #17125: Make multiline docstrings start with a newline.
- PR #17124: Widgets cleanup.
- PR #17123: Cleanup/Simplify Cell._set_text_position.
- PR #16862: FIX: turn off title autopos if pad is set
- PR #15214: Inline wx icon loading.
- PR #16831: Simplify interactive zoom handling.
- PR #17094: DOC: drop the experimental tag constrained_layout and tight_layout
- PR #17101: Avoid "wrapped C/C++ object has been deleted" when closing wx window.
- PR #17028: Changed return type of get_{x,y}ticklabels to plain list
- PR #16058: Deprecate {ContourSet,Quiver}.ax in favor of .axes.
- PR #15349: Use checkboxes as bullet points for the PR review checklists
- PR #17112: Fix some link redirects in docs
- PR #17090: DOCS: add examples of how one "should" use Bbox
- PR #17110: Simplify connection of the default key_press and button_press handlers.
- PR #17070: Cleanups to Qt backend.
- PR #16776: Make cursor text precision actually correspond to pointing precision.
- PR #17026: Add eslint & prettier, and re-format JS
- PR #17091: Make sure slider uses "x" sign before multiplicative factor.

- PR #17082: Cleanup TextBox implementation.
- PR #17067: Simplify and generalize _set_view_from_bbox.
- PR #17081: Update animation_api.rst
- PR #17077: Improve default formatter for Slider values.
- PR #17079: Use True instead of 1 for boolean parameters.
- PR #17074: Fixed a typo in Lifecycle of a Plot
- PR #17072: Cleanup multi_image example.
- PR #15287: Allow sharex/y after axes creation.
- PR #16987: Deprecate case-insensitive properties.
- PR #17059: More missing refs fixes, and associated doc rewordings.
- PR #17057: Simplify subgridspec example/tutorial.
- PR #17058: Fix minor doc typos.
- PR #17024: Clarify docs of Rectangle
- PR #17043: Avoid spurious deprecation warning in TextBox.
- PR #17047: Highlighted .cbook.warn_deprecated() in contributing.rst
- PR #17054: Use slope in axline example
- PR #17048: More missing refs fixes.
- PR #17021: File name made more understandable
- PR #16903: Shorten implementation of Axes methods that just wrap Axis methods.
- PR #17039: Cleanups to contour docs.
- PR #17011: ci: Publish result images as Azure artifacts.
- PR #17038: Improve readability of documenting_mpl.rst
- PR #16996: Clean up get_proj() docstring (used view_init docstring as reference)
- PR #17019: Add return field to documentation of 'get_major_ticks'
- PR #16999: Add section on artifacts to imshow docs
- PR #17029: Fix table.Cell docstrings.
- PR #17025: Fix RecursionError when closing nbAgg figures.
- PR #16971: Don't change Figure DPI if value unchanged
- PR #16972: Fix resize bugs in GTK
- PR #17008: Change the description of Rectangle's xy parameter
- PR #16337: Create axline() using slope

- PR #16947: Fix missing parameter initialization in Axes.specgram()
- PR #17001: Cleanup imshow_extent tutorial.
- PR #17000: More stringent eventplot orientations.
- PR #16771: Deprecate non-string values as legend labels
- PR #15910: Simplify init of EventCollection.
- PR #16998: Made INSTALL.rst consistent
- PR #15393: Cleanup shape manipulations.
- PR #10924: Clear() methods to Radio and CheckButtons and other improvements
- PR #16988: Make plt.{r,theta}grids act as setters even when all args are kwargs.
- PR #16986: update tox.ini to match pythons supported and allow flags for pytest
- PR #16111: Move locking of fontlist.json *into* json_dump.
- PR #13110: Slightly tighten the Bbox/Transform API.
- PR #16973: TST: don't actually render 1k+ date ticks
- PR #16967: Simplify animation writer fallback.
- PR #16812: Bezier/Path API Cleanup: fix circular import issue
- PR #16968: Add link to 3.2 min-supported-requirements.
- PR #16957: Remove unused, private aliases Polygon._{get,set}_xy.
- PR #16960: Improve error for quoted values in matplotlibrc.
- PR #16530: Fix violinplot support list of pandas.Series
- PR #16939: Cleanup/tighten axes_grid.
- PR #16942: Cleanup and avoid refleaks OSX Timer__timer_start.
- PR #16944: TST: update default junit_family
- PR #16823: Dedupe implementation of axes grid switching in toolmanager.
- PR #16951: Cleanup dates docstrings.
- PR #16769: Fix some small style issues
- PR #16936: FIX: Plot is now rendered with correct inital value
- PR #16937: Making sure to keep over/under/bad in cmap resample/reverse.
- PR #16915: Tighten/cleanup wx backend.
- PR #16923: Test the macosx backend on Travis.
- PR #15369: Update style docs
- PR #16893: Robustify `AffineBase.__eq__` against comparing to other classes.

- PR #16904: Turn fontdict & minor into kwonly parameters for set_{x,y}ticklabels.
- PR #16917: Add test for close_event.
- PR #16920: Remove unused _read_ppm_image from macosx.m.
- PR #16877: Cleanup new_fixed_axis examples.
- PR #15049: Annotate argument in axes class match upstream
- PR #16774: Cleanup demo_axes_hbox_divider.
- PR #16873: More fixes to pydocstyle D403 (First word capitalization)
- PR #16896: set_tick_params(label1On=False) should also make offset text invisible.
- PR #16907: Fix typo in implementation of quit_all_keys.
- PR #16900: Document and test common_texification()
- PR #16902: Remove dot from suffix in testing.compare.
- PR #16828: Use more _setattr_cm, thus fix Text('').get_window_extent(dpi=...)
- PR #16901: Cleanup many docstrings.
- PR #16840: Deprecate support for Qt4.
- PR #16899: Remove optional returns from TriAnalyzer._get_compressed_triangulation.
- PR #16618: Use SubplotSpec row/colspans more, and deprecate get_rows_columns.
- PR #15392: Autoscale for ax.arrow()
- PR #14626: Add support for minor ticks in 3d axes.
- PR #16897: Add back missing import.
- PR #14725: Move the debug-mode TransformNode.write_graphviz out.
- PR #15437: Improve handling of alpha when saving to jpeg.
- PR #15606: Simplify OldAutoLocator and AutoDateLocator.
- PR #16863: Shortcut for closing all figures
- PR #16876: Small cleanups to dviread.
- PR #15680: Use more kwonly arguments, less manual kwargs-popping.
- PR #15318: Deprecate unused rcParams["animation.html_args"].
- PR #15303: Make it possible to use rc_context as a decorator.
- PR #16890: Enables hatch alpha on SVG
- PR #16887: Shorter event mocking in tests.

- PR #16881: Validate tickdir strings
- PR #16846: Disconnect manager when resizing figure for animation saving.
- PR #16871: Shorter Path import in setupext.
- PR #16892: Warn in the docs that MouseEvent.key can be wrong.
- PR #16209: Dedupe boilerplate for "adoption" of figure into pyplot.
- PR #16098: Deprecate parameter props of Shadow
- PR #15747: Move Text init to end of Annotation init.
- PR #15679: np.concatenate cleanups.
- PR #16778: Remove more API deprecated in 3.1(part 7)
- PR #16886: Finish removing mentions of idle_event.
- PR #16882: Fix trivial docstring typos.
- PR #16874: Fix pydocstyle D209 (Multi-line docstring closing separate line)
- PR #14044: Remove font preamble caching in TexManager.
- PR #16724: Fixed incorrect colour in ErrorBar when Nan value is presented
- PR #15254: Propagate signature-modifying decorators to pyplot wrappers.
- PR #16868: Update release guide
- PR #14442: In the build, declare all (compulsory) extension modules together.
- PR #16866: Cleanup/update deprecations.
- PR #16850: use validate_[cap/join]style
- PR #16858: Fix various numpydoc style issues
- PR #16848: Cleanup CI setup
- PR #16845: Fix checking of X11 builds with PySide2.
- PR #14199: Deprecate Path helpers in bezier.py
- PR #16838: Inline some more kwargs into setup.py's setup() call.
- PR #16841: Cleanup errorbar subsampling example
- PR #16839: spines doc cleanup
- PR #16844: fix example hist(density=...)
- PR #16827: Fix warnings in doc examples
- PR #16772: Remove more API deprecated in 3.1
- PR #16822: fix bug where make_compound_path kept all STOPs
- PR #16819: Destroy figures by manager instance, not by number.
- PR #16824: Deprecate NavigationToolbar2QT.parent.

- PR #16825: Don't use deprecated Gtk add_with_viewport.
- PR #16816: Merge v3.2.x into master
- PR #16786: Simple cleanups to formatters.
- PR #16807: Update barchart_demo.
- PR #16804: Deprecate some mathtext glue helper classes.
- PR #16808: One more instance of check_in_list.
- PR #16802: Fix incorrect super class of VCentered.
- PR #16789: Update markup for collections docstrings.
- PR #16781: Update image tutorial wrt. removal of native png handler.
- PR #16787: Avoid vstack() when possible.
- PR #16689: Add a fast path for NumPy arrays to Collection.set_verts
- PR #15373: Further shorten quiver3d computation...
- PR #16780: Don't import rcParams but rather use mpl.rcParams (part 3)
- PR #16775: Cleanup axes_divider examples.
- PR #15949: Simplify implementation of SubplotTool.
- PR #14869: Deduplicate code for text-to-path conversion in svg backend.
- PR #16527: Validate positional parameters of add_subplot()
- PR #15622: Cleanup mpl_toolkits locators.
- PR #16744: Reword axes_divider tutorial.
- PR #16746: Reword colorbar-with-axes-divider example.
- PR #15211: Various backend cleanups.
- PR #15890: Remove API deprecated in 3.1 (part 2)
- PR #16757: Simplify interactive zoom handling.
- PR #15515: Combine withEffect PathEffect definitions.
- PR #15977: pgf backend cleanups.
- PR #15981: Reuse colorbar outline and patch when updating the colorbar.
- PR #14852: Use Path.arc() to interpolate polar arcs.
- PR #16686: Deprecate Substitution.from_params.
- PR #16675: Vectorize patch extraction in Axes3D.plot_surface
- PR #15846: Standardize signature mismatch error messages.
- PR #16740: Fix type of `dpi` in docstrings.
- PR #16741: Dedupe RGBAxes examples.

- PR #16755: Reword docstring of panning callbacks, and pass them a MouseButton.
- PR #16749: Document behavior of savefig("extensionless-name").
- PR #16754: Cleanup image.py.
- PR #14606: Generic cleanup to hist().
- PR #16692: Allow MarkerStyle instances as input for lines
- PR #15479: Cleanup axes_rgb.
- PR #16617: Use Path(..., closed=True) more.
- PR #16710: Make format_coord messagebox resize with the window and the content in osx backend
- PR #16681: Simplify docstring interpolation for Box/Arrow/ConnectionStyles.
- PR #16576: Deprecate arg-less calls to subplot_class_factory (and similar factories)
- PR #16652: Deprecate {Locator,Axis}.{pan,zoom}.
- PR #16596: Deprecate dviread.Encoding.
- PR #16231: Deprecate JPEG-specific kwargs and rcParams to savefig.
- PR #16636: Deprecate autofmt_xdate(which=None) to mean which="major".
- PR #16644: Deprecate validate_webagg_address.
- PR #16619: Fix overindented lines.
- PR #15233: backend_ps cleanup.
- PR #16604: Deprecate more rc validators.
- PR #16601: Small unrelated cleanups.
- PR #16584: Rename font_bunch to psfont in textpath.
- PR #16023: Dedupe implementations of fill_between & fill_betweenx.
- PR #16485: Simplify validate_color_for_prop_cycle.
- PR #16285: Deprecate RendererCairo.font{weights,angles}
- PR #16410: Fix support for empty usetex strings.
- PR #11644: Add feature to fallback to stix font in mathtext
- PR #16537: Delay checking for existence of postscript distillers.
- PR #16351: Group all init of Legend.legendPatch together.
- PR #15988: Refactor Annotation properties.
- PR #16421: Shorten the type1-to-unicode name table.
- PR #16200: Deprecate Artist.{set,get}_contains.

- PR #15828: Deprecate support for dash-offset = None.
- PR #16338: Document SymmetricalLogLocator parameters.
- PR #16504: DOC: more pcolor fixes
- PR #15996: Cleanup axes_size.
- PR #16108: Deprecate DraggableBase.on_motion_blit.
- PR #16706: Fix exception causes all over the codebase
- PR #15855: Simplify 3d axes callback setup.
- PR #16219: Simplify CallbackRegistry pickling.
- PR #16002: relax two test tolerances on x86_64
- PR #16063: Make the signature of Axes.draw() consistent with Artist.draw().
- PR #16177: Further simplify setupext.
- PR #16191: Make Figure._axobservers a CallbackRegistry.
- PR #16698: Small edits to toolkits docs.
- PR #15430: Simplify setupext.download_or_cache.
- PR #16694: Lower Text's FontProperties priority when updating
- PR #16511: Add more detailed kwargs docstrings to Axes methods.
- PR #16653: Tutorials: make path/URL option clearer in matplotlibrc tutorial
- PR #16697: Update docstrings for plot_directive.
- PR #16684: Fix exception causes in 19 modules
- PR #16674: Docstring + import cleanups to legend.py.
- PR #16683: Turn mathtext.GlueSpec into a (private) namedtuple.
- PR #16660: Cleanup fancybox_demo.
- PR #16691: Clarify tiny comment re: AnnotationBbox constructor.
- PR #16676: Cleanup animation docstrings.
- PR #16673: DOC: correct title_fontsize docstring
- PR #16669: DOC: update doc release guide
- PR #16563: Parametrize imshow antialiased tests.
- PR #16658: In docs, add multi-axes connectionpatches to Figure, not Axes.
- PR #16647: Update annotation tutorial.
- PR #16638: Remove unused, outdated division operators on jpl_units.
- PR #16509: Add custom math fallback
- PR #16609: Fix exception causes in rcsetup.py

- PR #16637: Update docstrings in figure.py.
- PR #16534: DOC: MaxNLocator and contour/contourf doc update (replaces #16428)
- PR #16597: close #16593: setting ecolor turns off color cycling
- PR #16615: Update custom boxstyles example.
- PR #16610: Added graphviz_docs to conf.py
- PR #16608: Stricter validation of rcParams["axes.axisbelow"].
- PR #16614: Cleanup quiver3d examples.
- PR #16556: Make backend_ps test robust against timestamp changes in ghostscript.
- PR #16602: Cleanup testing.compare.
- PR #16575: Style fix for dynamic axes subclass generation in mpl_toolkits.
- PR #16587: Remove warnings control from tests.py.
- PR #16599: Cleanup dolphin example.
- PR #16586: Deprecate recursionlimit kwarg to matplotlib.test().
- PR #16595: Minor docstring/references update.
- PR #16579: Update usetex_fonteffects example.
- PR #16578: Use rc() less often in examples/tutorials.
- PR #16572: Remove some remnants of hist{,2d}(normed=...).
- PR #16491: Expire the _rename_parameters API changes.
- PR #14592: In SecondaryAxis.set_functions, reuse _set_scale's parent scale caching.
- PR #16279: STY: Fix underindented continuation lines.
- PR #16549: Improve documentation for examples/widgets/textbox.py
- PR #16560: Update URL to pyparsing.
- PR #16292: More edits to Normalize docstrings.
- PR #16536: API/TST: minimum versions
- PR #16559: 3D example avoid using statefull .gca()
- PR #16553: DOC: clarify the expected shapes of eventplot input
- PR #16535: Clarify docs of num parameter of plt.figure()
- PR #16547: Reformat/reword mathtext docstrings.
- PR #16545: Add a smoketest for ps.usedistiller="xpdf".
- PR #16529: Deprecate toggling axes navigatability using the keyboard.

- PR #16521: Remove more API deprecated in 3.1.
- PR #16481: Update set_thetalim documentation
- PR #16524: Cleanup docstrings
- PR #16540: Cleanup imports
- PR #16429: CI: update codecov
- PR #16533: Recommend to amend pull requests
- PR #16531: Also deprecate ignorecase ValidateInStrings.
- PR #16428: DOC: MaxNLocator and contour/contourf doc update
- PR #16525: Don't import rcParams but rather use mpl.rcParams (part 2)
- PR #16528: Improve test failure messages on warnings.
- PR #16393: Shorten PyFT2Font_get_charmap.
- PR #16483: Deprecate most ValidateInStrings validators.
- PR #16523: Reorder mathtext rcparams in matplotlibrc template.
- PR #16520: Update a comment re: minimum version of numpy working around bug.
- PR #16522: Fix deprecation warning
- PR #16515: Fix doc for set_{x,y}label, and then some more.
- PR #16516: Fixes to boxplot() docstring & error messages.
- PR #16508: Multi-dim transforms are non-separable by default.
- PR #16507: Factor out common parts of `__str__` for Transform subclasses.
- PR #16514: Various delayed PR reviews
- PR #16512: Fix a bunch of random typos.
- PR #16510: Doc markup cleanups.
- PR #16500: Dedupe timer attribute docs.
- PR #16503: DOC: suppress warning on pcolor demo
- PR #16495: Deemphasize basemap in user-facing docs.
- PR #16484: Don't forget to set stretch when exporting font as svg reference.
- PR #16486: Simplify validate_color, and make it slightly stricter.
- PR #16246: Avoid using FontProperties when not needed.
- PR #16432: Prefer geomspace() to logspace().
- PR #16099: Consistently name callback arguments event instead of evt
- PR #16477: Remove some APIs deprecated in mpl3.1.

- PR #16475: Use vlines() and plot(), not stem(), in timeline example.
- PR #16474: Switch default of stem(use_line_collection=...) to True.
- PR #16467: Convert named_colors example to use Rectangle
- PR #16047: Remove more API deprecated in 3.1
- PR #16373: Fix usetex_baseline_test.
- PR #16433: Simplify demo_curvelinear_grid2.
- PR #16472: Fix mplot3d projection
- PR #16092: Deprecate clear_temp param/attr of FileMovieWriter.
- PR #15504: Warn when trying to start a GUI event loop out of the main thread.
- PR #15023: Simplify formatting of matplotlibrc.template.
- PR #13535: Validate inputs to ScalarMappable constructor
- PR #16469: FIX: colorbar minorticks when rcParams['x/ytick.minor.visible'] = True
- PR #16401: BLD: Auto-detect PlatformToolset
- PR #16024: Keep parameter names in preprocess_data.
- PR #13390: Make sure that scatter3d copies its inputs.
- PR #16107: Deprecate DraggableBase.artist_picker.
- PR #16455: Update some docstrings in colors.py
- PR #16456: Enable more font_manager tests to be run locally.
- PR #16459: Update backend dependency docs.
- PR #16444: Dedupe spectral plotting tests.
- PR #16460: Remove some mentions of avconv, following its deprecation.
- PR #16443: Parametrize some spectral tests.
- PR #16204: Expire deprecation of mathcircled
- PR #16446: Replace matshow baseline test by check_figures_equal.
- PR #16418: Backend timer simplifications.
- PR #16454: Use pytest.raises(match=...)
- PR #14916: Make kwargs names in scale.py not include the axis direction.
- PR #16258: ENH: add shading='nearest' and 'auto' to `pcolormesh`
- PR #16228: Allow directly passing explicit font paths.
- PR #16445: Remove a bunch of imports-within-tests.
- PR #16440: Expire deprecation of stackrel.

- PR #16439: Rework pylab docstring.
- PR #16441: Rework pylab docstring.
- PR #16442: Expire deprecation of stackrel.
- PR #16365: TST: test_acorr (replaced image comparison with figure comparion)
- PR #16206: Expire deprecation of stackrel
- PR #16437: Rework pylab docstring.
- PR #8896: Fix mplot3d projection
- PR #16430: Remove unnecessary calls to np.array in examples.
- PR #16407: Remove outdated comment re: PYTHONHASHSEED and pytest.
- PR #16225: Cleanup animation examples.
- PR #16336: Include axline() in infinite lines example
- PR #16395: Add set/get for ellipse width/height
- PR #16431: CI: add py38 to azure matrix
- PR #16415: Expire some APIs deprecated in mpl3.1.
- PR #16425: MNT: rename internal variable
- PR #16427: Style-fix some examples and update .flake8 per-file-ignores.
- PR #16423: Slightly improve streamplot code legibility.
- PR #16414: DOC: Fix `axes:plot` method docstring verb tense
- PR #16408: Deprecate avconv animation writers.
- PR #16406: Don't import rcParams but rather use mpl.rcParams.
- PR #16326: Cleanup stack
- PR #16193: Catch shadowed imports in style checks.
- PR #16374: Log about font manager generation beforehand.
- PR #16372: Dedupe ImageGrid doc from tutorial and docstring.
- PR #16380: "gif" third-party package added to the extension page
- PR #16327: Cleanup list copying
- PR #16366: Special-case usetex minus to zero depth.
- PR #16350: TST: Improved test (getting rid of image comparison test for test_titletwiny)
- PR #16359: Make Text.update_from copy usetex state.
- PR #16355: typo in `ticker.ScalarFormatter` doc
- PR #15440: Use rcParams to control default "raise window" behavior (Qt,Gtk,Tk,Wx)

- PR #16302: Cleanup Legend._auto_legend_data.
- PR #16329: ENH: add zorder kwarg to contour clabel (and a better default value for zorder)
- PR #16341: Remove mention of now-removed --verbose-foo flags.
- PR #16265: Fix spy(..., marker=<not-None>, origin="lower")
- PR #16333: Document animation HTML writer.
- PR #16334: Fix doc regarding deprecation of properties.
- PR #16335: Fix some more missing references.
- PR #16304: Simplify Legend.get_children.
- PR #16309: Remove duplicated computations in Axes.get_tightbbox.
- PR #16314: Avoid repeatedly warning about too many figures open.
- PR #16319: Put doc for XAxis befor YAxis and likewise for XTick, YTick.
- PR #16313: Cleanup constrainedlayout_guide.
- PR #16312: Remove unnecessary Legend._approx_text_height.
- PR #16307: Cleanup axes_demo.
- PR #16303: Dedupe Legend.draw_frame which is the same as set_frame_on.
- PR #16261: TST: move the Qt-specific handling to conftest
- PR #16297: DOC: fix description of vmin/vmax in scatter
- PR #16288: Remove the private, unused _csv2rec.
- PR #16281: Update/cleanup pgf tutorial.
- PR #16283: Cleanup backend_agg docstrings.
- PR #16282: Replace "unicode" by "str" in docs, messages when referring to the type.
- PR #16289: axisartist tutorial markup fixes.
- PR #16293: Revert "Fix doc CI by pointing to dev version of scipy docs."
- PR #16287: Improve markup for rcParams in docs.
- PR #16271: Clean up and clarify Normalize docs
- PR #16290: Fix doc CI by pointing to dev version of scipy docs.
- PR #16276: Cleanup docstring of print_figure, savefig.
- PR #16277: Prefer using MouseButton to numeric values in docs and defaults.
- PR #16270: numpydoc-ify SymLogNorm
- PR #16274: Tiny cleanups to set_xlabel(..., loc=...).
- PR #16273: DOC: Changing the spelling of co-ordinates.

- PR #15974: Enable set_{x|y|}label(loc={'left'|'right'|'center'}...)
- PR #16248: Update matplotlib.__doc__.
- PR #16262: Dedupe update of rcParams["backend"] in use() and in switch_backend()
- PR #9629: Make pcolor(mesh) preserve all data
- PR #16254: DOC: pdf.preamble --> pgf.preamble
- PR #16245: Cleanup image docs
- PR #16117: CI: Unify required dependencies installation
- PR #16240: Cleanup custom_scale example.
- PR #16227: Make Animation.repeat_delay an int, not an int-or-None.
- PR #16242: CI: Remove PYTHONUNBUFFERED=1 on Appveyor
- PR #16183: Remove some baseline images for plot() tests.
- PR #16229: And more missing refs.
- PR #16215: Concise dates test
- PR #16233: Reword ScalarFormatter docstrings.
- PR #16218: Cleanup animation docs.
- PR #16172: And more missing references.
- PR #16205: Deprecate the empty matplotlib.compat.
- PR #16214: Fix overindented line in AnchoredOffsetbox doc.
- PR #15943: Deprecate the TTFPATH & AFMPATH environment variables.
- PR #16039: Deprecate unused features of normalize_kwargs.
- PR #16202: Remove outdated statement in tight_layout guide.
- PR #16201: UnCamelCase examples.
- PR #16194: Numpydoc ticklabel_format.
- PR #16195: Numpydoc ContourSet.find_nearest_contour.
- PR #16198: Remove em dash
- PR #16199: Do not use camel case for variables in examples
- PR #15644: Rewrite cursor example to include speedup possibilities
- PR #16196: Cleanup patches docstrings.
- PR #16184: Expire a mpl2.2-deprecated API
- PR #16188: Remove ref. to non-existent method in animation tests.
- PR #16170: Deprecate old and little used formatters.

- PR #16187: Fix overly long lines in examples & tutorials.
- PR #15982: Colorbar cleanup.
- PR #16154: Deprecate setting pickradius via set_picker
- PR #16174: Numpydocify artist.getp().
- PR #16165: Remove rcParams deprecated in mpl3.0/3.1.
- PR #16141: Update _base.py
- PR #16169: Add missing spaces after commas.
- PR #15847: Remove some dead branches from texmanager code.
- PR #16125: Fix more missing references again.
- PR #16150: Simplify transforms addition.
- PR #16152: Inline _init_axes_pad into Grid.__init__.
- PR #16129: Deprecate some Transform aliases in scale.py.
- PR #16162: (Mostly) avoid the term "command" in the docs.
- PR #16159: Simple cleanups for contour.py.
- PR #16164: Fix trivial typo in deprecation warning message.
- PR #16160: Cleanup hist() docstring.
- PR #16149: DOC: reword density desc in `ax.hist`
- PR #16151: Remove outdated comment re: blended transforms.
- PR #16102: Rework example "Scatter Star Poly" to "Marker examples"
- PR #16134: Validate Line2D pickradius when setting it, not when reading it.
- PR #15019: Add step option where='edges' to facilitate pre-binned hist plots
- PR #16142: Avoid using np.r_, np.c_.
- PR #16146: Remove LICENSE_CONDA.
- PR #16133: Reword docstring of Line2D.contains.
- PR #16120: Minor fontproperty fixes.
- PR #15670: Reuse Grid.__init__ in ImageGrid.__init__.
- PR #16025: Deprecate update_datalim_bounds.
- PR #16001: Remove parameters deprecated in 3.1
- PR #16049: Add __repr__ to SubplotSpec.
- PR #16100: Consistently name event callbacks on_[event]
- PR #16106: In DraggableLegend, inherit DraggableBase.artist_picker.
- PR #16109: Name Axes variables ax instead of a

- PR #16115: Fix more missing references.
- PR #16096: Deprecate unused parameters
- PR #16085: Improve docstrings in offsetbox.py
- PR #16097: Cleanup unused variables
- PR #16101: Fix incorrect doc regarding projections.
- PR #16095: Deprecate MovieWriter.{exec,args}_key, making them private.
- PR #16078: Refactor a bit animation start/save interaction.
- PR #16081: Delay resolution of animation extra_args.
- PR #16088: Use C++ true/false in ttconv.
- PR #16082: Defaut to writing animation frames to a temporary directory.
- PR #16070: Make animation blit cache robust against 3d viewpoint changes.
- PR #5056: MNT: more control of colorbar with CountourSet
- PR #16051: Deprecate parameters to colorbar which have no effect.
- PR #16045: Use triple-double-quotes for docstrings
- PR #16076: Cleanup path_editor example.
- PR #16059: Simplify colorbar test.
- PR #16072: Cleanup category.py docstrings.
- PR #15769: scatter() should not rescale if norm is given
- PR #16060: Cleanup pcolor_demo.
- PR #16057: Trivial docstring fix for cbook.deprecated.
- PR #16043: Simplify some comparisons
- PR #16044: Code style cleanup
- PR #15894: rcsetup cleanups.
- PR #16050: Unbreak CI.
- PR #16034: Update comments re: colors._vector_magnitude.
- PR #16035: Make eventplot use the standard alias resolution mechanism.
- PR #15798: Better default behavior for boxplots when rcParams['lines.marker'] is set
- PR #16004: Improve documentation of text module
- PR #15507: Use FixedFormatter only with FixedLocator
- PR #16008: Remove unused imports
- PR #16036: Rely on pytest to record warnings, rather than doing it manually.

- PR #15734: Fix home/forward/backward buttons for 3d plots.
- PR #16038: Cleanup contour_demo.
- PR #15998: Join marker reference and marker fiillstyle reference
- PR #15976: Cleanup span_where.
- PR #15990: Remove deprecated support for setting single property via multiple aliases
- PR #15940: Some unicode-support related cleanups.
- PR #15836: Compactify a bit the EventCollection tests.
- PR #16013: Relayout some conditions in axes_grid.
- PR #16010: Inherit the Artist.draw docstring in subclasses.
- PR #16017: Document support for no-args plt.subplot() call.
- PR #16014: Simplify calls to AxesGrid/ImageGrid.
- PR #16012: Normalize aspect="equal" to aspect=1 in the setter.
- PR #15997: Shorten wx _onMouseWheel.
- PR #15993: Style fixes for axes_divider.
- PR #15989: Simplify Artist.update.
- PR #16015: Some small extension cleanups
- PR #16011: Replace axes_size.Fraction by multiplication.
- PR #15719: Templatize spectral helpers.
- PR #15995: Remove toolkit functions deprecated in 3.1
- PR #16003: prevent needless float() conversion
- PR #16000: De-deprecate *min/*max parameters to set_x/y/zlim()
- PR #15684: Avoid RuntimeError at wx exit.
- PR #15992: Avoid using np.matrix.
- PR #15961: Be more opinionated for setting up a dev env.
- PR #15991: Avoid setting dtypes as strings...
- PR #15985: Remove unnecessary :func:, :meth: from examples markup.
- PR #15983: Fix some examples docstrings.
- PR #15979: Remove references to scipy cookbook.
- PR #15966: FIX: check subplot kwargs
- PR #15947: Merge the two usetex demos.
- PR #15939: Exceptions should start with a capital letter

- PR #15948: Use rc_context more.
- PR #15962: Add tests for IndexFormatter
- PR #15965: Test registering cmaps
- PR #15950: Remove deprecated TextWithDash
- PR #15942: Update docs of type1font
- PR #15927: Trying to set the labels without setting ticks through pyplot now raises TypeError*
- PR #15944: Minor doc cleanups
- PR #15945: Do not use "object" or "instance" when documenting types
- PR #15897: Cleanup TriAnalyzer docs
- PR #15777: Don't bother disconnecting idle_draw at gtk shutdown.
- PR #15929: Remove unused cbook._lockstr.
- PR #15935: Raise an ValueError when Axes.pie accepts negative values #15923
- PR #15895: Deprecate unused illegal_s attribute.
- PR #15900: Rewrite test_cycles to avoid image comparison tests.
- PR #15892: Update docs of backend_manager
- PR #15878: Remove API deprecated in 3.1
- PR #15928: DOC: use markers as slanted breaks in broken axis example
- PR #14659: Update some widget docstrings.
- PR #15919: Remove mod_python specific code.
- PR #15883: Improve error when passing 0d array to scatter().
- PR #15907: More docstrings cleanup.
- PR #15906: Cleanup legend docstrings.
- PR #15776: Improve doc for data kwarg.
- PR #15904: Deemphasize ACCEPTS blocks in documenting_mpl docs.
- PR #15891: Mark self.* expressions in docstrings as literal
- PR #15875: Deprecate implicit creation of colormaps in register_cmap()
- PR #15885: Cleanup text.py docstrings.
- PR #15888: Cleanup backend_bases docs.
- PR #15887: Fix AnnotationBbox docstring.
- PR #15858: Avoid some uses of len-1 tuples.
- PR #15873: Standardize parameter types in docs

- PR #15874: Cleanup backend_bases docs
- PR #15876: Deprecate case-insensitive capstyles and joinstyles.
- PR #15877: Suppress exception chaining on rc validator failure.
- PR #15880: Use True/False instead of 0/1 as booleans in backend_ps.
- PR #15827: Fix validation of linestyle in rcparams and cycler.
- PR #15850: Docstrings cleanup in matplotlib.axes
- PR #15853: np.abs -> (builtins).abs
- PR #15854: Simplify Axes3D init.
- PR #15822: More cleanup defaults in docstrings
- PR #15838: Remove some references to Py2.
- PR #15834: Optimize colors.to_rgba.
- PR #15830: Allow failure on nightly builds.
- PR #15788: Fixes pyplot xticks() and yticks() by allowing setting only the labels
- PR #15805: Improve docs on figure size
- PR #15783: Fix stepfilled histogram polygon bottom perimeter
- PR #15812: Cleanup defaults in docstrings
- PR #15804: Cleanup many docstrings.
- PR #15790: Update docs of PolyCollection
- PR #15792: Cleanup dviread docs.
- PR #15801: Cleanup some references to rcParams in docs.
- PR #15787: Cleanup `Colormap.__call__`.
- PR #15766: Shorten description on search page
- PR #15786: Slightly clarify the implementation of safe_masked_invalid.
- PR #15767: Update badges in README.rst
- PR #15778: Fix typos and comma splices in legend guide
- PR #15775: Some pathlibification.
- PR #15772: Directly dedent the spectral parameter docs.
- PR #15765: Reword some docstrings.
- PR #15686: Simplify and unify character tracking in pdf and ps backends (with linked fonts)
- PR #9321: Add Axes method for drawing infinite lines
- PR #15749: Fix travis links in README

- PR #15673: Rely on findfont autofallback-to-default in pdf/ps backends.
- PR #15740: Small animation cleanup.
- PR #15739: ImageMagick animators now can use extra_args
- PR #15591: Remove FAQ on 'Search' -- already referenced in search file
- PR #15629: Consistently use realpaths to build XObject names
- PR #15696: Improve mathtext.fontset docs and fix :mathmpl: cache bug.
- PR #15721: Render default values in :rc: directive as literal
- PR #15720: Suppress triage_tests warning on Py3.8.
- PR #15709: Make 3d plot accept scalars as arguments.
- PR #15711: Don't explicitly list scalez kwarg in Axes3D constructor and docs.
- PR #14948: Simplify Tick and Axis initialization.
- PR #15693: Also test PySide2 on CI.
- PR #15701: Tried to solve Issue #15650: Print URL when webbrowser.open Fails
- PR #15704: Fix more broken refs.
- PR #15687: Add tooltips to HTML animation controls
- PR #15592: Offset text position
- PR #15697: Fix some broken doc refs.
- PR #15700: Parametrize some spectral tests.
- PR #15699: Fix some incorrect ValueErrors.
- PR #15698: Bump numpy dependency to >=1.15.
- PR #15694: Handle upcoming deprecation of np.float.
- PR #15691: Correctly handle high dpi in Pillow animation writer.
- PR #15676: Doc adopt nep29
- PR #15692: Update FUNDING.yml
- PR #15645: Bump minimal numpy version to 1.12.
- PR #15646: Hide sphinx-gallery config comments
- PR #15642: Remove interpolation="nearest" from most examples.
- PR #15671: Don't mention tcl in tkagg commments anymore.
- PR #15607: Simplify tk loader.
- PR #15651: Simplify axes_pad handling in axes_grid.
- PR #15652: Remove mention of Enthought Canopy from the docs.

- PR #15655: Remove outdated license files.
- PR #15639: Simplify axes_grid.Grid/axes_grid.ImageGrid construction.
- PR #15640: Remove some commented-out code from axes_grid.
- PR #15643: Fix examples claiming matplotlib can't plot np.datetime64.
- PR #15375: Add note to hist docstring about speed
- PR #15461: Fix invalid checks for axes_class parameter in ImageGrid.
- PR #15635: Deprecate "U" mode passed to cbook.to_filehandle().
- PR #15563: In backend_pgf, directly open subprocess in utf8 mode.
- PR #15462: Simplify azure setup.
- PR #13075: Remove logic for optionally building Agg and TkAgg.
- PR #15262: Declare qt figureoptions tool in toolitems.
- PR #15292: Shorten RendererWx.get_wx_font.
- PR #15569: Allow linking against a system qhull as well.
- PR #15589: Make sure that figures are closed when check_figures_equal finishes
- PR #15465: Validate and simplify set_tick_params(which=...)
- PR #15090: Coerce MxNx1 images into MxN images for imshow
- PR #15578: BLD: set the max line length on the flake8 config
- PR #15564: Use True instead of 1 as filternorm default
- PR #15536: Add a backend kwarg to savefig.
- PR #15571: Cleanup following using Pillow as universal image reader
- PR #15476: Default to local_freetype builds.
- PR #15557: Skip failing pgf test when sfmath.sty is not present.
- PR #15555: Add pgf to list of builtin backends in docs.
- PR #15534: BLD: update pillow dependency
- PR #15427: Separate plots using #### in demo_fixed_size_axes.py
- PR #15505: Cleanup axisartist tutorial.
- PR #15506: Rename locator.den to the clearer locator.nbins in mpl_toolkits.
- PR #15502: Get rid of trivial compiler warning.
- PR #15451: Ci py38
- PR #15484: Cleanup docs regarding compilers.
- PR #15467: Validate locator_params(axis=...)

- PR #15330: Add axes method for drawing infinite lines.
- PR #15482: Trivial style fixes to constrained_layout.
- PR #15418: Use correct pip/pytest on azure
- PR #15466: Update tick_params() docs
- PR #15463: Remove staticbuild option from setup.cfg.template.
- PR #15378: Don't link ft2font to zlib by default.
- PR #15270: When no gui event loop is running, propagate callback exceptions.
- PR #15447: Move testing of Py3.8 to Travis.
- PR #15431: Fix range(len()) usages
- PR #15390: Simplify implementation of vectorized date operations.
- PR #15403: Fix DeprecationWarning in nightly testing
- PR #15394: Deprecate {NonUniformImage,PcolorImage}.is_grayscale.
- PR #15400: Updated INSTALL.rst to correct install commands
- PR #13788: Autoscale for ax.arrow()
- PR #15367: Update the readme on providing API changes
- PR #15193: Switch to using pillow for png as well.
- PR #15346: vectorized calc_arrow loop in quiver
- PR #15011: Adding example for drawstyle
- PR #15371: Deprecate Colorbar.config_axis()
- PR #15361: Update next API changes to new structure
- PR #15274: NavigationToolbar2Tk: make packing optional.
- PR #15158: Change the way API changes are documented
- PR #15356: Fix broken imports.
- PR #15200: Simplify SubplotParams.update().
- PR #15210: Explicitly list allowed "unused" imports, remove the rest.
- PR #15348: Some figure and related docs cleanup
- PR #13355: Simplify and generalize BezierSegment.
- PR #14917: ENH: box aspect for axes
- PR #14949: Use fix_minus in format_data_short.
- PR #15341: Move non-gui warning message to backend_bases.
- PR #15335: Add discourse link to readme
- PR #15293: Fixes for wx savefig dialog.

- PR #15324: Update PR guidelines
- PR #15301: Update colorbar docs
- PR #15340: Always attach a manager attribute (possibly None) on canvas.
- PR #15319: Make validate_movie_writer actually check registered writers.
- PR #10973: PGF: Replace pgfimage by includegraphics to fix import regression
- PR #15302: fix warning used by cbook.warn_deprecated()
- PR #15321: Sort missing_references.json.
- PR #15290: Unify fig.delaxes(ax) and ax.remove().
- PR #15309: Simplify sca().
- PR #15201: Autogenerate gca(), gci() from boilerplate.py.
- PR #15305: Autogenerate footer Copyright year
- PR #15294: Replace custom logging in wx by stdlib logging.
- PR #15288: More properties aliases.
- PR #15286: throw deprecation warning on empty call to fig.add_axes()
- PR #15282: Colorbar cleanup.
- PR #15250: Cleanup font_manager.
- PR #13581: Cleanup _pylab_helpers.
- PR #15273: DOC: don't use term units in transform tutorial
- PR #15263: Correctly setup comparisons in test_compare_images.
- PR #15226: Turn gtk3 pan/zoom button into togglable buttons.
- PR #14609: Simplify implementation of set_{x,y}bound.
- PR #15261: Change layout of test_triager to avoid cropping images.
- PR #15236: Dedupe SubplotSpec construction in mpl_toolkits.
- PR #14130: Add decorator to inherit keyword-only deprecations
- PR #15249: In findfont(fallback_to_default=False), throw if default font is missing
- PR #15175: Simplify pdf image output.
- PR #7506: [WIP] Add Axes method for drawing infinite lines.

Issues (198):

- #16501: Setting a thetalim > 2pi gives odd results
- #15035: security exposure in the packaged jquery library
- #10375: Coordinate text wrapping in navigation toolbar

- #10720: Option to set the text color in legend to be same as the line
- #17868: plt.bar with nan input fails rendering in notebook using 3.3.0rc1
- #17773: gtk3 rubberband is invisible
- #5726: Cursor displays x, y coordinates with too much or too little precision
- #5164: Sort out qt_compat
- #17905: macosx backend warns when using the zoom method
- #17703: QuadMesh.get_clim changed behavior in 3.3.0rc1
- #17875: animation.writers['ffmpeg']" is hung when run in background.
- #17591: Single-character colors do not match long names
- #16905: if pie normalizes depends on input values
- #17829: trunk fails to build in AIX
- #17820: Regression: _reshape_2D no longer preserves the shape of lists of lists of one scalar each
- #17807: "%matplotlib notebook" Download is Noise After Interacting with Plot
- #17763: matplotlib.use('agg', force=True) does not ignore unavailable configured backend
- #17586: Surprising datetime autoscaling after passing empty data
- #17792: when using plt.tight_layout(), figure title overlaps subplot titles
- #17736: ax.set_xticklabels([]) for categorical plots is broken in 3.3.0rc1
- #17757: Plotting Hist with histtype 'stepfilled' does not respect bottom correctly
- #17744: BUG: AttributeError: 'FigureCanvasBase' object has no attribute 'print_png' in 3.3rc0
- #17730: Using backend Template and plt.tight_layout raises UnboundLocalError
- #17716: Error using "set_window_title" for canvas via backend_qt5agg
- #17681: PDF cannot be built due to Zenodo SVGs
- #17627: AttributeError: 'Figure' object has no attribute '_cachedRenderer'
- #17658: Feature request: Add advanced zoom and inspect feature to GUI for more productivity
- #17629: Use of Python deprecated APIs.
- #17670: BUG: Setting ticksize xx-small broken by #17348
- #17673: RuntimeError: latex was not able to process the following string: b'$\\mathdefault{-2}$'
- #17412: Document the dependency on the type1ec LaTeX package

- #17643: AutoDateLocator docs has a typo
- #9118: make TeXManager more user-configurable
- #11131: Make pyplot.pause not give focus to the figure window
- #17646: more conservative setattr_cm broke mplcairo
- #17634: Cannot copy LinearSegmentedColormap
- #16496: Single path optimisation for Collection w/ offsets broken
- #192: Savefig does not issue a warning on a non-existent keyword n
- #17624: _DummyAxis needs a __name__ attribute for ScalarFormatter
- #16910: Axes.imshow draws invalid color at value is 0 when max of 'X' not equal to vmax
- #17637: streamplot and sticky edges interaction
- #17633: Stackplot fails for small numbers
- #17616: waitforbuttonpress in Linux
- #17615: small bug in documentation of backend.FigureCanvasBase.start_event_loop
- #17093: Zero size figure use case
- #17608: How avoid PyQt5 to crash when I move Qslitter to the edge with a matplotlib figure in it?
- #9829: Vertices clipped for certain markers when plotting more than two points and saving as pdf
- #15815: bar3d color length bug
- #15376: ScalarFormatter.set_powerlimits documentation seems inconsistent
- #17595: Master doc builds broken
- #16482: Pyplot hlines and vlines do not use the 'lines.color' property in rcParams by default
- #16388: rethink how we display DOI svg badges
- #17172: set_aspect for 3D plots
- #16463: Jupyter "inline" backend seems to misinterpret "figsize" with Axes3D
- #17527: The markers are not hollow when I use ax.scatter() and set markers.MarkerStyle()'s fillstyle to 'none'. My usage is wrong?
- #7491: sort out if the high-resolution ellipse code still works
- #17398: Plotting an error band along a curve
- #8550: Matplotlib chooses the wrong font for unrecognized weights

- #8788: Font issue: findfonts should differentiate between thin and regular ttf fonts
- #10194: legend is not present in the generated image if I use 'tight' for bbox_inches
- #17336: set_url without effect for instances of Line2D
- #9695: set_url() without effect in the plot for instances of Tick
- #17192: How to change the thickness of the marker "x" when using scatter?
- #17507: pyplot.savefig() throwing warning suggesting a bug (possibly in figManger)
- #17502: dict unpacking broken for `.plot` in 3.2
- #15546: plt.imshow: clip_on=False has no effect
- #17023: DOC: Tutorial/Sample plots should use same fig/axis creation method
- #7537: Conflict between different AGG static libraries in a same binary
- #16836: Dropping support for PyQt4; preparing support for PyQt6.
- #17455: LightSource.shade fails on a masked array
- #16353: BUG: VisibleDeprecationWarning in boxplot
- #11820: Compressed Triangulation Masking in CubicTriInterpolator
- #11823: Animation Examples
- #15410: Change in OSX Catalina makes matplotlib + multiprocessing crash
- #17467: Bug Report: saved Figure ignores figure.facecolor
- #17343: Regression in add_subplot..
- #7093: ordering issues between `set_xmargin` and `set_xscale`
- #13971: Unnecessary drawing with NbAgg
- #17432: Scatter accepts marker=MarkerStyle(), but plot does not
- #15675: Boxplot line color with style dark_background should be bright
- #5962: No output from pyplot on cygwin64 python3 virtualenv
- #17393: TexManager.get_rgba fails
- #5830: Incorrect overlap of markers in scatter3D
- #11937: Limiting ticks on colorbar axes falsify tick labels.
- #17354: Converter detection fails for inkscape if on headless system without DISPLAY
- #17352: Zoom In-Out not behaving as expected in QT backend example
- #15409: Datetime plot fails with 'Agg' backend in interactive mode

- #14155: Adding GridSpec.subplots?
- #16583: matplotlibrc validates some parameters wrongly
- #16946: Pick_event on AnnotationBbox fires at wrong position
- #15131: set_size_inches doesn't resize window on macosx backend
- #7619: Figure background colors
- #15899: Describe possible kwargs that may be input into a function
- #17304: constrained-layout gives wrong results when explicitly equal width ratios are set
- #17295: DOC: https://matplotlib.org/api/_as_gen/matplotlib.quiver.Quiver.html
- #17294: DOC: matplotlib.axes.Axes.annotate.html
- #17290: backend_svg fails with dashed line style
- #16677: tmp_config_or_cache_dir atexit cleanup fails after forks()
- #15091: Turbo color map
- #7372: Moving get_ax and do_event to testing
- #15225: Show offset text on subplots after sharing axis
- #7138: misplaced spines in dates plot
- #17243: Misleading error message in _parse_scatter_color_args
- #16461: Hexbin if singular and mincnt used
- #14596: forward port jquery removal from ipympl
- #17217: Transform operators are not publicly documented....
- #2253: matplotlib makes python lose focus
- #7184: margins does not handle bézier curves
- #16830: `_path.get_extents` does not correctly handle bezier curves
- #17176: Print figure using PS backend is needlessly slow
- #17141: flake8-docstrings does not check all of our requirements
- #16567: Let legend get the handles from the provided objects if not specified explicitly.
- #16805: Titles cannot be padded to negative numbers anymore.
- #17114: `add_axes` shows deprecation warning when called with only `kwargs`
- #16885: Change return type get_{x,y}ticklabels to plain list
- #17044: widgets.TextBox continuously creates new text objects and linecollection objects.
- #17066: documentation of animation contains non-working code example

- #16588: Rename next_api_changes to api_changes_3.x (whatever number makes sense)
- #17015: `get_major_ticks` docs missing return type
- #16976: Thin line color distortion on large scale
- #16934: gtk3 window immediately resizes down to zero-height upon showing up.
- #16941: test_determinism_check is failing (sometimes)
- #16982: pyplot.rgrids don't do anything
- #16952: How to solve an error of "ValueError: Key backend: Unrecognized backend string '"agg"'
- #15272: Axes.violinplot has small issue in using pandas.DataFrame without index 0.
- #16926: tk window immediately resizes down to zero-height upon showing up.
- #16919: wx backends don't send close_event if window is closed via "q" keypress
- #16854: small typo in the documentation
- #16895: offset text still visible with ImageGrid axis "L"
- #12712: Autoscale does not work for ax.arrow()
- #14208: shift + w does not close all figures (has no effect)
- #15745: Failed to add annotate to figure
- #11432: Pressing the "1" key kills the zoom/pan tool
- #13799: BUG: incorrect error bar colors when NaN values are present
- #16185: hist demo appears to incorrectly mention `normed` and something odd about `density` as well.
- #15203: Closing figures is done by number
- #16016: Better argument checking of subplot definition in `add_subplot()`
- #15980: Is the reset of the colorbar's edgecolor when updating the corresponding image clim wanted behaviour?
- #16718: Float figure DPI
- #16498: long string of format_coord in osx backend
- #8405: BUG: PDF export seems wrong with dash sequences that include a None offset
- #8619: Feature request: allow mathtext fallback font other than Computer Modern for custom mathtext setup
- #14996: format error saving eps figure using custom linestyle
- #16493: Example/tutorials warn due to new pcolormesh shading

- #16022: Cleanup Artist.draw() signatures
- #16389: “Size” ignored if placed before fontproperties
- #16687: Creating a figure of size (0, 0) raises an error
- #12729: Docs for contour levels argument is incorrect
- #16593: specifying ecolor in errobar turns off cycling
- #15621: secondary_xaxis doesn't seem to use formatters
- #16116: travis36minver.txt needs an update
- #16546: Problem with eventplot - error message claims events & lineoffsets are unequal sizes
- #16462: Allow wedges of polar plots to include theta = 0.
- #15142: pyplot.annotate() API deprecation
- #16479: font-stretch property missing in svg export
- #14304: 'NSWindow drag regions should only be invalidated on the Main Thread!' - macos/python
- #12085: Tcl_AsyncDelete: async handler deleted by the wrong thread
- #14669: cm.ScalarMappable should fail early when norm input is wrong
- #16468: incorrect cbar minor ticks for extend regions when x/ytick.minor.visible is True
- #16243: windows builds: devenv freetype /update appears not to have an effect
- #11525: Axes3D scatter plot for Numpy arrays in F-order does not give correct z-values
- #8894: mplot3d projection results in non-orthogonal axes
- #1104: Resizing a GUI window with Axes3D
- #16371: Incomplete documentation in axes_grid1
- #6323: Vertical alignment of tick labels with usetex=True
- #7957: clabel not respecting zorder parameter
- #16252: axes.spy plotting function doesn't respect origin='lower' kwarg when marker is not None
- #16299: The interactive polar plot animation's axis label won't scale.
- #15182: More tests `ConciseDateFormatter` needed
- #16140: Unclear Documentation for get_xticklabels
- #16147: pp.hist parmeter 'density' does not scale data appropriately
- #16069: matplotlib glitch when rotating interactively a 3d animation
- #14603: Scatterplot: should vmin/vmax be ignored when a norm is specified?

- #15730: Setting lines.marker = s in matplotlibrc also sets markers in boxplots
- #11178: home/back/forward buttons do nothing in 3d mode
- #14520: pylab with wx backend not exiting cleanly
- #15964: Guard `plt.subplot` kwargs a bit better?
- #15404: Add python 3.8 tests
- #15773: Warning:... GLib.source_remove(self._idle_draw_id) when using plt.savefig()
- #15923: pie takes negative values
- #10317: Setting plt.rc('text', usetex=True) after ticker.ScalarFormatter(useMathText=True) causes Error
- #15825: Customised dashed linstyle in matplotlib.cycler throws ValueError when using in Axes.set_prop_cycle
- #9792: Error with linestyles rcParams entries under the form (on, off, ...) and a style context manager
- #15782: Invalid polygon in stepfilled histogram when bottom is set
- #15628: Invalid unicode characters in PDF when font is a symlink
- #8577: mplot3D scalar arguments for plot function
- #15650: URL is not shown when webagg failed to open the browser.
- #5238: the offset of the scientific notation in xaxis stays at bottom when axis is set to top
- #15678: Error at save animation with pillow
- #15079: check_figures_equal decorator reuses figures if called multiple times inside a single test.
- #15089: Coerce MxNx1 images into MxN images for imshow
- #5253: abline() - for drawing arbitrary lines on a plot, given specifications.
- #15165: Switch to requiring Pillow rather than having our own png wrapper?
- #15280: Add pull request checklist to Reviewers Guidlines
- #15289: cbook.warn_deprecated() should warn with a MatplotlibDeprecation-Warning not a UserWarning
- #15285: DOC: make copy right year auto-update
- #15059: fig.add_axes() with no arguments silently does nothing
- #14546: Setting lines.markeredgecolor in rcParams affects the ticks' mark color too

7.1.4 GitHub Stats for Matplotlib 3.2.2

GitHub stats for 2020/03/18 - 2020/06/17 (tag: v3.2.1)

These lists are automatically generated, and may be incomplete or contain duplicates.

We closed 34 issues and merged 92 pull requests. The full list can be seen on GitHub

The following 19 authors contributed 183 commits.

- Antony Lee
- Ben Root
- Clemens Brunner
- David Stansby
- Elliott Sales de Andrade
- Eric Firing
- Eric Wieser
- hannah
- Jody Klymak
- Lawrence D'Anna
- Leo Singer
- Luke Davis
- Matt Newville
- Max
- Ryan May
- Sidharth Bansal
- Stefan Mitic
- Thomas A Caswell
- Tim Hoffmann

GitHub issues and pull requests:

Pull Requests (92):

- PR #17655: Auto backport of pr 17564 on v3.2.x
- PR #17564: FIX: correctly handle large arcs
- PR #17641: Qt backports
- PR #17640: More qt fractional DPI fixes
- PR #17638: V3.2.1 doc

- PR #15656: Support fractional HiDpi scaling with Qt backends
- PR #17600: FIX: work with PyQt 5.15
- PR #17598: DOC: remove banner
- PR #17618: Doc event loop
- PR #17614: DOC: Remove duplicated line.
- PR #17611: Backport #17606 to v3.2.x
- PR #17609: Backport PR #17602: FIX: propagate _is_saving state when changing can…
- PR #17606: Move codecov.yml to .github.
- PR #17602: FIX: propagate _is_saving state when changing canvases
- PR #17605: Backport PR #17560: FIX: do not let no-op monkey patches to renderer …
- PR #17601: Backport PR #16948 on branch v3.2.x (solution: All subclasses of LocationEvent could be used in cbook.callbacks before being fully initialized - issue 15139)
- PR #17560: FIX: do not let no-op monkey patches to renderer leak out
- PR #16948: solution: All subclasses of LocationEvent could be used in cbook.callbacks before being fully initialized - issue 15139
- PR #17588: Backport PR #17565: FIX: support Qt 5.15
- PR #17593: Backport PR #17587 on branch v3.2.x (Add a docstring to toolkit's BezierPath.__init__.)
- PR #17587: Add a docstring to toolkit's BezierPath.__init__.
- PR #17565: FIX: support Qt 5.15
- PR #17562: Backport PR #17470 on branch v3.2.x (FIX: add guardrails for too big tk figures)
- PR #17470: FIX: add guardrails for too big tk figures
- PR #17553: Backport PR #17552 on branch v3.2.x (ci: Add xcb libraries that were removed from PyQt5.)
- PR #17552: ci: Add xcb libraries that were removed from PyQt5.
- PR #17533: Backport PR #17408 on branch v3.2.x
- PR #17408: FIX: cancel pending autoscale on manually setting limits
- PR #17501: Backport PR #17499: Fix scatter singlecolor
- PR #17499: Fix scatter singlecolor
- PR #17468: v3.2.x: Fix leaks in C++ code
- PR #17457: Backport PR #17391 on branch v3.2.x

- PR #17391: tk/wx: Fix saving after the window is closed
- PR #17435: Backport PR #17422: Unstale viewlims before draw()ing polar axes.
- PR #17422: Unstale viewlims before draw()ing polar axes.
- PR #17407: FIX: don't try to use non-standard functions on standard status bars
- PR #17346: Backport #17084 and #17210 to v3.2.x
- PR #17084: Fix macosx segfault
- PR #17300: Backport PR #17263 on branch v3.2.x (you can't call CGDataProviderCreateWithData on a stack pointer)
- PR #17263: you can't call CGDataProviderCreateWithData on a stack pointer
- PR #17272: Backport PR #17271 on branch v3.2.x (MNT: do not try to import xml.etree.cElementTree)
- PR #17271: MNT: do not try to import xml.etree.cElementTree
- PR #17268: Backport PR #17261 on branch v3.2.x (avoid calling wx.Bitmap() if width or height is zero)
- PR #17261: avoid calling wx.Bitmap() if width or height is zero
- PR #17257: Backport eps work
- PR #17255: Fix eps + usetex combo.
- PR #17254: Backport PR #17252 on branch v3.2.x (Fix bug where matplotlib.style('default') resets the backend)
- PR #17252: Fix bug where matplotlib.style('default') resets the backend
- PR #17250: Merge pull request #17206 from jklymak/fix-bypass-inverse-collection
- PR #17206: FIX: bypass inverse in collection
- PR #17241: Backport PR #17240 on branch v3.2.x (CI: Download wx wheels for the correct Ubuntu version.)
- PR #17240: CI: Download wx wheels for the correct Ubuntu version.
- PR #17210: Fix missing attribute in _SVGConverter.
- PR #17186: Backport PR #17131 on branch v3.2.x
- PR #17188: Backport PR #16958: MAINT: Replace uses of tostring with tobytes
- PR #17187: Backport PR #17076: Fix SyntaxErrors when running setup in old Python
- PR #16913: Fix use of psfrags in ps backend + usetex.
- PR #16476: Fix baseline alignment when using usetex.
- PR #17131: BUG: Fix formatting error in GridSpec.__repr__

- PR #17132: Backport PR #17126 on branch v3.2.x (Remove Python2/3 info box)
- PR #17126: Remove Python2/3 info box
- PR #17076: Fix SyntaxErrors when running setup in old Python
- PR #17071: Backport PR #17065 on branch v3.2.x (Fix macOS CI test failure)
- PR #17065: Fix macOS CI test failure
- PR #17051: Backport PR #17045: Fix missing-references.json.
- PR #17045: Fix missing-references.json.
- PR #17020: Merge pull request #17017 from jklymak/fix-blended-transform
- PR #17017: FIX: force blended transforms with data to be in data space
- PR #16989: Backport PR #16980 on branch v3.2.x (Correctly disable more drawing methods in tight_bboxing renderer.)
- PR #16980: Correctly disable more drawing methods in tight_bboxing renderer.
- PR #16974: Backport PR #16940 on branch v3.2.x (DOC/FIX: clarify the docs for check_figures_equal)
- PR #16979: Backport PR #16970 on branch v3.2.x (tk: Don't resize toolbar during resize event.)
- PR #16970: tk: Don't resize toolbar during resize event.
- PR #16940: DOC/FIX: clarify the docs for check_figures_equal
- PR #16969: Backport PR #16966 on branch v3.2.x (Fix animation writer fallback.)
- PR #16966: Fix animation writer fallback.
- PR #16958: MAINT: Replace uses of tostring with tobytes
- PR #16950: Backport PR #16949 on branch v3.2.x (TST: Don't modify actual pyplot file for boilerplate test.)
- PR #16949: TST: Don't modify actual pyplot file for boilerplate test.
- PR #16932: Backport PR #16929 on branch v3.2.x (tk: Resize the canvas, not the figure.)
- PR #16929: tk: Resize the canvas, not the figure.
- PR #16880: Backport PR #16870: Unbreak CI by xfailing wxAgg test on macOS
- PR #16870: Unbreak CI by xfailing wxAgg test on macOS
- PR #16869: Backport PR #16867 on branch v3.2.x (BLD: Auto-trigger macOS/Linux wheels on tags.)
- PR #16867: BLD: Auto-trigger macOS/Linux wheels on tags.
- PR #16852: Backport PR #16851 on branch v3.2.x (DOC: Fix docstring of Axes.secondary_yaxis.)

- PR #16855: Fix typo in deprecation warning
- PR #16851: DOC: Fix docstring of Axes.secondary_yaxis.
- PR #16842: Backport PR #16835 on branch v3.2.x (Don't forget to export isdeleted on Qt4.)
- PR #16835: Don't forget to export isdeleted on Qt4.
- PR #15695: Define mathdefault as a noop in the usetex preamble.
- PR #14694: Vectorize Arc.draw.

Issues (34):

- #17547: Arcs with large radii in small
- #17440: Low quality window plots on hidpi display
- #17104: input() caused _tkinter.TclError: invalid command name XXX after plot.close()
- #17613: Matplotlib.pdf duplication
- #15139: All subclasses of LocationEvent could be used in cbook.callbacks before being fully initialized
- #17004: Output regression in 3.2 that affects SymPy's plotting
- #17599: Saving issue with pdf backend
- #17542: Matplotlib 3.2.1 savefig empty image when fig size matches data size exactly
- #17594: Cannot use Qt4Agg backend in mpl 3.2.1
- #17460: set_size_inches with a width over 14090 crashes Xorg
- #17331: Surprising/changed axis limit (autoscale) behavior
- #17423: Scatter produce multiple colors for a single RGB/RGBA input
- #17385: Matplotlib memory leaks when save figure in a file with qt5 backend
- #15474: Memory leak with log scale in pcolorfast, pcolormesh, imshow ...
- #17388: savefig error: tkinter.TclError: invalid command name "."
- #16909: plot save and plot show
- #17085: set_function not working properly in backend_wx
- #17418: Issue rendering polar plot (agg backend?) with rorigin set
- #17061: Segmentation fault with macosx backend
- #17253: EPS + usetex is broken
- #16700: Deprecation warnings from stylelib
- #17203: Subplots using bad axis limits in 3.2

- #16898: EPS and `usetex` give blank output
- #16409: Confusing error on fully commented-out usetex strings
- #17075: Installation error downloading jquery on python3 on Ubuntu
- #17037: Travis Failing in many PRs
- #17033: Using a `TextBox` in current master produces a seemingly unrelated warning.
- #17016: Issues with autoscaling and transforms with 3.2+
- #16978: savefig("myplot.svgz", bbox_inches="tight") fails
- #16965: FuncAnimation.save throws TypeError
- #16916: check_figures_equal regression from 3.2.0 to 3.2.1
- #10566: blocking UI functions cause figure size to change
- #10083: Wrong figure height after set_size_inches within event handler
- #16834: Error importing FigureCanvas

7.1.5 GitHub Stats for Matplotlib 3.2.1

GitHub stats for 2020/03/03 - 2020/03/17 (tag: v3.2.0)

These lists are automatically generated, and may be incomplete or contain duplicates.

We closed 11 issues and merged 52 pull requests. The full list can be seen on GitHub and on GitHub

The following 12 authors contributed 154 commits.

- Amy Roberts
- Antony Lee
- Elliott Sales de Andrade
- hannah
- Hugo van Kemenade
- Jody Klymak
- Kyle Sunden
- MarcoGorelli
- Maximilian Nöthe
- Sandro Tosi
- Thomas A Caswell
- Tim Hoffmann

GitHub issues and pull requests:

Pull Requests (52):

- PR #15199: MNT/TST: generalize check_figures_equal to work with pytest.marks
- PR #15685: Avoid a RuntimeError at animation shutdown with PySide2.
- PR #15969: Restart pgf's latex instance after bad latex inputs.
- PR #16640: ci: Fix Azure on v3.2.x
- PR #16648: Document filling of Poly3DCollection
- PR #16649: Fix typo in docs
- PR #16650: Backport PR #16649 on branch v3.2.x (Fix typo in docs)
- PR #16651: Docs: Change Python 2 note to past tense
- PR #16654: Backport PR #16651 on branch v3.2.0-doc (Docs: Change Python 2 note to past tense)
- PR #16656: Make test_imagegrid_cbar_mode_edge less flaky.
- PR #16661: added Framework :: Matplotlib to setup
- PR #16665: Backport PR #16661 on branch v3.2.x (added Framework :: Matplotlib to setup)
- PR #16671: Fix some readme bits
- PR #16672: Update CircleCI and add direct artifact link
- PR #16682: Avoid floating point rounding causing bezier.get_parallels to fail
- PR #16690: Backport PR #16682 on branch v3.2.x (Avoid floating point rounding causing bezier.get_parallels to fail)
- PR #16693: TST: use pytest name in naming files for check_figures_equal
- PR #16695: Restart pgf's latex instance after bad latex inputs.
- PR #16705: Backport PR #16656 on branch v3.2.x (Make test_imagegrid_cbar_mode_edge less flaky.)
- PR #16708: Backport PR #16671: Fix some readme bits
- PR #16709: Fix saving PNGs to file objects in some places
- PR #16722: Deprecate rcParams["datapath"] in favor of mpl.get_data_path().
- PR #16725: TST/CI: also try to run test_user_fonts_win32 on azure
- PR #16734: Disable draw_foo methods on renderer used to estimate tight extents.
- PR #16735: Make test_stem less flaky.
- PR #16736: xpdf: Set AutoRotatePages to None, not false.

- PR #16742: nbagg: Don't send events if manager is disconnected.
- PR #16745: Allow numbers to set uvc for all arrows in quiver.set_UVC, fixes #16743
- PR #16751: Backport PR #16742 on branch v3.2.x (nbagg: Don't send events if manager is disconnected.)
- PR #16752: ci: Disallow pytest 5.4.0, which is crashing.
- PR #16753: Backport #16752 to v3.2.x
- PR #16760: Backport PR #16735 on branch v3.2.x (Make test_stem less flaky.)
- PR #16761: Backport PR #16745 on branch v3.2.x (Allow numbers to set uvc for all arrows in quiver.set_UVC, fixes #16743)
- PR #16763: Backport PR #16648 on branch v3.2.x (Document filling of Poly3DCollection)
- PR #16764: Backport PR #16672 on branch v3.2.0-doc
- PR #16765: Backport PR #16736 on branch v3.2.x (xpdf: Set AutoRotatePages to None, not false.)
- PR #16766: Backport PR #16734 on branch v3.2.x (Disable draw_foo methods on renderer used to estimate tight extents.)
- PR #16767: Backport PR #15685 on branch v3.2.x (Avoid a RuntimeError at animation shutdown with PySide2.)
- PR #16768: Backport PR #16725 on branch v3.2.x (TST/CI: also try to run test_user_fonts_win32 on azure)
- PR #16770: Fix tuple markers
- PR #16779: Documentation: make instructions for documentation contributions easier to find, add to requirements for building docs
- PR #16784: Update CircleCI URL for downloading humor-sans.ttf.
- PR #16790: Backport PR #16784 on branch v3.2.x (Update CircleCI URL for downloading humor-sans.ttf.)
- PR #16791: Backport PR #16770 on branch v3.2.x (Fix tuple markers)
- PR #16794: DOC: Don't mention drawstyle in `set_linestyle` docs.
- PR #16795: Backport PR #15199 on branch v3.2.x (MNT/TST: generalize check_figures_equal to work with pytest.marks)
- PR #16797: Backport #15589 and #16693, fixes for check_figures_equal
- PR #16799: Backport PR #16794 on branch v3.2.0-doc (DOC: Don't mention drawstyle in `set_linestyle` docs.)
- PR #16800: Fix check_figures_equal for tests that use its fixtures.
- PR #16803: Fix some doc issues

- PR #16806: Backport PR #16803 on branch v3.2.0-doc (Fix some doc issues)
- PR #16809: Backport PR #16779 on branch v3.2.0-doc (Documentation: make instructions for documentation contributions easier to find, add to requirements for building docs)

Issues (11):

- #12820: [Annotations] ValueError: lines do not intersect when computing tight bounding box containing arrow with filled paths
- #16538: xpdf distiller seems broken
- #16624: Azure pipelines are broken on v3.2.x
- #16633: Wrong drawing Poly3DCollection
- #16645: Minor typo in API document of patches.ConnectionPatch
- #16670: BLD: ascii codec decode on 3.2.0 in non-UTF8 locales
- #16704: 3.2.0: `setup.py clean` fails with `NameError: name 'long_description' is not defined`
- #16721: nbAgg backend does not allow saving figures as png
- #16731: PGF backend + savefig.bbox results in I/O error in 3.2
- #16743: Breaking change in 3.2: quiver.set_UVC does not support single numbers any more
- #16801: Doc: figure for colormaps off

7.1.6 GitHub Stats for Matplotlib 3.2.0

GitHub stats for 2019/05/18 - 2020/03/03 (tag: v3.1.0)

These lists are automatically generated, and may be incomplete or contain duplicates.

We closed 125 issues and merged 839 pull requests. The full list can be seen on GitHub

The following 164 authors contributed 3455 commits.

- Abhinav Sagar
- Abhinuv Nitin Pitale
- Adam Gomaa
- Akshay Nair
- Alex Rudy
- Alexander Rudy
- Antony Lee
- Ao Liu (frankliuao)

- Ardie Orden
- Ashley Whetter
- Ben Root
- Benjamin Bengfort
- Benjamin Congdon
- Bharat123rox
- Bingyao Liu
- Brigitta Sipocz
- Bruno Pagani
- brut
- Carsten
- Carsten Schelp
- chaoyi1
- Cho Yin Yong
- Chris Barnes
- Christer Jensen
- Christian Brodbeck
- Christoph Pohl
- chuanzhu xu
- Colin
- Cong Ma
- dabana
- DanielMatu
- David Chudzicki
- David Stansby
- Deng Tian
- depano.carlos@gmail.com
- djdt
- donchanee
- Dora Fraeman Caswell
- Elan Ernest
- Elliott Sales de Andrade

- Emlyn Price
- Eric Firing
- Eric Wieser
- Federico Ariza
- Filipe Fernandes
- fourpoints
- fredrik-1
- Gazing
- Greg Lucas
- hannah
- Harshal Prakash Patankar
- Ian Hincks
- Ian Thomas
- ilopata1
- ImportanceOfBeingErnest
- Jacobson Okoro
- James A. Bednar
- Jarrod Millman
- Javad
- jb-leger
- Jean-Benoist Leger
- jfbu
- joaonsg
- Jody Klymak
- Joel Frederico
- Johannes H. Jensen
- Johnny Gill
- Jonas Camillus Jeppesen
- Jorge Moraleda
- Joscha Reimer
- Joseph Albert
- Jouni K. Seppänen

- Joy Bhalla
- Juanjo Bazán
- Julian Mehne
- kolibril13
- krishna katyal
- ksunden
- Kyle Sunden
- Larry Bradley
- lepuchi
- luftek
- Maciej Dems
- Maik Riechert
- Marat K
- Mark Wolf
- Mark Wolfman
- Matte
- Matthias Bussonnier
- Matthias Geier
- MatthieuDartiailh
- Max Chen
- Max Humber
- Max Shinn
- MeeseeksMachine
- Michael Droettboom
- Mingkai Dong
- MinRK
- miquelastein
- Molly Rossow
- Nathan Goldbaum
- nathan78906
- Nelle Varoquaux
- Nick White

- Nicolas Courtemanche
- Nikita Kniazev
- njwhite
- O. Castany
- Oliver Natt
- Olivier
- Om Sitapara
- omsitapara23
- Oriol (Prodesk)
- Oriol Abril
- Patrick Feiring
- Patrick Shriwise
- PatrickFeiring
- Paul
- Paul Hobson
- Paul Hoffman
- Paul Ivanov
- Peter Schutt
- pharshalp
- Phil Elson
- Philippe Pinard
- Rebecca W Perry
- ResidentMario
- Richard Ji-Cathriner
- RoryIAngus
- Ryan May
- S. Fukuda
- Samesh
- Samesh Lakhotia
- sasoripathos
- SBCV
- Sebastian Bullinger

- Sergey Royz
- Siddhesh Poyarekar
- Simon Legner
- SojiroFukuda
- Steve Dower
- Taras
- Ted Drain
- teddyrendahl
- Thomas A Caswell
- Thomas Hisch
- Thomas Robitaille
- Till Hoffmann
- tillahoffmann
- Tim Hoffmann
- Tom Flannaghan
- Travis CI
- V. Armando Solé
- Vincent L.M. Mazoyer
- Viraj Mohile
- Wafa Soofi
- Warren Weckesser
- y1thof
- yeo
- Yong Cho Yin
- Yuya
- Zhili (Jerry) Pan
- zhoubecky
- Zulko

GitHub issues and pull requests:

Pull Requests (839):

- PR #16626: Updated Readme + Setup.py for PyPa
- PR #16627: ci: Restore nuget install step on Azure for v3.2.x.

- PR #16625: v3.2.x: Make Azure use local FreeType.
- PR #16622: Backport PR #16613 on branch v3.2.x (Fix edge-case in preprocess_data, if label_namer is optional and unset.)
- PR #16613: Fix edge-case in preprocess_data, if label_namer is optional and unset.
- PR #16612: Backport PR #16605: CI: tweak the vm images we use on azure
- PR #16611: Backport PR #16585 on branch v3.2.x (Fix _preprocess_data for Py3.9.)
- PR #16605: CI: tweak the vm images we use on azure
- PR #16585: Fix _preprocess_data for Py3.9.
- PR #16541: Merge pull request #16404 from jklymak/fix-add-base-symlognorm
- PR #16542: Backport PR #16006: Ignore pos in StrCategoryFormatter.__call__ to di…
- PR #16543: Backport PR #16532: Document default value of save_count parameter in…
- PR #16532: Document default value of save_count parameter in FuncAnimation
- PR #16526: Backport PR #16480 on v.3.2.x: Re-phrase doc for bottom kwarg to hist
- PR #16404: FIX: add base kwarg to symlognor
- PR #16518: Backport PR #16502 on branch v3.2.x (Document theta getters/setters)
- PR #16519: Backport PR #16513 on branch v3.2.x (Add more FreeType tarball hashes.)
- PR #16513: Add more FreeType tarball hashes.
- PR #16502: Document theta getters/setters
- PR #16506: Backport PR #16505 on branch v3.2.x (Add link to blog to front page)
- PR #16505: Add link to blog to front page
- PR #16480: Re-phrase doc for bottom kwarg to hist
- PR #16494: Backport PR #16490 on branch v3.2.x (Fix some typos on the front page)
- PR #16489: Backport PR #16272 on branch v3.2.x (Move mplot3d autoregistration api changes to 3.2.)
- PR #16490: Fix some typos on the front page
- PR #16465: Backport PR #16450 on branch v3.2.x (Fix interaction between sticky_edges and shared axes.)

- PR #16466: Backport PR #16392: FIX colorbars for Norms that do not have a scale.
- PR #16392: FIX colorbars for Norms that do not have a scale.
- PR #16450: Fix interaction between sticky_edges and shared axes.
- PR #16453: Backport PR #16452 on branch v3.2.x (Don't make InvertedLogTransform inherit from deprecated base class.)
- PR #16452: Don't make InvertedLogTransform inherit from deprecated base class.
- PR #16436: Backport PR #16435 on branch v3.2.x (Reword intro to colors api docs.)
- PR #16435: Reword intro to colors api docs.
- PR #16399: Backport PR #16396 on branch v3.2.x (font_manager docs cleanup.)
- PR #16396: font_manager docs cleanup.
- PR #16397: Backport PR #16394 on branch v3.2.x (Mark inkscape 1.0 as unsupported (at least for now).)
- PR #16394: Mark inkscape 1.0 as unsupported (at least for now).
- PR #16286: Fix cbars for different norms
- PR #16385: Backport PR #16226 on branch v3.2.x: Reorganize intro section on main page
- PR #16383: Backport PR #16379 on branch v3.2.x (FIX: catch on message content, not module)
- PR #16226: Reorganize intro section on main page
- PR #16364: Backport PR #16344 on branch v3.2.x (Cast vmin/vmax to floats before nonsingular-expanding them.)
- PR #16344: Cast vmin/vmax to floats before nonsingular-expanding them.
- PR #16360: Backport PR #16347 on branch v3.2.x (FIX: catch warnings from pandas in cbook._check_1d)
- PR #16357: Backport PR #16330 on branch v3.2.x (Clearer signal handling)
- PR #16349: Backport PR #16255 on branch v3.2.x (Move version info to sidebar)
- PR #16346: Backport PR #16298 on branch v3.2.x (Don't recursively call draw_idle when updating artists at draw time.)
- PR #16331: Backport PR #16308 on branch v3.2.x (CI: Use Ubuntu Bionic compatible package names)
- PR #16332: Backport PR #16308 on v3.2.x: CI: Use Ubuntu Bionic compatible package names

- PR #16324: Backport PR #16323 on branch v3.2.x (Add sphinx doc for Axis.axis_name.)
- PR #16325: Backport PR #15462 on v3.2.x: Simplify azure setup.
- PR #16323: Add sphinx doc for Axis.axis_name.
- PR #16321: Backport PR #16311 on branch v3.2.x (don't override non-Python signal handlers)
- PR #16308: CI: Use Ubuntu Bionic compatible package names
- PR #16306: Backport PR #16300 on branch v3.2.x (Don't default to negative radii in polar plot.)
- PR #16305: Backport PR #16250 on branch v3.2.x (Fix zerolen intersect)
- PR #16300: Don't default to negative radii in polar plot.
- PR #16278: Backport PR #16273 on branch v3.2.x (DOC: Changing the spelling of co-ordinates.)
- PR #16260: Backport PR #16259 on branch v3.2.x (TST: something changed in pytest 5.3.3 that breaks our qt fixtures)
- PR #16259: TST: something changed in pytest 5.3.3 that breaks our qt fixtures
- PR #16238: Backport PR #16235 on branch v3.2.x (FIX: AttributeError in TimerBase.start)
- PR #16211: DOC: ValidateInterval was deprecated in 3.2, not 3.1
- PR #16224: Backport PR #16223 on branch v3.2.x (Added DNA Features Viewer description + screenshot in docs/thirdparty/)
- PR #16223: Added DNA Features Viewer description + screenshot in docs/thirdparty/
- PR #16222: Backport PR #16212 on branch v3.2.x (Fix deprecation from #13544)
- PR #16212: Fix deprecation from #13544
- PR #16207: Backport PR #16189 on branch v3.2.x (MNT: set default canvas when un-pickling)
- PR #16189: MNT: set default canvas when un-pickling
- PR #16179: Backport PR #16175: FIX: ignore axes that aren't visible
- PR #16175: FIX: ignore axes that aren't visible
- PR #16168: Backport PR #16166 on branch v3.2.x (Add badge for citing 3.1.2)
- PR #16148: Backport PR #16128 on branch v3.2.x (CI: Do not use nbformat 5.0.0/5.0.1 for testing)
- PR #16145: Backport PR #16053 on branch v3.2.x (Fix v_interval setter)
- PR #16128: CI: Do not use nbformat 5.0.0/5.0.1 for testing

- PR #16135: Backport PR #16112 on branch v3.2.x (CI: Fail when failed to install dependencies)
- PR #16132: Backport PR #16126 on branch v3.2.x (TST: test_fork: Missing join)
- PR #16124: Backport PR #16105 on branch v3.2.x (Fix legend dragging.)
- PR #16122: Backport PR #16113 on branch v3.2.x (Renderer Graphviz inheritance diagrams as svg)
- PR #16105: Fix legend dragging.
- PR #16113: Renderer Graphviz inheritance diagrams as svg
- PR #16112: CI: Fail when failed to install dependencies
- PR #16119: Backport PR #16065 on branch v3.2.x (Nicer formatting of community aspects on front page)
- PR #16074: Backport PR #16061 on branch v3.2.x (Fix deprecation message for axes_grid1.colorbar.)
- PR #16093: Backport PR #16079 on branch v3.2.x (Fix restuctured text formatting)
- PR #16094: Backport PR #16080 on branch v3.2.x (Cleanup docstrings in backend_bases.py)
- PR #16086: FIX: use supported attribute to check pillow version
- PR #16084: Backport PR #16077 on branch v3.2.x (Fix some typos)
- PR #16077: Fix some typos
- PR #16079: Fix restuctured text formatting
- PR #16080: Cleanup docstrings in backend_bases.py
- PR #16061: Fix deprecation message for axes_grid1.colorbar.
- PR #16006: Ignore pos in StrCategoryFormatter.__call__ to display correct label in the preview window
- PR #16056: Backport PR #15864 on branch v3.2.x ([Add the info of 'sviewgui' in thirdparty package])
- PR #15864: Add 'sviewgui' to list of thirdparty packages
- PR #16055: Backport PR #16037 on branch v3.2.x (Doc: use empty ScalarMappable for colorbars with no associated image.)
- PR #16054: Backport PR #16048 on branch v3.2.x (Document that colorbar() takes a label kwarg.)
- PR #16037: Doc: use empty ScalarMappable for colorbars with no associated image.
- PR #16048: Document that colorbar() takes a label kwarg.
- PR #16042: Backport PR #16031 on branch v3.2.x (Fix docstring of hillshade().)

- PR #16033: Backport PR #16028 on branch v3.2.x (Prevent FigureCanvasQT_draw_idle recursively calling itself.)
- PR #16021: Backport PR #16007 on branch v3.2.x (Fix search on nested pages)
- PR #16019: Backport PR #15735 on branch v3.2.x (Cleanup some mplot3d docstrings.)
- PR #15987: Backport PR #15886 on branch v3.2.x (Fix Annotation using different units and different coords on x/y.)
- PR #15886: Fix Annotation using different units and different coords on x/y.
- PR #15984: Backport PR #15970 on branch v3.2.x (Process clip paths the same way as regular Paths.)
- PR #15970: Process clip paths the same way as regular Paths.
- PR #15963: Backport PR #15937 on branch v3.2.x (Don't hide exceptions in FontManager.addfont.)
- PR #15956: Backport PR #15901 on branch v3.2.x (Update backend_nbagg for removal of Gcf._activeQue.)
- PR #15937: Don't hide exceptions in FontManager.addfont.
- PR #15959: Backport PR #15953 on branch v3.2.x (Update donation link)
- PR #15901: Update backend_nbagg for removal of Gcf._activeQue.
- PR #15954: Backport PR #15914 on branch v3.2.x (Example for sigmoid function with horizontal lines)
- PR #15914: Example for sigmoid function with horizontal lines
- PR #15930: Backport PR #15925 on branch v3.2.x (Optimize setting units to None when they're already None.)
- PR #15925: Optimize setting units to None when they're already None.
- PR #15915: Backport PR #15903 on branch v3.2.x (Correctly handle non-affine transData in Collection.get_datalim.)
- PR #15903: Correctly handle non-affine transData in Collection.get_datalim.
- PR #15908: Backport PR #15857 on branch v3.2.x (LassoSelection shouldn't useblit on canvas not supporting blitting.)
- PR #15857: LassoSelection shouldn't useblit on canvas not supporting blitting.
- PR #15905: Backport PR #15763 on branch v3.2.x (Skip webagg test if tornado is not available.)
- PR #15882: Backport PR #15859 on branch v3.2.x (Doc: Move search field into nav bar)
- PR #15868: Backport PR #15848 on branch v3.2.x: Cleanup environment variables FAQ

- PR #15872: Backport PR #15869 on branch v3.2.x (Update markers docs.)
- PR #15869: Update markers docs.
- PR #15867: Backport PR #15789 on branch v3.2.x (Cleanup xticks/yticks docstrings.)
- PR #15870: Backport PR #15865 on branch v3.2.x (Fix a typo)
- PR #15871: Backport PR #15824 on branch v3.2.x (Document doc style for default values)
- PR #15824: Document doc style for default values
- PR #15865: Fix a typo
- PR #15789: Cleanup xticks/yticks docstrings.
- PR #15862: Backport PR #15851 on branch v3.2.x (ffmpeg is available on default ubuntu packages now)
- PR #15848: Cleanup environment variables FAQ.
- PR #15844: Backport PR #15841 on branch v3.2.x (DOC: specify the expected shape in the Collection.set_offset)
- PR #15841: DOC: specify the expected shape in the Collection.set_offset
- PR #15837: Backport PR #15799 on branch v3.2.x (Improve display of author names on PDF titlepage of matplotlib own docs)
- PR #15799: Improve display of author names on PDF titlepage of matplotlib own docs
- PR #15831: Backport PR #15829 on branch v3.2.x (In C extensions, use FutureWarning, not DeprecationWarning.)
- PR #15829: In C extensions, use FutureWarning, not DeprecationWarning.
- PR #15818: Backport PR #15619 on branch v3.2.x (Improve zorder demo)
- PR #15819: Backport PR #15601 on branch v3.2.x (Fix FontProperties conversion to/from strings)
- PR #15601: Fix FontProperties conversion to/from strings
- PR #15619: Improve zorder demo
- PR #15810: Backport PR #15809 on branch v3.2.x (Exclude artists from legend using label attributte)
- PR #15809: Exclude artists from legend using label attributte
- PR #15808: Backport PR #15513 on branch v3.2.x (Separate plots using #### in make_room_for_ylabel_using_axesgrid.py)
- PR #15513: Separate plots using #### in make_room_for_ylabel_using_axesgrid.py
- PR #15807: Backport PR #15791 on branch v3.2.x (Cleanup backend_bases docstrings.)

- PR #15791: Cleanup backend_bases docstrings.
- PR #15803: Backport PR #15795 on branch v3.2.x (Remove incorrect statement re2: colorbars in image tutorial.)
- PR #15795: Remove incorrect statement re: colorbars in image tutorial.
- PR #15794: Backport PR #15793 on branch v3.2.x (fix a couple typos in tutorials)
- PR #15793: fix a couple typos in tutorials
- PR #15774: Backport PR #15748 on branch v3.2.x (Fix incorrect macro in FT2Font setup.)
- PR #15748: Fix incorrect macro in FT2Font setup.
- PR #15759: Backport PR #15751 on branch v3.2.x (Modernize FAQ entry for plt.show().)
- PR #15762: Backport PR #15752 on branch v3.2.x (Update boxplot/violinplot faq.)
- PR #15755: Backport PR #15661 on branch v3.2.x (Document scope of 3D scatter depthshading.)
- PR #15742: Backport PR #15729 on branch v3.2.x (Catch correct parse error type for dateutil >= 2.8.1)
- PR #15738: Backport PR #15737 on branch v3.2.x (Fix env override in WebAgg backend test.)
- PR #15724: Backport PR #15718 on branch v3.2.x (Update donation link)
- PR #15716: Backport PR #15683 on branch v3.2.x (Cleanup dates.py docstrings.)
- PR #15683: Cleanup dates.py docstrings.
- PR #15688: Backport PR #15682 on branch v3.2.x (Make histogram_bin_edges private.)
- PR #15682: Make histogram_bin_edges private.
- PR #15666: Backport PR #15649 on branch v3.2.x (Fix searchindex.js loading when ajax fails (because e.g. CORS in embedded iframes))
- PR #15669: Backport PR #15654 on branch v3.2.x (Fix some broken links.)
- PR #15660: Backport PR #15647 on branch v3.2.x (Update some links)
- PR #15653: Backport PR #15623 on branch v3.2.x (Docstring for Artist.mouseover)
- PR #15623: Docstring for Artist.mouseover
- PR #15634: Backport PR #15626 on branch v3.2.x (Note minimum supported version for fontconfig.)

- PR #15633: Backport PR #15620 on branch v3.2.x (TST: Increase tolerance of some tests for aarch64)
- PR #15626: Note minimum supported version for fontconfig.
- PR #15632: Backport PR #15627 on branch v3.2.x (Make it easier to test various animation writers in examples.)
- PR #15620: TST: Increase tolerance of some tests for aarch64
- PR #15627: Make it easier to test various animation writers in examples.
- PR #15618: Backport PR #15613 on branch v3.2.x (Revert "Don't bother with manually resizing the Qt main window.")
- PR #15613: Revert "Don't bother with manually resizing the Qt main window."
- PR #15593: Backport PR #15590 on branch v3.2.x (Rename numpy to NumPy in docs.)
- PR #15590: Rename numpy to NumPy in docs.
- PR #15588: Backport PR #15478 on branch v3.2.x (Make ConciseDateFormatter obey timezone)
- PR #15478: Make ConciseDateFormatter obey timezone
- PR #15583: Backport PR #15512 on branch v3.2.x
- PR #15584: Backport PR #15579 on branch v3.2.x (Remove matplotlib.sphinxext.tests from __init__.py)
- PR #15579: Remove matplotlib.sphinxext.tests from __init__.py
- PR #15577: Backport PR #14705 on branch v3.2.x (Correctly size non-ASCII characters in agg backend.)
- PR #14705: Correctly size non-ASCII characters in agg backend.
- PR #15572: Backport PR #15452 on branch v3.2.x (Improve example for tick formatters)
- PR #15570: Backport PR #15561 on branch v3.2.x (Update thirdparty scalebar)
- PR #15452: Improve example for tick formatters
- PR #15545: Backport PR #15429 on branch v3.2.x (Fix OSX build on azure)
- PR #15544: Backport PR #15537 on branch v3.2.x (Add a third party package in the doc: matplotlib-scalebar)
- PR #15561: Update thirdparty scalebar
- PR #15567: Backport PR #15562 on branch v3.2.x (Improve docsting of AxesImage)
- PR #15562: Improve docsting of AxesImage
- PR #15565: Backport PR #15556 on branch v3.2.x (Fix test suite compat with ghostscript 9.50.)

- PR #15556: Fix test suite compat with ghostscript 9.50.
- PR #15560: Backport PR #15553 on branch v3.2.x (DOC: add cache-buster query string to css path)
- PR #15552: Backport PR #15528 on branch v3.2.x (Declutter home page)
- PR #15554: Backport PR #15523 on branch v3.2.x (numpydoc AxesImage)
- PR #15523: numpydoc AxesImage
- PR #15549: Backport PR #15516 on branch v3.2.x (Add logo like font)
- PR #15543: Backport PR #15539 on branch v3.2.x (Small cleanups to backend docs.)
- PR #15542: Backport PR #15540 on branch v3.2.x (axisartist tutorial fixes.)
- PR #15537: Add a third party package in the doc: matplotlib-scalebar
- PR #15541: Backport PR #15533 on branch v3.2.x (Use svg instead of png for website logo)
- PR #15539: Small cleanups to backend docs.
- PR #15540: axisartist tutorial fixes.
- PR #15538: Backport PR #15535 on branch v3.2.x (Avoid really long lines in event handling docs.)
- PR #15535: Avoid really long lines in event handling docs.
- PR #15531: Backport PR #15527 on branch v3.2.x (Clarify imshow() docs concerning scaling and grayscale images)
- PR #15527: Clarify imshow() docs concerning scaling and grayscale images
- PR #15522: Backport PR #15500 on branch v3.2.x (Improve antialiasing example)
- PR #15524: Backport PR #15499 on branch v3.2.x (Do not show path in font table example)
- PR #15525: Backport PR #15498 on branch v3.2.x (Simplify matshow example)
- PR #15498: Simplify matshow example
- PR #15499: Do not show path in font table example
- PR #15521: Backport PR #15519 on branch v3.2.x (FIX: fix anti-aliasing zoom bug)
- PR #15500: Improve antialiasing example
- PR #15519: FIX: fix anti-aliasing zoom bug
- PR #15510: Backport PR #15489 on branch v3.2.x (DOC: adding main nav to site)

- PR #15495: Backport PR #15486 on branch v3.2.x (Fixes an error in the documentation of Ellipse)
- PR #15488: Backport PR #15372 on branch v3.2.x (Add example for drawstyle)
- PR #15490: Backport PR #15487 on branch v3.2.x (Fix window not always raised in Qt example)
- PR #15487: Fix window not always raised in Qt example
- PR #15372: Add example for drawstyle
- PR #15485: Backport PR #15454 on branch v3.2.x (Rewrite Anscombe's quartet example)
- PR #15483: Backport PR #15480 on branch v3.2.x (Fix wording in [packages] section of setup.cfg)
- PR #15454: Rewrite Anscombe's quartet example
- PR #15480: Fix wording in [packages] section of setup.cfg
- PR #15477: Backport PR #15464 on branch v3.2.x (Remove unused code (remainder from #15453))
- PR #15471: Backport PR #15460 on branch v3.2.x (Fix incorrect value check in axes_grid.)
- PR #15456: Backport PR #15453 on branch v3.2.x (Improve example for tick locators)
- PR #15457: Backport PR #15450 on branch v3.2.x (API: rename DivergingNorm to TwoSlopeNorm)
- PR #15450: API: rename DivergingNorm to TwoSlopeNorm
- PR #15434: In imsave, let pnginfo have precedence over metadata.
- PR #15445: Backport PR #15439 on branch v3.2.x (DOC: mention discourse main page)
- PR #15425: Backport PR #15422 on branch v3.2.x (FIX: typo in attribute lookup)
- PR #15449: DOC: fix build
- PR #15429: Fix OSX build on azure
- PR #15420: Backport PR #15380 on branch v3.2.x (Update docs of BoxStyle)
- PR #15380: Update docs of BoxStyle
- PR #15300: CI: use python -m to make sure we are using the pip/pytest we want
- PR #15414: Backport PR #15413 on branch v3.2.x (catch OSError instead of FileNotFoundError in _get_executable_info to resolve #15399)
- PR #15413: catch OSError instead of FileNotFoundError in _get_executable_info to resolve #15399
- PR #15406: Backport PR #15347 on branch v3.2.x (Fix axes.hist bins units)

- PR #15405: Backport PR #15391 on branch v3.2.x (Increase fontsize in inheritance graphs)
- PR #15347: Fix axes.hist bins units
- PR #15391: Increase fontsize in inheritance graphs
- PR #15389: Backport PR #15379 on branch v3.2.x (Document formatting strings in the docs)
- PR #15379: Document formatting strings in the docs
- PR #15386: Backport PR #15385 on branch v3.2.x (Reword hist() doc.)
- PR #15385: Reword hist() doc.
- PR #15377: Backport PR #15357 on branch v3.2.x (Add 'step' and 'barstacked' to histogram_histtypes demo)
- PR #15357: Add 'step' and 'barstacked' to histogram_histtypes demo
- PR #15366: Backport PR #15364 on branch v3.2.x (DOC: fix typo in colormap docs)
- PR #15362: Backport PR #15350 on branch v3.2.x (Don't generate double-reversed cmaps ("viridis_r_r", ...).)
- PR #15360: Backport PR #15258 on branch v3.2.x (Don't fallback to view limits when autoscale()ing no data.)
- PR #15350: Don't generate double-reversed cmaps ("viridis_r_r", ...).
- PR #15258: Don't fallback to view limits when autoscale()ing no data.
- PR #15299: Backport PR #15296 on branch v3.2.x (Fix typo/bug from 18cecf7)
- PR #15327: Backport PR #15326 on branch v3.2.x (List of minimal versions of dependencies)
- PR #15326: List of minimal versions of dependencies
- PR #15317: Backport PR #15291 on branch v3.2.x (Remove error_msg_qt from backend_qt4.)
- PR #15316: Backport PR #15283 on branch v3.2.x (Don't default axes_grid colorbar locator to MaxNLocator.)
- PR #15291: Remove error_msg_qt from backend_qt4.
- PR #15283: Don't default axes_grid colorbar locator to MaxNLocator.
- PR #15315: Backport PR #15308 on branch v3.2.x (Doc: Add close event to list of events)
- PR #15308: Doc: Add close event to list of events
- PR #15312: Backport PR #15307 on branch v3.2.x (DOC: center footer)
- PR #15307: DOC: center footer

- PR #15276: Backport PR #15271 on branch v3.2.x (Fix font weight validation)
- PR #15279: Backport PR #15252 on branch v3.2.x (Mention labels and milestones in PR review guidelines)
- PR #15252: Mention labels and milestones in PR review guidelines
- PR #15268: Backport PR #15266 on branch v3.2.x (Embedding in Tk example: Fix toolbar being clipped.)
- PR #15269: Backport PR #15267 on branch v3.2.x (added multi-letter example to mathtext tutorial)
- PR #15267: added multi-letter example to mathtext tutorial
- PR #15266: Embedding in Tk example: Fix toolbar being clipped.
- PR #15243: Move some new API changes to the correct place
- PR #15245: Fix incorrect calls to warn_deprecated.
- PR #15239: Composite against white, not the savefig.facecolor rc, in print_jpeg.
- PR #15227: contains_point() docstring fixes
- PR #15242: Cleanup widgets docstrings.
- PR #14889: Support pixel-by-pixel alpha in imshow.
- PR #14928: Logit scale nonsingular
- PR #14998: Fix nonlinear spine positions & inline Spine._calc_offset_transform into get_spine_transform.
- PR #15231: Doc: Do not write default for non-existing rcParams
- PR #15222: Cleanup projections/__init__.py.
- PR #15228: Minor docstring style cleanup
- PR #15237: Cleanup widgets.py.
- PR #15229: Doc: Fix Bbox and BboxBase links
- PR #15235: Kill FigureManagerTk._num.
- PR #15234: Drop mention of msinttypes in Windows build.
- PR #15224: Avoid infinite loop when switching actions in qt backend.
- PR #15230: Doc: Remove hard-documented rcParams defaults
- PR #15149: pyplot.style.use() to accept pathlib.Path objects as arguments
- PR #15220: Correctly format floats passed to pgf backend.
- PR #15216: Update docstrings of contains_point(s) methods
- PR #15209: Exclude s-g generated files from flake8 check.
- PR #15204: PEP8ify some variable names.

- PR #15196: Force html4 writer for sphinx 2
- PR #13544: Improve handling of subplots spanning multiple gridspec cells.
- PR #15194: Trivial style fixes.
- PR #15202: Deprecate the renderer parameter to Figure.tight_layout.
- PR #15195: Fix integers being passed as length to quiver3d.
- PR #15180: Add some more internal links to 3.2.0 what's new
- PR #13510: Change Locator MAXTICKS checking to emitting a log at WARNING level.
- PR #15184: Mark missing_references extension as parallel read safe
- PR #15150: Autodetect whether pgf can use includegraphics[interpolate].
- PR #15163: 3.2.0 API changes page
- PR #15176: What's new for 3.2.0
- PR #11947: Ensure streamplot Euler step is always called when going out of bounds.
- PR #13702: Deduplicate methods shared between Container and Artist.
- PR #15169: TST: verify warnings fail the test suite
- PR #14888: Replace some polar baseline images by check_figures_equal.
- PR #15027: More readability improvements on axis3d.
- PR #15171: Add useful error message when trying to add Slider to 3DAxes
- PR #13775: Doc: Scatter Hist example update
- PR #15164: removed a typo
- PR #15152: Support for shorthand hex colors.
- PR #15159: Follow up on #14424 for docstring
- PR #14424: ENH: Add argument size validation to quiver.
- PR #15137: DOC: add example to power limit API change note
- PR #15144: Improve local page contents CSS
- PR #15143: Restore doc references.
- PR #15124: Replace parameter lists with square brackets
- PR #13077: fix FreeType build on Azure
- PR #15123: Improve categorical example
- PR #15134: Fix missing references in doc build.
- PR #13937: Use PYTHONFAULTHANDLER to switch on the Python fault handler.

- PR #13452: Replace axis_artist.AttributeCopier by normal inheritance.
- PR #15045: Resize canvas when changing figure size
- PR #15122: Fixed app creation in qt5 backend (see #15100)
- PR #15099: Add lightsource parameter to bar3d
- PR #14876: Inline some afm parsing code.
- PR #15119: Deprecate a validator for a deprecated rcParam value.
- PR #15121: Fix Stacked bar graph example
- PR #15113: Cleanup layout_from_subplotspec.
- PR #13543: Remove zip_safe=False flag from setup.py.
- PR #12860: ENH: LogLocator: check for correct dimension of subs added
- PR #14349: Replace ValidateInterval by simpler specialized validators.
- PR #14352: Remove redundant is_landscape kwarg from backend_ps helpers.
- PR #15087: Pass gid to renderer
- PR #14703: Don't bother with manually resizing the Qt main window.
- PR #14833: Reuse TexManager implementation in convert_psfrags.
- PR #14893: Update layout.html for sphinx themes
- PR #15098: Simplify symlog range determination logic
- PR #15112: Cleanup legend() docstring.
- PR #15108: Fix doc build and resync matplotlibrc.template with actual defaults.
- PR #14940: Fix text kerning calculations and some FT2Font cleanup
- PR #15082: Privatize font_manager.JSONEncoder.
- PR #15106: Update docs of GridSpec
- PR #14832: ENH:made default tick formatter to switch to scientific notation earlier
- PR #15086: Style fixes.
- PR #15073: Add entry for blume to thirdparty package index
- PR #15095: Simplify _png extension by handling file open/close in Python.
- PR #15092: MNT: Add test for aitoff-projection
- PR #15101: Doc: fix typo in contour doc
- PR #14624: Fix axis inversion with loglocator and logitlocator.
- PR #15088: Fix more doc references.
- PR #15063: Add Comic Neue as a fantasy font.

- PR #14867: Propose change to PR merging policy.
- PR #15068: Add FontManager.addfont to register fonts at specific paths.
- PR #13397: Deprecate axes_grid1.colorbar (in favor of matplotlib's own).
- PR #14521: Move required_interactive_framework to canvas class.
- PR #15083: Cleanup spines example.
- PR #14997: Correctly set formatters and locators on removed shared axis
- PR #15064: Fix eps hatching in MacOS Preview
- PR #15074: Write all ACCEPTS markers in docstrings as comments.
- PR #15078: Clarify docstring of FT2Font.get_glyph_name.
- PR #15080: Fix cross-references in API changes < 3.0.0.
- PR #15072: Cleanup patheffects.
- PR #15071: Cleanup offsetbox.py.
- PR #15070: Fix cross-references in API changes < 2.0.0.
- PR #10691: Fix for shared axes diverging after setting tick markers
- PR #15069: Style fixes for font_manager.py.
- PR #15067: Fix cross-references in API changes < 1.0
- PR #15061: Fix cross-references in tutorials and FAQ
- PR #15060: Fix cross-references in examples.
- PR #14957: Documentation for using ConnectionPatch across Axes with constrained...
- PR #15053: Make citation bit of README less wordy
- PR #15044: numpydoc set_size_inches docstring
- PR #15050: Clarify unnecessary special handling for colons in paths.
- PR #14797: DOC: create a Agg figure without pyplot in buffer example
- PR #14844: Add citation info to README
- PR #14884: Do not allow canvas size to become smaller than MinSize in wx backend...
- PR #14941: Improvements to make_icons.py.
- PR #15048: DOC: more nitpick follow up
- PR #15043: Fix Docs: Don’t warn for unused ignores
- PR #15025: Re-write text wrapping logic
- PR #14840: Don't assume transform is valid on access to matrix.
- PR #14862: Make optional in docstrings optional

- PR #15028: Python version conf.py
- PR #15033: FIX: un-break nightly wheels on py37
- PR #15046: v3.1.x merge up
- PR #15015: Fix bad missing-references.json due to PR merge race condition.
- PR #14581: Make logscale bar/hist autolimits more consistents.
- PR #15034: Doc fix nitpick
- PR #14614: Deprecate {x,y,z}axis_date.
- PR #14991: Handle inherited is_separable, has_inverse in transform props detection.
- PR #15032: Clarify effect of axis('equal') on explicit data limits
- PR #15031: Update docs of GridSpec
- PR #14106: Describe FigureManager
- PR #15024: Update docs of GridSpecBase
- PR #14906: Deprecate some FT2Image methods.
- PR #14963: More Axis3D cleanup.
- PR #15009: Provide signatures to some C-level classes and methods.
- PR #14968: DOC: colormap manipulation tutorial update
- PR #15006: Deprecate get/set_*ticks minor positional use
- PR #14989: DOC:Update axes documentation
- PR #14871: Parametrize determinism tests.
- PR #14768: DOC: Enable nitpicky
- PR #15013: Matplotlib requires Python 3.6, which in turn requires Mac OS X 10.6+
- PR #15012: Fix typesetting of "GitHub"
- PR #14954: Cleanup polar_legend example.
- PR #14519: Check parameters of ColorbarBase
- PR #14942: Make _classic_test style a tiny patch on top of classic.
- PR #14988: pathlibify/fstringify setup/setupext.
- PR #14511: Deprecate allowing scalars for fill_between where
- PR #14493: Remove deprecated fig parameter form GridSpecBase.get_subplot_params()
- PR #14995: Further improve backend tutorial.

- PR #15000: Use warnings.warn, not logging.warning, in microseconds locator warning.
- PR #14990: Fix nonsensical transform in mixed-mode axes aspect computation.
- PR #15002: No need to access filesystem in test_dates.py.
- PR #14549: Improve backends documentation
- PR #14774: Fix image bbox clip.
- PR #14978: Typo fixes in pyplot.py
- PR #14702: Don't enlarge toolbar for Qt high-dpi.
- PR #14922: Autodetect some transform properties.
- PR #14962: Replace inspect.getfullargspec by inspect.signature.
- PR #14958: Improve docs of toplevel module.
- PR #14926: Save a matrix unpacking/repacking in offsetbox.
- PR #14961: Cleanup demo_agg_filter.
- PR #14924: Kill the C-level (private) RendererAgg.buffer_rgba, which returns a copy.
- PR #14946: Delete virtualenv faq.
- PR #14944: Shorten style.py.
- PR #14931: Deprecate some obscure rcParam synonyms.
- PR #14947: Fix inaccuracy re: backends in intro tutorial.
- PR #14904: Fix typo in secondary_axis.py example.
- PR #14925: Support passing spine bounds as single tuple.
- PR #14921: DOC: Make abbreviation of versus consistent.
- PR #14739: Improve indentation of Line2D properties in docstrings.
- PR #14923: In examples, prefer buffer_rgba to print_to_buffer.
- PR #14908: Make matplotlib.style.available sorted alphabetically.
- PR #13567: Deprecate MovieWriterRegistry cache-dirtyness system.
- PR #14879: Error out when unsupported kwargs are passed to Scale.
- PR #14512: Logit scale, changes in LogitLocator and LogitFormatter
- PR #12415: ENH: fig.set_size to allow non-inches units
- PR #13783: Deprecate disable_internet.
- PR #14886: Further simplify the flow of pdf text output.
- PR #14894: Make slowness warning for legend(loc="best") more accurate.
- PR #14891: Fix nightly test errors

- PR #14895: Fix typos
- PR #14890: Remove unused private helper method in mplot3d.
- PR #14872: Unify text layout paths.
- PR #8183: Allow array alpha for imshow
- PR #13832: Vectorize handling of stacked/cumulative in hist().
- PR #13630: Simplify PolarAxes.can_pan.
- PR #14565: Rewrite an argument check to _check_getitem
- PR #14875: Cleanup afm module docstring.
- PR #14880: Fix animation blitting for plots with shared axes
- PR #14870: FT2Font.get_char_index never returns None.
- PR #13463: Deprecate Locator.autoscale.
- PR #13724: ENH: anti-alias down-sampled images
- PR #14848: Clearer error message for plt.axis()
- PR #14660: colorbar(label=None) should give an empty label
- PR #14654: Cleanup of docstrings of scales
- PR #14868: Update bar stacked example to directly manipulate axes.
- PR #14749: Fix get_canvas_width_height() for pgf backend.
- PR #14776: Make ExecutableUnavailableError
- PR #14843: Don't try to cleanup CallbackRegistry during interpreter shutdown.
- PR #14849: Improve tkagg icon resolution
- PR #14866: changed all readme headings to verbs
- PR #13364: Numpyfy tick handling code in Axis3D.
- PR #13642: FIX: get_datalim for collection
- PR #14860: Stopgap fix for pandas converters in tests.
- PR #6498: Check canvas identity in Artist.contains.
- PR #14707: Add titlecolor in rcParams
- PR #14853: Fix typo in set_adjustable check.
- PR #14845: More cleanups.
- PR #14809: Clearer calls to ConnectionPatch.
- PR #14716: Use str instead of string as type in docstrings
- PR #14338: Simplify/pathlibify image_comparison.
- PR #8930: timedelta formatter

- PR #14733: Deprecate FigureFrameWx.statusbar & NavigationToolbar2Wx.statbar.
- PR #14713: Unite masked and NaN plot examples
- PR #14576: Let Axes3D share have_units, _on_units_changed with 2d axes.
- PR #14575: Make ticklabel_format work both for 2D and 3D axes.
- PR #14834: DOC: Webpage not formatted correctly on gallery docs
- PR #14730: Factor out common parts of wx event handlers.
- PR #14727: Fix axes aspect for non-linear, non-log, possibly mixed-scale axes.
- PR #14835: Only allow set_adjustable("datalim") for axes with standard data ratios.
- PR #14746: Simplify Arrow constructor.
- PR #14752: Doc changes to git setup
- PR #14732: Deduplicate wx configure_subplots tool.
- PR #14715: Use array-like in docs
- PR #14728: More floating_axes cleanup.
- PR #14719: Make Qt navtoolbar more robust against removal of either pan or zoom.
- PR #14695: Various small simplifications
- PR #14745: Replace Affine2D().scale(x, x) by Affine2D().scale(x).
- PR #14687: Add missing spaces after commas in docs
- PR #14810: Lighten icons of NavigationToolbar2QT on dark-themes
- PR #14786: Deprecate axis_artist.BezierPath.
- PR #14750: Misc. simplifications.
- PR #14807: API change note on automatic blitting detection for backends
- PR #11004: Deprecate smart_bounds handling in Axis and Spine
- PR #14785: Kill some never-used attributes.
- PR #14723: Cleanup some parameter descriptions in matplotlibrc.template
- PR #14808: Small docstring updates
- PR #14686: Inset orientation
- PR #14805: Simplify text_layout example.
- PR #12052: Make AxesImage.contains account for transforms
- PR #11860: Let MovieFileWriter save temp files in a new dir
- PR #11423: FigureCanvas Designer

- PR #10688: Add legend handler and artist for FancyArrow
- PR #8321: Added ContourSet clip_path kwarg and set_clip_path() method (#2369)
- PR #14641: Simplify _process_plot_var_args.
- PR #14631: Refactor from_levels_and_colors.
- PR #14790: DOC:Add link to style examples in matplotlib.style documentation
- PR #14799: Deprecate dates.mx2num.
- PR #14793: Remove sudo tag in travis
- PR #14795: Autodetect whether a canvas class supports blitting.
- PR #14794: DOC: Update the documentation of homepage of website
- PR #14629: Delete HTML build sources to save on artefact upload time
- PR #14792: Fix spelling typos
- PR #14789: Prefer Affine2D.translate to offset_transform in examples.
- PR #14783: Cleanup mlab.detrend.
- PR #14791: Make 'extended' and 'expanded' synonymous in font_manager
- PR #14787: Remove axis_artist _update, which is always a noop.
- PR #14758: Compiling C-ext with incorrect FreeType libs makes future compiles break
- PR #14763: Deprecate math_symbol_table function directive
- PR #14762: Decrease uses of get_canvas_width_height.
- PR #14748: Cleanup demo_text_path.
- PR #14740: Remove sudo tag in travis
- PR #14737: Cleanup twin axes docstrings.
- PR #14729: Small simplifications.
- PR #14726: Trivial simplification to Axis3d._get_coord_info.
- PR #14718: Add explanations for single character color names.
- PR #14710: Pin pydocstyle<4.0
- PR #14709: Try to improve the readability and styling of matplotlibrc.template file
- PR #14278: Inset axes bug and docs fix
- PR #14478: MNT: protect from out-of-bounds data access at the c level
- PR #14569: More deduplication of backend_tools.
- PR #14652: Soft-deprecate transform_point.

- PR #14664: Improve error reporting for scatter c as invalid RGBA.
- PR #14625: Don't double-wrap in silent_list.
- PR #14689: Update embedding_in_wx4 example.
- PR #14679: Further simplify colormap reversal.
- PR #14667: Move most of pytest's conf to conftest.py.
- PR #14632: Remove reference to old Tk/Windows bug.
- PR #14673: More shortening of setup.py prints.
- PR #14678: Fix small typo
- PR #14680: Format parameters in descriptions with emph instead of backticks
- PR #14674: Simplify colormap reversal.
- PR #14672: Artist tutorial fixes
- PR #14653: Remove some unnecessary prints from setup.py.
- PR #14662: Add a _check_getitem helper to go with _check_in_list/_check_isinstance.
- PR #14666: Update IPython's doc link in Image tutorial
- PR #14671: Improve readability of matplotlibrc.template
- PR #14665: Fix a typo in pyplot tutorial
- PR #14616: Use builtin round instead of np.round for scalars.
- PR #12554: backend_template docs and fixes
- PR #14635: Fix bug when setting negative limits and using log scale
- PR #14604: Update hist() docstring following removal of normed kwarg.
- PR #14630: Remove the private Tick._name attribute.
- PR #14555: Coding guidelines concerning the API
- PR #14516: Document and test _get_packed_offsets()
- PR #14628: matplotlib > Matplotlib in devel docs
- PR #14627: gitignore pip-wheel-metadta/ directory
- PR #14612: Update some mplot3d docs.
- PR #14617: Remove a Py2.4(!) backcompat fix.
- PR #14605: Update hist2d() docstring.
- PR #13084: When linking against libpng/zlib on Windows, use upstream lib names.
- PR #13685: Remove What's new fancy example
- PR #14573: Cleanup jpl_units.

- PR #14583: Fix overly long lines in setupext.
- PR #14588: Remove [status] suppress from setup.cfg.
- PR #14591: Style fixes for secondary_axis.
- PR #14594: DOC: Make temperature scale example use a closure for easier reusability
- PR #14447: FIX: allow secondary axes minor locators to be set
- PR #14567: Fix unicode_minus + usetex.
- PR #14351: Remove some redundant check_in_list calls.
- PR #14550: Restore thumbnail of usage guide
- PR #10222: Use symlinks instead of copies for test result_images.
- PR #14267: cbook docs cleanup
- PR #14556: Improve @deprecated's docstring.
- PR #14557: Clarify how to work with threads.
- PR #14545: In contributing.rst, encourage kwonly args and minimizing public APIs.
- PR #14533: Misc. style fixes.
- PR #14542: Move plot_directive doc to main API index.
- PR #14499: Improve custom figure example
- PR #14543: Remove the "Developing a new backend" section from contributing guide.
- PR #14540: Simplify backend switching in plot_directive.
- PR #14539: Don't overindent enumerated list in plot_directive docstring.
- PR #14537: Slightly tighten the Bbox API.
- PR #14223: Rewrite intro to usage guide.
- PR #14495: Numpydocify axes_artist.py
- PR #14529: mpl_toolkits style fixes.
- PR #14528: mathtext style fixes.
- PR #13536: Make unit converters also handle instances of subclasses.
- PR #13730: Include FreeType error codes in FreeType exception messages.
- PR #14500: Fix pydocstyle D403 (First word of the first line should be properly capitalized) in examples
- PR #14506: Simplify Qt tests.
- PR #14513: More fixes to pydocstyle D403 (First word capitalization)

- PR #14496: Fix pydocstyle D208 (Docstring is over-indented)
- PR #14347: Deprecate rcsetup.validate_path_exists.
- PR #14383: Remove the ``package_data.dlls`` setup.cfg entry.
- PR #14346: Simplify various validators in rcsetup.
- PR #14366: Move test_rcparams test files inline into test_rcparams.py.
- PR #14401: Assume that mpl-data is in its standard location.
- PR #14454: Simplify implementation of svg.image_inline.
- PR #14470: Add _check_isinstance helper.
- PR #14479: fstringify backend_ps more.
- PR #14484: Support unicode minus with ps.useafm.
- PR #14494: Style fixes.
- PR #14465: Docstrings cleanups.
- PR #14466: Let SecondaryAxis inherit get_tightbbox from _AxesBase.
- PR #13940: Some more f-strings.
- PR #14379: Remove unnecessary uses of unittest.mock.
- PR #14483: Improve font weight guessing.
- PR #14419: Fix test_imshow_pil on Windows.
- PR #14460: canvas.blit() already defaults to blitting the full figure canvas.
- PR #14462: Register timeout pytest marker.
- PR #14414: FEATURE: Alpha channel in Gouraud triangles in the pdf backend
- PR #13659: Clarify behavior of the 'tight' kwarg to autoscale/autoscale_view.
- PR #13901: Only test png output for mplot3d.
- PR #13338: Replace list.extend by star-expansion or other constructs.
- PR #14448: Misc doc style cleanup
- PR #14310: Update to Bounding Box for Qt5 FigureCanvasATAgg.paintEvent()
- PR #14380: Inline $MPLLOCALFREETYPE/$PYTEST_ADDOPTS/$NPROC in .travis.yml.
- PR #14413: MAINT: small improvements to the pdf backend
- PR #14452: MAINT: Minor cleanup to make functions more self consisntent
- PR #14441: Misc. docstring cleanups.
- PR #14440: Interpolations example
- PR #14402: Prefer `mpl.get_data_path()`, and support Paths in FontProperties.

- PR #14420: MAINT: Upgrade pytest again
- PR #14423: Fix docstring of subplots().
- PR #14410: Use aspect=1, not aspect=True.
- PR #14412: MAINT: Don't install pytest 4.6.0 on Travis
- PR #14377: Rewrite assert np.* tests to use numpy.testing
- PR #14399: Improve warning for case where data kwarg entry is ambiguous.
- PR #14390: Cleanup docs of bezier
- PR #14400: Fix to_rgba_array() for empty input
- PR #14308: Small clean to SymmetricalLogLocator
- PR #14311: travis: add c code coverage measurements
- PR #14393: Remove remaining unicode-strings markers.
- PR #14391: Remove explicit inheritance from object
- PR #14343: acquiring and releaseing keypresslock when textbox is being activated
- PR #14353: Register flaky pytest marker.
- PR #14373: Properly hide __has_include to support C++<17 compilers.
- PR #14378: Remove setup_method
- PR #14368: Finish removing jquery from the repo.
- PR #14360: Deprecate `boxplot(..., whis="range")`.
- PR #14376: Simplify removal of figure patch from bbox calculations.
- PR #14363: Make is_natively_supported private.
- PR #14330: Remove remaining unittest.TestCase uses
- PR #13663: Kill the PkgConfig singleton in setupext.
- PR #13067: Simplify generation of error messages for missing libpng/freetype.
- PR #14358: DOC boxplot `whis` parameter
- PR #14014: Disallow figure argument for pyplot.subplot() and Figure.add_subplot()
- PR #14350: Use cbook._check_in_list more often.
- PR #14348: Cleanup markers.py.
- PR #14345: Use importorskip for tests depending on pytz.
- PR #14170: In setup.py, inline the packages that need to be installed into setup().
- PR #14332: Use raw docstrings instead of escaping backslashes

- PR #14336: Enforce pydocstyle D412
- PR #14144: Deprecate the 'warn' parameter to matplotlib.use().
- PR #14328: Remove explicit inheritance from object
- PR #14035: Improve properties formatting in interpolated docstrings.
- PR #14018: pep8ing.
- PR #13542: Move {setup,install}_requires from setupext.py to setup.py.
- PR #13670: Simplify the logic of axis().
- PR #14046: Deprecate checkdep_ps_distiller.
- PR #14236: Simplify StixFonts.get_sized_alternatives_for_symbol.
- PR #14101: Shorten _ImageBase._make_image.
- PR #14246: Deprecate public use of makeMappingArray
- PR #13740: Deprecate plotfile.
- PR #14216: Walk the artist tree when preparing for saving with tight bbox.
- PR #14305: Small grammatical error.
- PR #14104: Factor out retrieval of data relative to datapath
- PR #14016: pep8ify backends.
- PR #14299: Fix #13711 by importing cbook.
- PR #14244: Remove APIs deprecated in mpl3.0.
- PR #14068: Alternative fix for passing iterator as frames to FuncAnimation
- PR #13711: Deprecate NavigationToolbar2Tk.set_active.
- PR #14280: Simplify validate_markevery logic.
- PR #14273: pep8ify a couple of variable names.
- PR #14115: Reorganize scatter arguments parsing.
- PR #14271: Replace some uses of np.iterable
- PR #14257: Changing cmap(np.nan) to 'bad' value rather than 'under' value
- PR #14259: Deprecate string as color sequence
- PR #13506: Change colorbar for contour to have the proper axes limits...
- PR #13494: Add colorbar annotation example plot to gallery
- PR #14266: Make matplotlib.figure.AxesStack private
- PR #14166: Shorten usage of `@image_comparison`.
- PR #14240: Merge up 31x
- PR #14242: Avoid a buffer copy in PillowWriter.

- PR #9672: Only set the wait cursor if the last draw was >1s ago.
- PR #14224: Update plt.show() doc
- PR #14218: Use stdlib mimetypes instead of hardcoding them.
- PR #14082: In tk backend, don't try to update mouse position after resize.
- PR #14084: Check number of positional arguments passed to quiver()
- PR #14214: Fix some docstring style issues.
- PR #14201: Fix E124 flake8 violations (closing bracket indentation).
- PR #14096: Consistently use axs to refer to a set of Axes
- PR #14204: Fix various flake8 indent problems.
- PR #14205: Obey flake8 "don't assign a lambda, use a def".
- PR #14198: Remove unused imports
- PR #14173: Prepare to change the default pad for AxesDivider.append_axes.
- PR #13738: Fix TypeError when plotting stacked bar chart with decimal
- PR #14151: Clarify error with usetex when cm-super is not installed.
- PR #14107: Feature: draw percentiles in violinplot
- PR #14172: Remove check_requirements from setupext.
- PR #14158: Fix test_lazy_imports in presence of $MPLBACKEND or matplotlibrc.
- PR #14157: Isolate nbagg test from user ipython profile.
- PR #14147: Dedent overindented list in example docstring.
- PR #14134: Deprecate the dryrun parameter to print_foo().
- PR #14145: Remove warnings handling for fixed bugs.
- PR #13977: Always import pyplot when calling matplotlib.use().
- PR #14131: Make test suite fail on warnings.
- PR #13593: Only autoscale_view() when needed, not after every plotting call.
- PR #13902: Add support for metadata= and pil_kwargs= in imsave().
- PR #14140: Avoid backslash-quote by changing surrounding quotes.
- PR #14132: Move some toplevel strings into the only functions that use them.
- PR #13708: Annotation.contains shouldn't consider the text+arrow's joint bbox.
- PR #13980: Don't let margins expand polar plots to negative radii by default.
- PR #14075: Remove uninformative entries from glossary.
- PR #14002: Allow pandas DataFrames through norms

- PR #14114: Allow SVG Text-as-Text to Use Data Coordinates
- PR #14120: Remove mention of $QT_API in matplotlibrc example.
- PR #13878: Style fixes for floating_axes.
- PR #14108: Deprecate FigureCanvasMac.invalidate in favor of draw_idle.
- PR #13879: Clarify handling of "extreme" values in FloatingAxisArtistHelper.
- PR #5602: Automatic downsampling of images.
- PR #14112: Remove old code path in layout.html
- PR #13959: Scatter: make "c" and "s" argument handling more consistent.
- PR #14110: Simplify scatter_piecharts example.
- PR #14111: Trivial cleanups.
- PR #14085: Simplify get_current_fig_manager().
- PR #14083: Deprecate FigureCanvasBase.draw_cursor.
- PR #14089: Cleanup bar_stacked, bar_unit_demo examples.
- PR #14063: Add pydocstyle checks to flake8
- PR #14077: Fix tick label wobbling in animated Qt example
- PR #14070: Cleanup some pyplot docstrings.
- PR #6280: Added ability to offset errorbars when using errorevery.
- PR #13679: Fix passing iterator as frames to FuncAnimation
- PR #14023: Improve Unicode minus example
- PR #14041: Pretty-format subprocess logs.
- PR #14038: Cleanup path.py docstrings.
- PR #13701: Small cleanups.
- PR #14020: Better error message when trying to use Gtk3Agg backend without cairo
- PR #14021: Fix ax.legend Returns markup
- PR #13986: Support RGBA for quadmesh mode of pcolorfast.
- PR #14009: Deprecate compare_versions.
- PR #14010: Deprecate get_home()
- PR #13932: Remove many unused variables.
- PR #13854: Cleanup contour.py.
- PR #13866: Switch PyArg_ParseTupleAndKeywords from "es" to "s".
- PR #13945: Make unicode_minus example more focused.

- PR #13876: Deprecate factor=None in axisartist.
- PR #13929: Better handle deprecated rcParams.
- PR #13851: Deprecate setting Axis.major.locator to non-Locator; idem for Formatters
- PR #13938: numpydocify quiverkey.
- PR #13936: Pathlibify animation.
- PR #13984: Allow setting tick colour on 3D axes
- PR #13987: Deprecate mlab.{apply_window,stride_repeat}.
- PR #13983: Fix locator/formatter setting when removing shared Axes
- PR #13957: Remove many unused variables in tests.
- PR #13981: Test cleanups.
- PR #13970: Check vmin/vmax are valid when doing inverse in LogNorm
- PR #13978: Make normalize_kwargs more convenient for third-party use.
- PR #13972: Remove _process_plot_var_args.set{line,patch}_props.
- PR #13795: Make _warn_external correctly report warnings arising from tests.
- PR #13885: Deprecate axisartist.grid_finder.GridFinderBase.
- PR #13913: Fix string numbers in to_rgba() and is_color_like()
- PR #13935: Deprecate the useless switch_backend_warn parameter to matplotlib.test.
- PR #13952: Cleanup animation tests.
- PR #13942: Make Cursors an (Int)Enum.
- PR #13953: Unxfail a now fixed test in test_category.
- PR #13925: Fix passing Path to ps backend when text.usetex rc is True.
- PR #13943: Don't crash on str(figimage(...)).
- PR #13944: Document how to support unicode minus in pgf backend.
- PR #13802: New rcparam to set default axes title location
- PR #13855: `a and b or c -> b if a else c`
- PR #13923: Correctly handle invalid PNG metadata.
- PR #13926: Suppress warnings in tests.
- PR #13920: Style fixes for category.py.
- PR #13889: Shorten docstrings by removing unneeded :class:/:func: + rewordings.
- PR #13911: Fix joinstyles example

- PR #13917: Faster categorical tick formatter.
- PR #13918: Make matplotlib.testing assume pytest by default, not nose.
- PR #13894: Check for positive number of rows and cols
- PR #13895: Remove unused setupext.is_min_version.
- PR #13886: Shorten Figure.set_size_inches.
- PR #13859: Ensure figsize is positive finite
- PR #13877: `zeros_like(x) + y` -> `full_like(x, y)`
- PR #13875: Style fixes for grid_helper_curvelinear.
- PR #13873: Style fixes to grid_finder.
- PR #13782: Don't access internet during tests.
- PR #13833: Some more usage of _check_in_list.
- PR #13834: Cleanup FancyArrowPatch docstring
- PR #13811: Generate Figure method wrappers via boilerplate.py
- PR #13797: Move sphinxext test to matplotlib.tests like everyone else.
- PR #13770: broken_barh docstring
- PR #13757: Remove mention of "enabling fontconfig support".
- PR #13454: Add "c" as alias for "color" for Collections
- PR #13756: Reorder the logic of _update_title_position.
- PR #13744: Restructure boilerplate.py
- PR #13369: Use default colours for examples
- PR #13697: Delete pyplot_scales example.
- PR #13726: Clarify a bit the implementation of blend_hsv.
- PR #13731: Check for already running QApplication in Qt embedding example.
- PR #13736: Deduplicate docstrings and validation for set_alpha.
- PR #13737: Remove duplicated methods in FixedAxisArtistHelper.
- PR #13721: Kill pyplot docstrings that get overwritten by @docstring.copy.
- PR #13690: Cleanup hexbin.
- PR #13683: Remove axes border for examples that list styles
- PR #13280: Add SubplotSpec.add_subplot.
- PR #11387: Deprecate Axes3D.w_{x,y,z}axis in favor of .{x,y,z}axis.
- PR #13671: Suppress some warnings in tests.
- PR #13657: DOC: fail the doc build on errors, but keep going to end

- PR #13647: Fix FancyArrowPatch joinstyle
- PR #13637: BLD: parameterize python_requires
- PR #13633: plot_directive: Avoid warning if plot_formats doesn't contain 'png'
- PR #13629: Small example simplification.
- PR #13620: Improve watermark example
- PR #13589: Kill Axes._connected.
- PR #13428: free cart pendulum animation example
- PR #10487: fixed transparency bug
- PR #13551: Fix IndexError for pyplot.legend() when plotting empty bar chart with label
- PR #13524: Cleanup docs for GraphicsContextBase.{get,set}_dashes.
- PR #13556: Cleanup warnings handling in tests.
- PR #8100: Deprecate MAXTICKS, Locator.raise_if_exceeds.
- PR #13534: More followup to autoregistering 3d axes.
- PR #13327: pcolorfast simplifications.
- PR #13532: More use of cbook._check_in_list.
- PR #13520: Register 3d projection by default.
- PR #13394: Deduplicate some code between floating_axes and grid_helper_curvelinear.
- PR #13527: Make SubplotSpec.num2 never None.
- PR #12249: Replaced noqa-comments by using Axes3D.name instead of '3d' for proje…

Issues (125):

- #16487: Add link to blog to front page
- #16478: The bottom parameter of plt.hist() shifts the data as well, not just the baseline
- #16280: SymLogNorm colorbar incorrect on master
- #16448: Bad interaction between shared axes and pcolormesh sticky edges
- #16451: InvertedLogTransform inherits from deprecated base
- #16420: Error when adding colorbar to pcolormesh of a boolean array
- #16114: Prose error on website (first paragraph)
- #8291: Unable to pickle.load(fig) with mpl in jupyter notebook
- #16173: Constrained_layout creates extra axes when used with subgridspec

- #16127: nbformat 5.0.0 missing schema files
- #15849: Using pandas.Timestamp in blended coordinate system of ax.annotate.
- #6015: scatterplot axis autoscale fails for small data values
- #15806: 3.2.0 may break some Cartopy tests
- #15852: Lasso selector does not show in Jupyter notebook
- #15820: Show incomplete tick labels when using mixed chinese and english characters
- #15770: DOCS 2D Line label option `_nolegend_` is not documented
- #15332: Type promotion error with datetime bins in hist
- #15611: BUG: Qt5Agg window size regression
- #7130: Incorrect autoscaling of polar plot limits after scatter
- #15576: Multi-line ticks cause cut-offs
- #8609: Clipped tick labels
- #15517: antialiased image check seems wrong when used on zoomed image
- #13400: Qt Embedding w/ Spyder
- #14724: drawstyle parameter of line needs example
- #13619: Importing matplotlib.animation prevents python script from executing in the background
- #14270: Secondary axis called with [0, 1] might produce exceptions in case these are invalid data
- #15417: Why is smart_bounds() being deprecated?
- #9778: Blanks in colorbar just inside of 'extend' arrowpoints when using AxesGrid
- #15336: DivergingNorm is a misleading name
- #15399: OSError: [Errno 86] Bad CPU type in executable: 'convert' on import matplotlib.animation
- #15109: matplotlib.collections inheritance diagram small/blurry
- #15331: Log Scale: FloatingPointError: underflow encountered in power
- #15251: Large memory growth with log scaling and linear ticking
- #15247: Colorbar tick placement issues with ImageGrid and LogNorm
- #15306: Footer off centre
- #13485: Matplotlib NavigationToolbar2Tk disappears when reducing window size

- #15232: DOC: Automatic default rcParam expansion creates misleading sentences
- #14141: setting spine position on a log plot fails
- #15138: Make plt.style.use accept path-like objects in addition to string
- #14207: Check if point is in path or not by contains_point
- #13591: Style issues when building the docs with (future) Sphinx 2.0
- #8089: Using Minute Locator to set x-axis ticks exceeds Locator.MAXTICKS
- #15075: sphinxext.missing_references does not specify if it supports parallel file read.
- #10963: Replace pgfimage by includegraphics in PGF backend
- #15156: ax.text fails with positional argument error
- #14439: hist() fails when all data points are np.nan
- #15042: How to handle sphinx nitpicky mode
- #14060: quiver(C=...) argument is not reasonably validated
- #11335: TST: testing not catching bad escape sequences in doc strings
- #15040: Wrong figure window size after calling fig.set_size_inches() repeatedly
- #15100: Issue with creating QApplication in QT backend
- #14887: kerning seems generally wrong
- #14800: default tick formatter could switch to scientific notation earlier
- #14503: Add a test for #14451
- #14907: ConnectionPatch across axes needs to be excluded from layout management
- #14911: Removing a shared axes via `ax.remove()` leads to an error.
- #12462: cbar.add_lines should allow manually adding lines, not just contour sets
- #14796: Show user how to use Agg buffer in example
- #14883: MinSize not respected using wx backend causes wxAssertionError. Bug fix included.
- #15014: Wrapping of text adds leading newline character if first word is long
- #14918: constrained_layout fails with hidden axis...
- #14981: Barplot call crashes when called with yscale="log" and bins with h=0
- #4621: Default bottom of Stepfilled histograms should be set according to ymin
- #15030: Doc build broken
- #8093: set_ylim not working with plt.axis('equal')

- #6055: Serious problems on the axes documentation
- #9979: Axis limits are set badly with small values in scatter().
- #10842: Text bbox empty dict should be ignored
- #13698: The default logit minor locator should not display tick labels
- #14878: plt.yscale doesn't throw warning with invalid kwarg
- #5619: Symlog linear region
- #14564: Broken string interpolation
- #13668: Add better error message to plt.axis()
- #14563: colorbar label prints "None" when label=None
- #13660: Closing a matplotlib figure with event handling occasionally causes “TypeError: isinstance()”
- #13033: 'NoneType' has no attribute '_alive' when using plt in a context manager
- #13891: Blurry app icon on macOS
- #14656: Axes title default color
- #14831: DOC: Webpage not formatted correctly on gallery docs
- #13819: Aspect ratio for not so common scales
- #8878: Setting aspect ratio for semi-log plots
- #4900: UnboundLocalError: local variable 'aspect_scale_mode' referenced before assignment
- #14608: Issue with using plt.axis('equal') with plt.polar(theta,r) plot
- #12893: [PyQt] NavigationToolbar2QT : Error when removing tools
- #14670: indicate_inset rectangles is sensitive to axis-flipping
- #14362: Add link to style examples in matplotlib.style documentation
- #6295: restore_region is not documented as a method of FigureCanvas
- #14754: Better pointer to dev docs on website
- #14744: Savefig svg fails with "Cannot cast array data from dtype('<U7') to dtype('float64') according to the rule 'safe'"
- #11919: Wrong Error Message
- #6824: Image comparison decorator: symlinks to baseline images
- #12180: Deprecate and remove pyplot.plotfile?
- #14180: ImageComparisonFailure: Image sizes do not match expected size
- #14443: Secondary axis does not show minor ticks.

- #8423: UnicodeDecodeError when making a plot using the 'classic' style and text.usetex=True
- #11275: A "TypeError" is raised if subclass inherited from "datetime" is used
- #9127: ps.useafm and axes.unicode_minus are incompatible
- #7571: matplotlib.widget.TextBox not correctly stopping keyboard shortcuts
- #14370: gcc error when building matplotlib dev from source
- #14011: TypeError on plt.subplot(figure=plt.figure())
- #13676: FuncAnimation with generator causes crash on StopIteration
- #9892: colormaps (cm) do not properly handle NaN values.
- #14122: Unexpected behavior in matplotlib.colors.to_rgba_array when passing unknown color name string
- #9546: The busy cursor is annoying in some instances
- #10788: TypeError when plotting stacked bar chart with decimal
- #14146: Saving polar plots with MiKTeX on Windows fails for some file formats
- #8532: Feature Request: draw percentiles in violinplot
- #13883: In headless mode, matplotlib.use('tkagg') only errors after importing pyplot
- #13967: Creating colorbar without artist fails with LogNorm
- #12542: The plot function of the matplotlib 2 and 3 versions is much slower than 1.5.3
- #13292: Non-sensical negative radial scale minimum autoset in polar plot
- #10909: Calling a Normalize instance with a DataFrame
- #14076: Tick label positions wobble in animated Qt example
- #14007: GTK3Agg backend raises ImportError for missing cairo dependency
- #12911: Tick mark color cannot be set on Axes3D
- #12853: Remove()ing a shared axes prevents the remaining axes from using unit-provided formatters
- #13912: `is_color_like` returning erroneous value on strings of integers
- #13921: − with save fig in .pgf
- #13872: ValueError message requests impossible condition
- #13857: Zero-width figure crashes libpng
- #13768: broken_barh docstring incorrect information
- #13641: `joinstyle` is not respected for `FancyArrowPatch` (either the path or the arrow heads)

- #11923: ColorbarBase fails to show if the first two values map to the same result
- #11527: Inconsistent path intersection
- #13003: IndexError thrown by pyplot.legend() when plotting empty bar chart with label

7.1.7 GitHub Stats for Matplotlib 3.1.2

GitHub stats for 2019/05/18 - 2019/06/30 (tag: v3.1.0)

These lists are automatically generated, and may be incomplete or contain duplicates.

We closed 30 issues and merged 120 pulnl requests. The full list can be seen on GitHub

The following 30 authors contributed 323 commits.

- Adam Gomaa
- Antony Lee
- Ben Root
- Christer Jensen
- chuanzhu xu
- David Stansby
- Deng Tian
- djdt
- Dora Fraeman Caswell
- Elan Ernest
- Elliott Sales de Andrade
- Eric Firing
- Filipe Fernandes
- Ian Thomas
- ImportanceOfBeingErnest
- Jody Klymak
- Johannes H. Jensen
- Jonas Camillus Jeppesen
- LeiSurrre
- Matt Adamson
- MeeseeksMachine

- Molly Rossow
- Nathan Goldbaum
- Nelle Varoquaux
- Paul Ivanov
- RoryIAngus
- Ryan May
- Thomas A Caswell
- Thomas Robitaille
- Tim Hoffmann

GitHub issues and pull requests:

Pull Requests (120):

- PR #14636: Don't capture stderr in _check_and_log_subprocess.
- PR #14655: Backport PR #14649 on branch v3.1.x (Fix appveyor conda py37)
- PR #14649: Fix appveyor conda py37
- PR #14646: Backport PR #14640 on branch v3.1.x (FIX: allow secondary axes to be non-linear)
- PR #14640: FIX: allow secondary axes to be non-linear
- PR #14643: Second attempt at fixing axis inversion (for mpl3.1).
- PR #14623: Fix axis inversion with loglocator and logitlocator.
- PR #14619: Backport PR #14598 on branch v3.1.x (Fix inversion of shared axes.)
- PR #14621: Backport PR #14613 on branch v3.1.x (Cleanup DateFormatter docstring.)
- PR #14622: Backport PR #14611 on branch v3.1.x (Update some axis docstrings.)
- PR #14611: Update some axis docstrings.
- PR #14613: Cleanup DateFormatter docstring.
- PR #14598: Fix inversion of shared axes.
- PR #14610: Backport PR #14579 on branch v3.1.x (Fix inversion of 3d axis.)
- PR #14579: Fix inversion of 3d axis.
- PR #14600: Backport PR #14599 on branch v3.1.x (DOC: Add numpngw to third party packages.)
- PR #14574: Backport PR #14568 on branch v3.1.x (Don't assume tk canvas have a manager attached.)

- PR #14568: Don't assume tk canvas have a manager attached.
- PR #14571: Backport PR #14566 on branch v3.1.x (Move setting of AA_EnableHighDpiScaling before creating QApplication.)
- PR #14566: Move setting of AA_EnableHighDpiScaling before creating QApplication.
- PR #14541: Backport PR #14535 on branch v3.1.x (Invalidate FT2Font cache when fork()ing.)
- PR #14535: Invalidate FT2Font cache when fork()ing.
- PR #14522: Backport PR #14040 on branch v3.1.x (Gracefully handle non-finite z in tricontour (issue #10167))
- PR #14434: Backport PR #14296 on branch v3.1.x (Fix barbs to accept array of bool for `flip_barb`)
- PR #14518: Backport PR #14509 on branch v3.1.x (Fix too large icon spacing in Qt5 on non-HiDPI screens)
- PR #14509: Fix too large icon spacing in Qt5 on non-HiDPI screens
- PR #14514: Backport PR #14256 on branch v3.1.x (Improve docstring of Axes.barbs)
- PR #14256: Improve docstring of Axes.barbs
- PR #14505: Backport PR #14395 on branch v3.1.x (MAINT: work around non-zero exit status of "pdftops -v" command.)
- PR #14504: Backport PR #14445 on branch v3.1.x (FIX: fastpath clipped artists)
- PR #14502: Backport PR #14451 on branch v3.1.x (FIX: return points rather than path to fix regression)
- PR #14445: FIX: fastpath clipped artists
- PR #14497: Backport PR #14491 on branch v3.1.x (Fix uses of PyObject_IsTrue.)
- PR #14491: Fix uses of PyObject_IsTrue.
- PR #14492: Backport PR #14490 on branch v3.1.x (Fix links of parameter types)
- PR #14490: Fix links of parameter types
- PR #14489: Backport PR #14459 on branch v3.1.x (Cleanup docstring of DraggableBase.)
- PR #14459: Cleanup docstring of DraggableBase.
- PR #14485: Backport #14429 on v3.1.x
- PR #14486: Backport #14403 on v3.1.
- PR #14429: FIX: if the first elements of an array are masked keep checking

- PR #14481: Backport PR #14475 on branch v3.1.x (change ginoput docstring to match behavior)
- PR #14482: Backport PR #14464 on branch v3.1.x (Mention origin and extent tutorial in API docs for origin kwarg)
- PR #14464: Mention origin and extent tutorial in API docs for origin kwarg
- PR #14468: Backport PR #14449: Improve docs on gridspec
- PR #14475: change ginoput docstring to match behavior
- PR #14477: Backport PR #14461 on branch v3.1.x (Fix out of bounds read in backend_tk.)
- PR #14476: Backport PR #14474 on branch v3.1.x (Fix default value in docstring of errorbar func)
- PR #14461: Fix out of bounds read in backend_tk.
- PR #14474: Fix default value in docstring of errorbar func
- PR #14473: Backport PR #14472 on branch v3.1.x (Fix NameError in example code for setting label via method)
- PR #14472: Fix NameError in example code for setting label via method
- PR #14449: Improve docs on gridspec
- PR #14450: Backport PR #14422 on branch v3.1.x (Fix ReST note in span selector example)
- PR #14446: Backport PR #14438 on branch v3.1.x (Issue #14372 - Add degrees to documentation)
- PR #14438: Issue #14372 - Add degrees to documentation
- PR #14437: Backport PR #14387 on branch v3.1.x (Fix clearing rubberband on nbagg)
- PR #14387: Fix clearing rubberband on nbagg
- PR #14435: Backport PR #14425 on branch v3.1.x (Lic restore license paint)
- PR #14296: Fix barbs to accept array of bool for `flip_barb`
- PR #14430: Backport PR #14397 on branch v3.1.x (Correctly set clip_path on pcolorfast return artist.)
- PR #14397: Correctly set clip_path on pcolorfast return artist.
- PR #14409: Backport PR #14335 on branch v3.1.x (Add explanation of animation.embed_limit to matplotlibrc.template)
- PR #14335: Add explanation of animation.embed_limit to matplotlibrc.template
- PR #14403: Revert "Preserve whitespace in svg output."
- PR #14407: Backport PR #14406 on branch v3.1.x (Remove extra iint in math_symbol_table for document)

- PR #14398: Backport PR #14394 on branch v3.1.x (Update link to "MathML torture test".)
- PR #14394: Update link to "MathML torture test".
- PR #14389: Backport PR #14388 on branch v3.1.x (Fixed one little spelling error)
- PR #14385: Backport PR #14316 on branch v3.1.x (Improve error message for kiwisolver import error (DLL load failed))
- PR #14388: Fixed one little spelling error
- PR #14384: Backport PR #14369 on branch v3.1.x (Don't use deprecated mathcircled in docs.)
- PR #14316: Improve error message for kiwisolver import error (DLL load failed)
- PR #14369: Don't use deprecated mathcircled in docs.
- PR #14375: Backport PR #14374 on branch v3.1.x (Check that the figure patch is in bbox_artists before trying to remove.)
- PR #14374: Check that the figure patch is in bbox_artists before trying to remove.
- PR #14040: Gracefully handle non-finite z in tricontour (issue #10167)
- PR #14342: Backport PR #14326 on branch v3.1.x (Correctly apply PNG palette when building ImageBase through Pillow.)
- PR #14326: Correctly apply PNG palette when building ImageBase through Pillow.
- PR #14341: Backport PR #14337 on branch v3.1.x (Docstring cleanup)
- PR #14337: Docstring cleanup
- PR #14325: Backport PR #14126 on branch v3.1.x (Simplify grouped bar chart example)
- PR #14324: Backport PR #14139 on branch v3.1.x (TST: be more explicit about identifying qt4/qt5 imports)
- PR #14126: Simplify grouped bar chart example
- PR #14323: Backport PR #14290 on branch v3.1.x (Convert SymmetricalLogScale to numpydoc)
- PR #14139: TST: be more explicit about identifying qt4/qt5 imports
- PR #14290: Convert SymmetricalLogScale to numpydoc
- PR #14321: Backport PR #14313 on branch v3.1.x
- PR #14313: Support masked array inputs for to_rgba and to_rgba_array.
- PR #14320: Backport PR #14319 on branch v3.1.x (Don't set missing history buttons.)

- PR #14319: Don't set missing history buttons.
- PR #14317: Backport PR #14295: Fix bug in SymmetricalLogTransform.
- PR #14302: Backport PR #14255 on branch v3.1.x (Improve docsstring of Axes.streamplot)
- PR #14255: Improve docsstring of Axes.streamplot
- PR #14295: Fix bug in SymmetricalLogTransform.
- PR #14294: Backport PR #14282 on branch v3.1.x (Fix toolmanager's destroy subplots in tk)
- PR #14282: Fix toolmanager's destroy subplots in tk
- PR #14292: Backport PR #14289 on branch v3.1.x (BUG: Fix performance regression when plotting values from Numpy array sub-classes)
- PR #14289: BUG: Fix performance regression when plotting values from Numpy array sub-classes
- PR #14287: Backport PR #14286 on branch v3.1.x (fix minor typo)
- PR #14284: Backport PR #14279 on branch v3.1.x (In case fallback to Agg fails, let the exception propagate out.)
- PR #14254: Merge up 30x
- PR #14279: In case fallback to Agg fails, let the exception propagate out.
- PR #14268: Backport PR #14261 on branch v3.1.x (Updated polar documentation)
- PR #14261: Updated polar documentation
- PR #14264: Backport PR #14260 on branch v3.1.x (Remove old OSX FAQ page)
- PR #14260: Remove old OSX FAQ page
- PR #14249: Backport PR #14243 on branch v3.1.x (Update docstring of makeMappingArray)
- PR #14250: Backport PR #14149 on branch v3.1.x
- PR #14252: Backport PR #14248 on branch v3.1.x (Fix TextBox not respecting eventson)
- PR #14253: Backport PR #13596 on branch v3.1.x (Normalize properties passed to bxp().)
- PR #14251: Backport PR #14241 on branch v3.1.x (Fix linear segmented colormap with one element)
- PR #13596: Normalize properties passed to bxp().
- PR #14248: Fix TextBox not respecting eventson
- PR #14241: Fix linear segmented colormap with one element

- PR #14243: Update docstring of makeMappingArray
- PR #14238: Backport PR #14164 on branch v3.1.x (Fix regexp for dvipng version detection)
- PR #14149: Avoid using `axis([xlo, xhi, ylo, yhi])` in examples.
- PR #14164: Fix regexp for dvipng version detection
- PR #13739: Fix pressing tab breaks keymap in CanvasTk

Issues (30):

- #14620: Plotting on a log/logit scale overwrites axis inverting
- #14615: Inverting an axis using its limits does not work for log scale
- #14577: Calling invert_yaxis() on a 3D plot has either no effect or removes ticks
- #14602: NavigationToolbar2Tk save_figure function bug
- #1219: Show fails on figures created with the object-oriented system
- #10167: Segmentation fault with tricontour
- #13723: RuntimeError when saving PDFs via parallel processes (not threads!)
- #14315: Improvement: Better error message if kiwisolver fails to import
- #14356: matplotlib.units.ConversionError on scatter of dates with a NaN in the first position
- #14467: Docs for plt.ginput() have the wrong default value for show_clicks keyword argument.
- #14225: Matplotlib crashes on windows while maximizing plot window when using Multicursor
- #14458: DOC: small inconsistency in errobar docstring
- #14372: Document that view_init() arguments should be in degrees
- #12201: issues clearing rubberband on nbagg at non-default browser zoom
- #13576: pcolorfast misbehaves when changing axis limits
- #14303: Unable to import matplotlib on Windows 10 v1903
- #14283: RendererSVG CSS 'white-space' property conflicts with default HTML CSS
- #14293: imshow() producing "inverted" colors since 3.0.3
- #14322: Cannot import matplotlib with Python 3.7.x on Win10Pro
- #14137: Qt5 test auto-skip is not working correctly
- #14301: scatter() fails on nan-containing input when providing edgecolor
- #14318: Don't try to set missing history buttons.
- #14265: symlog looses some points since 3.1.0 (example given)

- #14274: BUG: plotting with Numpy array subclasses is slow with Matplotlib 3.1.0 (regression)
- #14263: import pyplot issue -
- #14227: Update "working with Mpl on OSX" docs
- #13448: boxplot doesn't normalize properties before applying them
- #14226: Modify matplotlib TextBox value without triggering callback
- #14232: LinearSegmentedColormap with N=1 gives confusing error message
- #10365: Scatter plot with non-sequence ´c´ color should give a better Error message.

7.1.8 GitHub Stats for Matplotlib 3.1.1

GitHub stats for 2019/05/18 - 2019/06/30 (tag: v3.1.0)

These lists are automatically generated, and may be incomplete or contain duplicates.

We closed 30 issues and merged 120 pull requests. The full list can be seen on GitHub

The following 30 authors contributed 323 commits.

- Adam Gomaa
- Antony Lee
- Ben Root
- Christer Jensen
- chuanzhu xu
- David Stansby
- Deng Tian
- djdt
- Dora Fraeman Caswell
- Elan Ernest
- Elliott Sales de Andrade
- Eric Firing
- Filipe Fernandes
- Ian Thomas
- ImportanceOfBeingErnest
- Jody Klymak
- Johannes H. Jensen

- Jonas Camillus Jeppesen
- LeiSurrre
- Matt Adamson
- MeeseeksMachine
- Molly Rossow
- Nathan Goldbaum
- Nelle Varoquaux
- Paul Ivanov
- RoryIAngus
- Ryan May
- Thomas A Caswell
- Thomas Robitaille
- Tim Hoffmann

GitHub issues and pull requests:

Pull Requests (120):

- PR #14636: Don't capture stderr in _check_and_log_subprocess.
- PR #14655: Backport PR #14649 on branch v3.1.x (Fix appveyor conda py37)
- PR #14649: Fix appveyor conda py37
- PR #14646: Backport PR #14640 on branch v3.1.x (FIX: allow secondary axes to be non-linear)
- PR #14640: FIX: allow secondary axes to be non-linear
- PR #14643: Second attempt at fixing axis inversion (for mpl3.1).
- PR #14623: Fix axis inversion with loglocator and logitlocator.
- PR #14619: Backport PR #14598 on branch v3.1.x (Fix inversion of shared axes.)
- PR #14621: Backport PR #14613 on branch v3.1.x (Cleanup DateFormatter docstring.)
- PR #14622: Backport PR #14611 on branch v3.1.x (Update some axis docstrings.)
- PR #14611: Update some axis docstrings.
- PR #14613: Cleanup DateFormatter docstring.
- PR #14598: Fix inversion of shared axes.
- PR #14610: Backport PR #14579 on branch v3.1.x (Fix inversion of 3d axis.)

- PR #14579: Fix inversion of 3d axis.
- PR #14600: Backport PR #14599 on branch v3.1.x (DOC: Add numpngw to third party packages.)
- PR #14574: Backport PR #14568 on branch v3.1.x (Don't assume tk canvas have a manager attached.)
- PR #14568: Don't assume tk canvas have a manager attached.
- PR #14571: Backport PR #14566 on branch v3.1.x (Move setting of AA_EnableHighDpiScaling before creating QApplication.)
- PR #14566: Move setting of AA_EnableHighDpiScaling before creating QApplication.
- PR #14541: Backport PR #14535 on branch v3.1.x (Invalidate FT2Font cache when fork()ing.)
- PR #14535: Invalidate FT2Font cache when fork()ing.
- PR #14522: Backport PR #14040 on branch v3.1.x (Gracefully handle non-finite z in tricontour (issue #10167))
- PR #14434: Backport PR #14296 on branch v3.1.x (Fix barbs to accept array of bool for `flip_barb`)
- PR #14518: Backport PR #14509 on branch v3.1.x (Fix too large icon spacing in Qt5 on non-HiDPI screens)
- PR #14509: Fix too large icon spacing in Qt5 on non-HiDPI screens
- PR #14514: Backport PR #14256 on branch v3.1.x (Improve docstring of Axes.barbs)
- PR #14256: Improve docstring of Axes.barbs
- PR #14505: Backport PR #14395 on branch v3.1.x (MAINT: work around non-zero exit status of "pdftops -v" command.)
- PR #14504: Backport PR #14445 on branch v3.1.x (FIX: fastpath clipped artists)
- PR #14502: Backport PR #14451 on branch v3.1.x (FIX: return points rather than path to fix regression)
- PR #14445: FIX: fastpath clipped artists
- PR #14497: Backport PR #14491 on branch v3.1.x (Fix uses of PyObject_IsTrue.)
- PR #14491: Fix uses of PyObject_IsTrue.
- PR #14492: Backport PR #14490 on branch v3.1.x (Fix links of parameter types)
- PR #14490: Fix links of parameter types
- PR #14489: Backport PR #14459 on branch v3.1.x (Cleanup docstring of DraggableBase.)

- PR #14459: Cleanup docstring of DraggableBase.
- PR #14485: Backport #14429 on v3.1.x
- PR #14486: Backport #14403 on v3.1.
- PR #14429: FIX: if the first elements of an array are masked keep checking
- PR #14481: Backport PR #14475 on branch v3.1.x (change ginoput docstring to match behavior)
- PR #14482: Backport PR #14464 on branch v3.1.x (Mention origin and extent tutorial in API docs for origin kwarg)
- PR #14464: Mention origin and extent tutorial in API docs for origin kwarg
- PR #14468: Backport PR #14449: Improve docs on gridspec
- PR #14475: change ginoput docstring to match behavior
- PR #14477: Backport PR #14461 on branch v3.1.x (Fix out of bounds read in backend_tk.)
- PR #14476: Backport PR #14474 on branch v3.1.x (Fix default value in docstring of errorbar func)
- PR #14461: Fix out of bounds read in backend_tk.
- PR #14474: Fix default value in docstring of errorbar func
- PR #14473: Backport PR #14472 on branch v3.1.x (Fix NameError in example code for setting label via method)
- PR #14472: Fix NameError in example code for setting label via method
- PR #14449: Improve docs on gridspec
- PR #14450: Backport PR #14422 on branch v3.1.x (Fix ReST note in span selector example)
- PR #14446: Backport PR #14438 on branch v3.1.x (Issue #14372 - Add degrees to documentation)
- PR #14438: Issue #14372 - Add degrees to documentation
- PR #14437: Backport PR #14387 on branch v3.1.x (Fix clearing rubberband on nbagg)
- PR #14387: Fix clearing rubberband on nbagg
- PR #14435: Backport PR #14425 on branch v3.1.x (Lic restore license paint)
- PR #14296: Fix barbs to accept array of bool for `flip_barb`
- PR #14430: Backport PR #14397 on branch v3.1.x (Correctly set clip_path on pcolorfast return artist.)
- PR #14397: Correctly set clip_path on pcolorfast return artist.

- PR #14409: Backport PR #14335 on branch v3.1.x (Add explanation of animation.embed_limit to matplotlibrc.template)
- PR #14335: Add explanation of animation.embed_limit to matplotlibrc.template
- PR #14403: Revert "Preserve whitespace in svg output."
- PR #14407: Backport PR #14406 on branch v3.1.x (Remove extra iint in math_symbol_table for document)
- PR #14398: Backport PR #14394 on branch v3.1.x (Update link to "MathML torture test".)
- PR #14394: Update link to "MathML torture test".
- PR #14389: Backport PR #14388 on branch v3.1.x (Fixed one little spelling error)
- PR #14385: Backport PR #14316 on branch v3.1.x (Improve error message for kiwisolver import error (DLL load failed))
- PR #14388: Fixed one little spelling error
- PR #14384: Backport PR #14369 on branch v3.1.x (Don't use deprecated mathcircled in docs.)
- PR #14316: Improve error message for kiwisolver import error (DLL load failed)
- PR #14369: Don't use deprecated mathcircled in docs.
- PR #14375: Backport PR #14374 on branch v3.1.x (Check that the figure patch is in bbox_artists before trying to remove.)
- PR #14374: Check that the figure patch is in bbox_artists before trying to remove.
- PR #14040: Gracefully handle non-finite z in tricontour (issue #10167)
- PR #14342: Backport PR #14326 on branch v3.1.x (Correctly apply PNG palette when building ImageBase through Pillow.)
- PR #14326: Correctly apply PNG palette when building ImageBase through Pillow.
- PR #14341: Backport PR #14337 on branch v3.1.x (Docstring cleanup)
- PR #14337: Docstring cleanup
- PR #14325: Backport PR #14126 on branch v3.1.x (Simplify grouped bar chart example)
- PR #14324: Backport PR #14139 on branch v3.1.x (TST: be more explicit about identifying qt4/qt5 imports)
- PR #14126: Simplify grouped bar chart example
- PR #14323: Backport PR #14290 on branch v3.1.x (Convert SymmetricalLogScale to numpydoc)

- PR #14139: TST: be more explicit about identifying qt4/qt5 imports
- PR #14290: Convert SymmetricalLogScale to numpydoc
- PR #14321: Backport PR #14313 on branch v3.1.x
- PR #14313: Support masked array inputs for to_rgba and to_rgba_array.
- PR #14320: Backport PR #14319 on branch v3.1.x (Don't set missing history buttons.)
- PR #14319: Don't set missing history buttons.
- PR #14317: Backport PR #14295: Fix bug in SymmetricalLogTransform.
- PR #14302: Backport PR #14255 on branch v3.1.x (Improve docsstring of Axes.streamplot)
- PR #14255: Improve docsstring of Axes.streamplot
- PR #14295: Fix bug in SymmetricalLogTransform.
- PR #14294: Backport PR #14282 on branch v3.1.x (Fix toolmanager's destroy subplots in tk)
- PR #14282: Fix toolmanager's destroy subplots in tk
- PR #14292: Backport PR #14289 on branch v3.1.x (BUG: Fix performance regression when plotting values from Numpy array sub-classes)
- PR #14289: BUG: Fix performance regression when plotting values from Numpy array sub-classes
- PR #14287: Backport PR #14286 on branch v3.1.x (fix minor typo)
- PR #14284: Backport PR #14279 on branch v3.1.x (In case fallback to Agg fails, let the exception propagate out.)
- PR #14254: Merge up 30x
- PR #14279: In case fallback to Agg fails, let the exception propagate out.
- PR #14268: Backport PR #14261 on branch v3.1.x (Updated polar documentation)
- PR #14261: Updated polar documentation
- PR #14264: Backport PR #14260 on branch v3.1.x (Remove old OSX FAQ page)
- PR #14260: Remove old OSX FAQ page
- PR #14249: Backport PR #14243 on branch v3.1.x (Update docstring of makeMappingArray)
- PR #14250: Backport PR #14149 on branch v3.1.x
- PR #14252: Backport PR #14248 on branch v3.1.x (Fix TextBox not respecting eventson)

- PR #14253: Backport PR #13596 on branch v3.1.x (Normalize properties passed to bxp().)
- PR #14251: Backport PR #14241 on branch v3.1.x (Fix linear segmented colormap with one element)
- PR #13596: Normalize properties passed to bxp().
- PR #14248: Fix TextBox not respecting eventson
- PR #14241: Fix linear segmented colormap with one element
- PR #14243: Update docstring of makeMappingArray
- PR #14238: Backport PR #14164 on branch v3.1.x (Fix regexp for dvipng version detection)
- PR #14149: Avoid using `axis([xlo, xhi, ylo, yhi])` in examples.
- PR #14164: Fix regexp for dvipng version detection
- PR #13739: Fix pressing tab breaks keymap in CanvasTk

Issues (30):

- #14620: Plotting on a log/logit scale overwrites axis inverting
- #14615: Inverting an axis using its limits does not work for log scale
- #14577: Calling invert_yaxis() on a 3D plot has either no effect or removes ticks
- #14602: NavigationToolbar2Tk save_figure function bug
- #1219: Show fails on figures created with the object-oriented system
- #10167: Segmentation fault with tricontour
- #13723: RuntimeError when saving PDFs via parallel processes (not threads!)
- #14315: Improvement: Better error message if kiwisolver fails to import
- #14356: matplotlib.units.ConversionError on scatter of dates with a NaN in the first position
- #14467: Docs for plt.ginput() have the wrong default value for show_clicks keyword argument.
- #14225: Matplotlib crashes on windows while maximizing plot window when using Multicursor
- #14458: DOC: small inconsistency in errobar docstring
- #14372: Document that view_init() arguments should be in degrees
- #12201: issues clearing rubberband on nbagg at non-default browser zoom
- #13576: pcolorfast misbehaves when changing axis limits
- #14303: Unable to import matplotlib on Windows 10 v1903

- #14283: RendererSVG CSS 'white-space' property conflicts with default HTML CSS
- #14293: imshow() producing "inverted" colors since 3.0.3
- #14322: Cannot import matplotlib with Python 3.7.x on Win10Pro
- #14137: Qt5 test auto-skip is not working correctly
- #14301: scatter() fails on nan-containing input when providing edgecolor
- #14318: Don't try to set missing history buttons.
- #14265: symlog looses some points since 3.1.0 (example given)
- #14274: BUG: plotting with Numpy array subclasses is slow with Matplotlib 3.1.0 (regression)
- #14263: import pyplot issue -
- #14227: Update "working with Mpl on OSX" docs
- #13448: boxplot doesn't normalize properties before applying them
- #14226: Modify matplotlib TextBox value without triggering callback
- #14232: LinearSegmentedColormap with N=1 gives confusing error message
- #10365: Scatter plot with non-sequence ´c´ color should give a better Error message.

7.1.9 GitHub Stats for Matplotlib 3.1.0

GitHub stats for 2018/09/18 - 2019/05/13 (tag: v3.0.0)

These lists are automatically generated, and may be incomplete or contain duplicates.

We closed 161 issues and merged 918 pull requests. The full list can be seen on GitHub

The following 150 authors contributed 3426 commits.

- Abhinuv Nitin Pitale
- Adam J. Stewart
- Alistair Muldal
- Alon Hershenhorn
- Andras Deak
- Ankur Dedania
- Antony Lee
- Anubhav Shrimal
- Ao Liu (frankliuao)

- Ayappan P
- azure-pipelines[bot]
- Bas van Schaik
- Ben Root
- Benjamin Bengfort
- Benjamin Congdon
- Bharat123rox
- Brigitta Sipocz
- btang02
- Carsten
- Carsten Schelp
- Cho Yin Yong
- Chris Zimmerman
- Christer Jensen
- Christoph Gohlke
- Christoph Reiter
- Christopher Bradshaw
- Colin
- Colin Carroll
- dabana
- Dana-Farber
- Daniele Nicolodi
- DanielMatu
- David Haberthür
- David Stansby
- Dietmar Schwertberger
- Dmitry Mottl
- E. G. Patrick Bos
- Elan Ernest
- Elliott Sales de Andrade
- Eric Firing
- Eric Larson

- Eric Wieser
- esvhd
- fredrik-1
- fuzzythecat
- Galen Lynch
- Gazing
- gwin-zegal
- hannah
- Harshal Prakash Patankar
- hershen
- Ildar Akhmetgaleev
- ImportanceOfBeingErnest
- Isa Hassen
- Jae-Joon Lee
- James A. Bednar
- James Adams
- Jan S. (Milania1)
- Jarrod Millman
- Jessica B. Hamrick
- Jody Klymak
- Joel T. Frederico
- Joel Wanner
- Johannes H. Jensen
- Joseph Albert
- Joshua Klein
- Jouni K. Seppänen
- Jun Tan
- Kai Muehlbauer
- Katrin Leinweber
- Kayla Ngan
- Kevin Rose
- Kjell Le

- KonradAdamczyk
- ksunden
- Kyle Sunden
- Leon Loopik
- Levi Kilcher
- LevN0
- luftek
- Maik Riechert
- Marcel Martin
- Mark Harfouche
- Marko Baštovanović
- Matthias Bussonnier
- Matthias Geier
- Matti Picus
- MeeseeksMachine
- Michael Droettboom
- Michael Jancsy
- Mike Frysinger
- Molly Rossow
- MortenSHUTE
- mromanie
- nathan78906
- Nelle Varoquaux
- Nick Papior
- Nicolas Courtemanche
- Nikita Kniazev
- njwhite
- Oliver Natt
- Paul
- Paul Hobson
- Paul Ivanov
- Paul J. Koprowski

- pharshalp
- Phil Elson
- Pierre Thibault
- QiCuiHub
- Rasmus Diederichsen
- Ratin_Kumar
- Rob Harrigan
- Roman Yurchak
- Ryan May
- Ryan Morshead
- Saket Choudhary
- saksmito
- SBCV
- Sebastian Bullinger
- Sebastian Hegler
- Seunghoon Park
- simon-kraeusel
- smheidrich
- Stephane Raynaud
- Stephen-Chilcote
- sxntxn
- Taehoon Lee
- Takafumi Arakaki
- Taras
- Taras Kuzyo
- teresy
- Thein Oo
- Thomas A Caswell
- Thomas Hisch
- Thomas Robitaille
- thoo
- Tim Hoffmann

- Tobia De Koninck
- Tobias Megies
- Tyler Makaro
- V. Armando Solé
- Viraj Mohile
- Will Handley
- woclass
- Yasaman-Mah
- yeo
- Yuxin Wu
- Yuya
- Zhili (Jerry) Pan
- zhoubecky

GitHub issues and pull requests:

Pull Requests (918):

- PR #14209: Backport PR #14197 on branch v3.1.x (Minor cleanup of acorr/xcoor docs)
- PR #14210: Make intro tutorial less jargony.
- PR #14197: Minor cleanup of acorr/xcoor docs
- PR #14203: Backport PR #14202 on branch v3.1.x (Fix docstring of Line2D.set_data.)
- PR #14202: Fix docstring of Line2D.set_data.
- PR #14196: Backport PR #14188 on branch v3.1.x (Clarify scope of MouseEvent attributes)
- PR #14188: Clarify scope of MouseEvent attributes
- PR #14194: Backport PR #14167 on branch v3.1.x (Fix backend_pgf header.)
- PR #14193: Backport PR #14153 on branch v3.1.x (Update qt_compat.py test for already imported binding.)
- PR #14167: Fix backend_pgf header.
- PR #14153: Update qt_compat.py test for already imported binding.
- PR #14190: Backport PR #14176 on branch v3.1.x (Merge doc/api/api_overview and doc/api/index.)
- PR #14192: Unbreak testsuite for pytest 4.5.

- PR #14189: Backport PR #14186 on branch v3.1.x (Update FancyBboxPatch docs to numpydoc style)
- PR #14176: Merge doc/api/api_overview and doc/api/index.
- PR #14186: Update FancyBboxPatch docs to numpydoc style
- PR #14187: Backport PR #13169 on branch v3.1.x (Add example code for current logo)
- PR #14165: Backport PR #14156 on branch v3.1.x (Fix glyph loading in textpath.)
- PR #14156: Fix glyph loading in textpath.
- PR #14162: Backport PR #14150 on branch v3.1.x (Fix deprecation of withdash for figtext().)
- PR #14150: Fix deprecation of withdash for figtext().
- PR #14136: Backport PR #14109 on branch v3.1.x
- PR #14109: Some simple pyplot doc improvements
- PR #14129: Backport PR #14117 on branch v3.1.x (Simplify ribbon_box example.)
- PR #14128: Backport PR #14057 on branch v3.1.x (Improve Gradient bar example)
- PR #14127: Backport PR #14125 on branch v3.1.x (Remove extra keyword from pytest.skip call.)
- PR #14117: Simplify ribbon_box example.
- PR #14057: Improve Gradient bar example
- PR #14125: Remove extra keyword from pytest.skip call.
- PR #14123: Backport PR #14119 on branch v3.1.x (Add ridge_map to third party packages documentation)
- PR #14119: Add ridge_map to third party packages documentation
- PR #14103: Backport PR #14088 on branch v3.1.x (Cleanup major_minor_demo.)
- PR #14102: Backport PR #14100 on branch v3.1.x (Improve docstring of axes_zoom_effect example.)
- PR #14099: Backport PR #14090 on branch v3.1.x (Pep8ify some variable names in examples.)
- PR #14100: Improve docstring of axes_zoom_effect example.
- PR #14088: Cleanup major_minor_demo.
- PR #14090: Pep8ify some variable names in examples.

- PR #14097: Backport PR #14079 on branch v3.1.x (Consistently use axs.flat instead of axs.flatten())
- PR #14095: Backport PR #14087 on branch v3.1.x (Cleanup date example.)
- PR #14094: Backport PR #14029 on branch v3.1.x (Fix doc building with numpydoc 0.9)
- PR #14093: Backport PR #14052 on branch v3.1.x (Check axes identity in image.contains.)
- PR #14092: Backport PR #14056 on branch v3.1.x (FIX: do not try to manage the visibility of un-drawn ticks)
- PR #14091: Backport PR #14078 on branch v3.1.x (Minor fix in multiple subplots example)
- PR #14079: Consistently use axs.flat instead of axs.flatten()
- PR #14087: Cleanup date example.
- PR #14029: Fix doc building with numpydoc 0.9
- PR #14052: Check axes identity in image.contains.
- PR #14056: FIX: do not try to manage the visibility of un-drawn ticks
- PR #14078: Minor fix in multiple subplots example
- PR #14080: Backport PR #14069 on branch v3.1.x (Don't try to use the colorbar formatter to format RGBA data.)
- PR #14069: Don't try to use the colorbar formatter to format RGBA data.
- PR #14074: Backport PR #14019 on branch v3.1.x (Update docstring of locator_params())
- PR #14019: Update docstring of locator_params()
- PR #14066: Backport PR #14053 on branch v3.1.x (Improve fill() example)
- PR #14065: Backport PR #14059 on branch v3.1.x (Improve Scatter hist example)
- PR #14067: Backport PR #14062 on branch v3.1.x (Improve advanced quiver example)
- PR #14062: Improve advanced quiver example
- PR #14053: Improve fill() example
- PR #14059: Improve Scatter hist example
- PR #14064: Backport PR #14043 on branch v3.1.x (Ensure errorbars are always drawn on top of bars in ax.bar)
- PR #14043: Ensure errorbars are always drawn on top of bars in ax.bar
- PR #14061: Backport PR #14051 on branch v3.1.x (Add Yellowbrick to third party packages)

- PR #14051: Add Yellowbrick to third party packages
- PR #14050: Backport PR #14048 on branch v3.1.x (Fix Animation.save)
- PR #14049: Backport PR #14047 on branch v3.1.x (Remove references to "Draws" in matplotlib.patches)
- PR #14048: Fix Animation.save
- PR #14047: Remove references to "Draws" in matplotlib.patches
- PR #14037: Backport PR #14033 on branch v3.1.x (Reword add_subplot docstring.)
- PR #14036: Backport PR #14001 on branch v3.1.x ([BUG] DOC: Remove broken references to vischeck)
- PR #14033: Reword add_subplot docstring.
- PR #14032: Backport PR #14030 on branch v3.1.x (Update colorcet link)
- PR #14030: Update colorcet link
- PR #14027: Backport PR #14026 on branch v3.1.x (Fix bug in plot_directive that caused links to plots in different formats to be missing)
- PR #14026: Fix bug in plot_directive that caused links to plots in different formats to be missing
- PR #14012: Backport PR #14008 on branch v3.1.x (Don't install tests by default.)
- PR #14017: Backport PR #14015 on branch v3.1.x (Fix docstring of pyplot.clim())
- PR #14015: Fix docstring of pyplot.clim()
- PR #14008: Don't install tests by default.
- PR #14006: Backport PR #13998 on branch v3.1.x (Fix patch contains logic for patches that don't have any codes)
- PR #14005: Backport PR #14004 on branch v3.1.x (DOC: pin numpydoc to less than 0.9)
- PR #13998: Fix patch contains logic for patches that don't have any codes
- PR #13999: Backport PR #13992 on branch v3.1.x (FIX: undeprecate MaxNLocator default_params)
- PR #13997: Backport PR #13995 on branch v3.1.x (DOC: explain zorder for gridlines in grid docstring)
- PR #13992: FIX: undeprecate MaxNLocator default_params
- PR #13995: DOC: explain zorder for gridlines in grid docstring
- PR #13990: Backport PR #13989 on branch v3.1.x (FIX: update not replace hist_kwargs when density is passed)

- PR #13989: FIX: update not replace hist_kwargs when density is passed
- PR #13975: Backport PR #13966 on branch v3.1.x (Fix colorbar setting without artist)
- PR #13976: Backport PR #13973 on branch v3.1.x (BUG: Ensure docstrings are not accessed with -OO)
- PR #13856: What's new page for 3.1
- PR #13966: Fix colorbar setting without artist
- PR #13973: BUG: Ensure docstrings are not accessed with -OO
- PR #13969: Backport PR #13950 on branch v3.1.x (confidence_ellipse_markup)
- PR #13950: confidence_ellipse_markup
- PR #13965: Backport PR #13962 on branch v3.1.x (Fix typo in code example in docstring.)
- PR #13964: Backport PR #13870 on branch v3.1.x (3.1.0 API changes page)
- PR #13962: Fix typo in code example in docstring.
- PR #13870: 3.1.0 API changes page
- PR #13961: Backport PR #13914 on branch v3.1.x (Improve Rainbow text example)
- PR #13960: Backport PR #13958 on branch v3.1.x (Remove transparent fancy legend example)
- PR #13914: Improve Rainbow text example
- PR #13958: Remove transparent fancy legend example
- PR #13956: Backport PR #13908 on branch v3.1.x (Enh control tick deconflict2)
- PR #13955: Backport PR #13941 on branch v3.1.x (Add project_urls to setup)
- PR #13908: Enh control tick deconflict2
- PR #13954: Backport PR #13949 on branch v3.1.x (DOC: Add documentation to Text.set_fontfamily)
- PR #13941: Add project_urls to setup
- PR #13949: DOC: Add documentation to Text.set_fontfamily
- PR #13951: Backport PR #13939 on branch v3.1.x (Bunch of docstring cleanups.)
- PR #13939: Bunch of docstring cleanups.
- PR #13947: Backport PR #13897 on branch v3.1.x (numpydocification.)
- PR #13897: numpydocification.
- PR #13946: Backport PR #13924 on branch v3.1.x (Followup to deprecation of usetex parameter in get_text_path.)

- PR #13924: Followup to deprecation of usetex parameter in get_text_path.
- PR #13916: Backport PR #13850 on branch v3.1.x (Cleanup STIX Font Demo)
- PR #13915: Backport PR #13835 on branch v3.1.x (Improve Conectionstyle Demo)
- PR #13850: Cleanup STIX Font Demo
- PR #13835: Improve Conectionstyle Demo
- PR #13846: Backport PR #13836 on branch v3.1.x (MNT: account for cpython deprecations)
- PR #13898: Backport PR #13896 on branch v3.1.x (Fix cbook.boxplot_stats docstring)
- PR #13896: Fix cbook.boxplot_stats docstring
- PR #13893: Backport PR #13890 on branch v3.1.x (rst seealso -> numpydoc "See Also".)
- PR #13890: rst seealso -> numpydoc "See Also".
- PR #13888: Backport PR #13862 on branch v3.1.x (Move 3.x API changes to prev_api_changes)
- PR #13862: Move 3.x API changes to prev_api_changes
- PR #13882: Backport PR #13867 on branch v3.1.x (Rename "docs" to "contents" in navigation bar)
- PR #13867: Rename "docs" to "contents" in navigation bar
- PR #13881: Backport PR #13874 on branch v3.1.x (Remove redundant call to Formatter.set_locs() before .format_ticks().)
- PR #13874: Remove redundant call to Formatter.set_locs() before .format_ticks().
- PR #13871: Backport PR #13868 on branch v3.1.x (Correctly handle fallout of defining PY_SSIZE_T_CLEAN on Windows.)
- PR #13869: Backport PR #13861 on branch v3.1.x (Fix remaining links in docs)
- PR #13868: Correctly handle fallout of defining PY_SSIZE_T_CLEAN on Windows.
- PR #13861: Fix remaining links in docs
- PR #13849: Backport PR #13845 on branch v3.1.x (Fix some broken documentation links)
- PR #13845: Fix some broken documentation links
- PR #13836: MNT: account for cpython deprecations
- PR #13841: Backport PR #12928 on branch v3.1.x (textpath encoding)

- PR #13842: Backport PR #13827 on branch v3.1.x (Better MovieWriter init error message)
- PR #13838: Backport PR #13570 on branch v3.1.x (Add new example for plotting a confidence_ellipse)
- PR #13827: Better MovieWriter init error message
- PR #13839: Backport PR #13815 on branch v3.1.x (Numpydocify FontManager.findfont())
- PR #13837: Backport PR #8638 on branch v3.1.x (FIX: if bins input to hist is str, treat like no bins)
- PR #12928: textpath encoding
- PR #13815: Numpydocify FontManager.findfont()
- PR #13570: Add new example for plotting a confidence_ellipse
- PR #8638: FIX: if bins input to hist is str, treat like no bins
- PR #13831: Backport PR #13780 on branch v3.1.x (numpydoc ListedColormap parameters)
- PR #13780: numpydoc ListedColormap parameters
- PR #13830: Backport PR #13829 on branch v3.1.x (numpydoc IndexFormatter)
- PR #13829: numpydoc IndexFormatter
- PR #13828: Backport PR #13821 on branch v3.1.x (Remove mathcircled from mathtext docs following its deprecation.)
- PR #13821: Remove mathcircled from mathtext docs following its deprecation.
- PR #13822: Backport PR #13817 on branch v3.1.x (Remove borders from barcode example)
- PR #13820: Backport PR #13816 on branch v3.1.x (Correct windows env variable format)
- PR #13816: Correct windows env variable format
- PR #13817: Remove borders from barcode example
- PR #13814: Merge pull request #13805 from timhoffm/pin-sphinx-1.x
- PR #13813: Backport PR #13764 on branch v3.1.x (Deprecate mathcircled.)
- PR #13764: Deprecate mathcircled.
- PR #13805: Pin Sphinx to 1.x
- PR #13807: Backport PR #13800 on branch v3.1.x (Doc typos.)
- PR #13800: Doc typos.
- PR #13806: Backport PR #13771 on branch v3.1.x (patches.Arc docstring update #13759)

- PR #13804: Backport PR #13766 on branch v3.1.x (Search for fonts in XDG directory as well.)
- PR #13771: patches.Arc docstring update #13759
- PR #13766: Search for fonts in XDG directory as well.
- PR #13794: Backport PR #13695 on branch v3.1.x (numpydocify transform_angles.)
- PR #13793: Backport PR #13762 on branch v3.1.x (Cleanup marker_reference example.)
- PR #13792: Backport PR #13789 on branch v3.1.x (BUG: Fix function signature mismatch for set_clim)
- PR #13791: Backport PR #13787 on branch v3.1.x (Fix failure to import matplotlib.animation on Windows.)
- PR #13695: numpydocify transform_angles.
- PR #13762: Cleanup marker_reference example.
- PR #13789: BUG: Fix function signature mismatch for set_clim
- PR #13787: Fix failure to import matplotlib.animation on Windows.
- PR #13781: Backport PR #13777 on branch v3.1.x (Use class-based directive for mathmpl sphinxext.)
- PR #13790: Backport PR #13564 on branch v3.1.x (Add an option to log progress while saving animations)
- PR #13564: Add an option to log progress while saving animations
- PR #13777: Use class-based directive for mathmpl sphinxext.
- PR #13765: Backport PR #13761 on branch v3.1.x (Deprecate verbose-related rcParams.)
- PR #13761: Deprecate verbose-related rcParams.
- PR #13760: Backport PR #13719 on branch v3.1.x (Doc: Update timeline example)
- PR #13704: Backport PR #13021 on branch v3.1.x (Undesirable behaviour of MixedModeRenderer)
- PR #13758: Backport PR #13674 on branch v3.1.x (Preserve whitespace in svg output.)
- PR #13719: Doc: Update timeline example
- PR #13674: Preserve whitespace in svg output.
- PR #13755: Backport PR #13741 on branch v3.1.x (FIX: make title move above ticklabels)

- PR #13754: Backport PR #13712 on branch v3.1.x (Deprecate NavigationToolbar2QT.adj_window (unused and always None).)
- PR #13741: FIX: make title move above ticklabels
- PR #13712: Deprecate NavigationToolbar2QT.adj_window (unused and always None).
- PR #13752: Backport PR #13732 on branch v3.1.x (Fix doc markup.)
- PR #13753: Backport PR #13751 on branch v3.1.x (DOC/FIX: try merging comments)
- PR #13751: DOC/FIX: try merging comments
- PR #13732: Fix doc markup.
- PR #13750: Backport PR #13743 on branch v3.1.x (Fix doc warning)
- PR #13743: Fix doc warning
- PR #13747: Backport PR #13745 on branch v3.1.x (Fix stem(use_line_collection))
- PR #13748: Backport PR #13716 on branch v3.1.x (Kill attributes that are never used/updated.)
- PR #13716: Kill attributes that are never used/updated.
- PR #13745: Fix stem(use_line_collection)
- PR #13710: TST: only test agg_filter extensions with baseline images
- PR #13709: Backport PR #8690 on branch v3.1.x
- PR #13707: Backport PR #12760 on branch v3.1.x (Deduplicate implementation of per-backend Tools.)
- PR #13706: Backport PR #13689 on branch v3.1.x (BUG: fix scaling of quiverkey when quiver scale_units='xy')
- PR #13705: Backport PR #12419 on branch v3.1.x (Add DivergingNorm (again, again, again))
- PR #13703: Backport PR #12170 on branch v3.1.x (Deprecate considering *args, **kwargs in Timer.remove_callback.)
- PR #12760: Deduplicate implementation of per-backend Tools.
- PR #13689: BUG: fix scaling of quiverkey when quiver scale_units='xy'
- PR #12419: Add DivergingNorm (again, again, again)
- PR #8690: Adds support for rgba and rgb images to pcolorfast
- PR #13021: Undesirable behaviour of MixedModeRenderer
- PR #12170: Deprecate considering *args, **kwargs in Timer.remove_callback.

- PR #13700: Backport PR #13588 on branch v3.1.x (FIX: fallback to viewlims if no data)
- PR #13694: Backport PR #13677 on branch v3.1.x (Log all failures to extract font properties.)
- PR #13588: FIX: fallback to viewlims if no data
- PR #13692: Backport PR #13677 on branch v3.0.x (Log all failures to extract font properties.)
- PR #13677: Log all failures to extract font properties.
- PR #13691: Backport PR #13687 on branch v3.1.x (Update stem example)
- PR #13687: Update stem example
- PR #13688: Backport PR #13684 on branch v3.1.x (Use format_data_short to format image cursor data.)
- PR #13684: Use format_data_short to format image cursor data.
- PR #13686: Backport PR #13363 on branch v3.1.x (Inline iter_ticks into _update_ticks, and use that in mplot3d.)
- PR #13363: Inline iter_ticks into _update_ticks, and use that in mplot3d.
- PR #13681: Backport PR #13678 on branch v3.1.x (Fix font deduplication logic in createFontList.)
- PR #13678: Fix font deduplication logic in createFontList.
- PR #13669: Backport PR #13667 on branch v3.1.x (Fix incorrect signature in axis() doc.)
- PR #13667: Fix incorrect signature in axis() doc.
- PR #13664: Backport PR #12637 on branch v3.1.x (Tell IPython the correct GUI event loop to use for all backends.)
- PR #13665: Backport PR #13601 on branch v3.1.x (Add a make-parameter-keyword-only-with-deprecation decorator.)
- PR #13601: Add a make-parameter-keyword-only-with-deprecation decorator.
- PR #12637: Tell IPython the correct GUI event loop to use for all backends.
- PR #13662: Backport PR #13064 on branch v3.1.x (Don't explicitly add default include paths to Extensions)
- PR #13064: Don't explicitly add default include paths to Extensions
- PR #13658: Backport PR #13652 on branch v3.1.x (Fix empty FancyArrow crash)
- PR #13652: Fix empty FancyArrow crash
- PR #13655: Backport PR #11692 on branch v3.1.x (Deprecate frameon kwarg and rcParam to savefig.)

- PR #13654: Backport PR #13614 on branch v3.1.x (Fix polar get window extent)
- PR #11692: Deprecate frameon kwarg and rcParam to savefig.
- PR #13614: Fix polar get window extent
- PR #13646: Backport PR #13645 on branch v3.1.x (widgets.py fix examples connect -> mpl_connect)
- PR #13645: widgets.py fix examples connect -> mpl_connect
- PR #13644: Backport PR #13612 on branch v3.1.x (Improve Demo Text Rotation Mode)
- PR #13612: Improve Demo Text Rotation Mode
- PR #13636: Backport PR #13621 on branch v3.1.x (Remove `asfileobj=False` from a bunch of examples loading sample_data.)
- PR #13635: Backport PR #13632 on branch v3.1.x (Clarify tick collision API change doc.)
- PR #13634: Backport PR #13631 on branch v3.1.x (Switch deprecation of Tick.label to pending.)
- PR #13621: Remove `asfileobj=False` from a bunch of examples loading sample_data.
- PR #13632: Clarify tick collision API change doc.
- PR #13631: Switch deprecation of Tick.label to pending.
- PR #13628: Backport PR #13603 on branch v3.1.x
- PR #13603: FIX: continue to bail tight layout if rect supplied
- PR #13627: Backport PR #13622 on branch v3.1.x (Change title of named colors example)
- PR #13626: Backport PR #13549 on branch v3.1.x (Simplify some annotation() calls in examples.)
- PR #13624: Backport PR #13610 on branch v3.1.x (Update centered ticklabels example)
- PR #13625: Backport PR #13611 on branch v3.1.x (Fix text position in Fancytextbox demo)
- PR #13622: Change title of named colors example
- PR #13610: Update centered ticklabels example
- PR #13611: Fix text position in Fancytextbox demo
- PR #13607: Backport PR #13605 on branch v3.1.x (Warn on attempts at semi-transparent outputs in ps backend.)
- PR #13608: Backport PR #13602 on branch v3.1.x (Deprecate cbook.is_hashable.)

- PR #13602: Deprecate cbook.is_hashable.
- PR #13605: Warn on attempts at semi-transparent outputs in ps backend.
- PR #13599: Backport PR #13590 on branch v3.1.x (Doc event loop requirements for Figure.show)
- PR #13590: Doc event loop requirements for Figure.show
- PR #13597: Backport PR #12359 on branch v3.1.x (ENH: Add boolean support for axis())
- PR #13594: Backport PR #13592 on branch v3.1.x (DOC: Make canonical URLs point to versioned path.)
- PR #13592: DOC: Make canonical URLs point to versioned path.
- PR #12359: ENH: Add boolean support for axis()
- PR #13587: Backport PR #13573 on branch v3.1.x (Fix mplot3d transparency)
- PR #13573: Fix mplot3d transparency
- PR #13585: Backport PR #13578 on branch v3.1.x (Revert invalid change in Centered Ticklabels example)
- PR #13584: Backport PR #13582 on branch v3.1.x (Cleanup two font-related examples.)
- PR #13578: Revert invalid change in Centered Ticklabels example
- PR #13582: Cleanup two font-related examples.
- PR #13579: Backport PR #13477 on branch v3.1.x (FIX: make EngFormatter respect axes.unicode_minus rcParam)
- PR #13577: Backport PR #12832 on branch v3.1.x (Deprecate redundant log-scale transform classes.)
- PR #13477: FIX: make EngFormatter respect axes.unicode_minus rcParam
- PR #12832: Deprecate redundant log-scale transform classes.
- PR #13574: Backport PR #12856 on branch v3.1.x (added property usemathtext to EngFormatter)
- PR #12856: added property usemathtext to EngFormatter
- PR #13572: Backport PR #12899 on branch v3.1.x (Small cleanups.)
- PR #13571: Backport PR #11553 on branch v3.1.x (Improved Code for Segments Intersect)
- PR #12899: Small cleanups.
- PR #11553: Improved Code for Segments Intersect
- PR #13568: Backport PR #13563 on branch v3.1.x (FIX: inverted colorbar ticks)
- PR #13563: FIX: inverted colorbar ticks

- PR #13530: BUG: keep the ticks when the colorbar axis is inverted
- PR #13565: Backport PR #13550 on branch v3.1.x (Strip out Py2-compat in setupext.)
- PR #13550: Strip out Py2-compat in setupext.
- PR #13562: Backport PR #13560 on branch v3.1.x (Improve GridSpec doc)
- PR #13560: Improve GridSpec doc
- PR #13558: Backport PR #13546 on branch v3.1.x (Modified docstring of the set_ylabel and set_xlabel)
- PR #13559: Backport PR #12062 on branch v3.1.x (Separate alpha and rbg interpolation then recombine to fix issue11316)
- PR #13557: Backport PR #13548 on branch v3.1.x (Deprecate TextWithDash.)
- PR #12062: Separate alpha and rbg interpolation then recombine to fix issue11316
- PR #13546: Modified docstring of the set_ylabel and set_xlabel
- PR #13548: Deprecate TextWithDash.
- PR #13549: Simplify some annotation() calls in examples.
- PR #13552: Backport PR #11241 on branch v3.1.x (Deprecate the MATPLOTLIBDATA environment variable.)
- PR #11241: Deprecate the MATPLOTLIBDATA environment variable.
- PR #13547: Backport PR #9314 on branch v3.1.x (Simplify units.Registry.get_converter.)
- PR #13545: Backport PR #13541 on branch v3.1.x (DOC: Remove mention of 'complex' mode in specgram docstring)
- PR #9314: Simplify units.Registry.get_converter.
- PR #13541: DOC: Remove mention of 'complex' mode in specgram docstring
- PR #13539: Backport PR #12950 on branch v3.1.x (Inline or simplify FooFormatter.pprint_val.)
- PR #13538: Backport PR #12748 on branch v3.1.x (Use the builtin GTK3 FileChooser rather than our custom subclass.)
- PR #13537: Backport PR #12781 on branch v3.1.x (Lazy import of private modules)
- PR #12950: Inline or simplify FooFormatter.pprint_val.
- PR #12748: Use the builtin GTK3 FileChooser rather than our custom subclass.
- PR #12781: Lazy import of private modules
- PR #11218: fix pkg-config handling to make cross-compiling work

- PR #13531: Backport PR #11964 on branch v3.1.x (Simplify extension setup.)
- PR #11964: Simplify extension setup.
- PR #13529: Backport PR #13525 on branch v3.1.x (Move some links in rst out of running text.)
- PR #13528: Backport PR #13526 on branch v3.1.x (DOC: fix Subplot calls)
- PR #13525: Move some links in rst out of running text.
- PR #13526: DOC: fix Subplot calls
- PR #13523: Backport PR #13521 on branch v3.1.x (Small cleanup to headings of 3d examples.)
- PR #13521: Small cleanup to headings of 3d examples.
- PR #13519: Backport PR #12716 on branch v3.1.x (FIX: return the actual ax.get_window_extent)
- PR #13518: Backport PR #12839 on branch v3.1.x (BUG: Prevent Tick params calls from overwriting visibility without being told to)
- PR #12716: FIX: return the actual ax.get_window_extent
- PR #12839: BUG: Prevent Tick params calls from overwriting visibility without being told to
- PR #13517: Fix heading hierarchy in annotation tutorial.
- PR #13516: Backport PR #13514 on branch v3.1.x (Add missing show() at end of example.)
- PR #13514: Add missing show() at end of example.
- PR #13512: Backport PR #13511 on branch v3.1.x (Add missing plt.show() at end of example.)
- PR #13511: Add missing plt.show() at end of example.
- PR #13508: Backport PR #13413 on branch v3.1.x (Simplify decade up- and down-rounding, and symmetrize expansion of degenerate log scales.)
- PR #13509: Backport PR #13492 on branch v3.1.x (Doc more release updates)
- PR #13492: Doc more release updates
- PR #13413: Simplify decade up- and down-rounding, and symmetrize expansion of degenerate log scales.
- PR #13507: Backport PR #13488 on branch v3.1.x (Animation: interactive zoom/pan with blitting does not work)
- PR #13488: Animation: interactive zoom/pan with blitting does not work
- PR #13505: Backport PR #13459 on branch v3.1.x (Document histogramming pre-binned data.)

- PR #13503: Backport PR #10776 on branch v3.1.x (fix FancyArrowPatch picker fails depending on arrowstyle)
- PR #13504: Backport PR #13123 on branch v3.1.x (Add shading to Axes3D.voxels, and enable it by default)
- PR #13502: Backport PR #13180 on branch v3.1.x (Various TextPath cleanups.)
- PR #13459: Document histogramming pre-binned data.
- PR #13501: Backport PR #13209 on branch v3.1.x (Deprecate support for (n, 1)-shaped error arrays in errorbar().)
- PR #13500: Backport PR #12763 on branch v3.1.x (Remove deprecated rcParams.)
- PR #13123: Add shading to Axes3D.voxels, and enable it by default
- PR #13499: Backport PR #13303 on branch v3.1.x (Unify checking of executable info.)
- PR #10776: fix FancyArrowPatch picker fails depending on arrowstyle
- PR #13180: Various TextPath cleanups.
- PR #13498: Backport PR #13314 on branch v3.1.x (Move major/minor tick overstrike logic to Axis.)
- PR #13209: Deprecate support for (n, 1)-shaped error arrays in errorbar().
- PR #12763: Remove deprecated rcParams.
- PR #13303: Unify checking of executable info.
- PR #13497: Backport PR #13057 on branch v3.1.x (Simplify callable(self._contains) checks)
- PR #13314: Move major/minor tick overstrike logic to Axis.
- PR #13057: Simplify callable(self._contains) checks
- PR #13496: Backport PR #13465 on branch v3.1.x (FIX: polar set_rlim allow bottom-only call)
- PR #13465: FIX: polar set_rlim allow bottom-only call
- PR #13495: Backport PR #12232 on branch v3.1.x (Add helper function to check that an argument is in a list of strings.)
- PR #12232: Add helper function to check that an argument is in a list of strings.
- PR #11708: Revert "Skip wx interactive tests on OSX."
- PR #13062: Update FAQ re: batch/webserver use.
- PR #12904: Support forward/backward mouse buttons
- PR #12150: Deprecate stackrel.
- PR #13449: Let boxplot() defer rcParams application to bxp()

- PR #13425: API: un-deprecate keyword only args to set_xlim, set_ylim
- PR #13447: Update axes_grid docs
- PR #13473: Deprecate backend_wx.IDLE_DELAY.
- PR #13476: Add font to pyplot.xkcd()
- PR #13475: Cleanup titles of embedding examples.
- PR #13468: Suppress chaining of cache lookup failure in color conversion.
- PR #13467: Add "c" shorthand for "color" for the Text class.
- PR #13398: FIX: let pandas IndexInt64 work for boxplot
- PR #13375: Improve Axes selection in Qt figure options.
- PR #13421: DOC: update release guide
- PR #13275: Simple logging interface.
- PR #13427: Simplify check for tight-bbox finiteness.
- PR #13444: Allow constructing boxplots over multiple calls.
- PR #13385: Remove/rework uses of np.where where possible.
- PR #13441: Make AFM parser both more compliant and less strict.
- PR #13384: Replace np.compress by boolean indexing.
- PR #13422: Clarify IndexError for out-of-bounds indexing of gridspec.
- PR #13443: Remove some outdated comments from rcsetup.py.
- PR #13357: Inherit some docstrings in backend code.
- PR #12380: Stem speedup2
- PR #13368: FIX: Fix shape of hist output when input is multidimensional empty list
- PR #5590: [mpl_toolkits] Fix picking for things drawn on parasite axes
- PR #13323: Move the call to Formatter.set_locs into Formatter.format_ticks.
- PR #13424: Deprecate Quiver.color in favor of Quiver.get_facecolor().
- PR #13434: More smoketesting of pcolorfast.
- PR #13395: Cleanup demo_curvelinear_grid.
- PR #13411: Deemphasize numeric locations for legend() in docs.
- PR #13419: FIX: secondary_axis resize
- PR #13020: Deprecate proj3d.mod.
- PR #13030: Deprecate internal functions exposed in the public API of mplot3d
- PR #13408: test_figure style fixes.

- PR #11127: Legend for Scatter
- PR #11855: Adding the possible to add full command line in animation
- PR #13409: Add nonsingular to the locator base class, and use it in set_*lim too.
- PR #11859: ENH: add secondary x/y axis
- PR #13235: Vectorize mplot3d.art3d.zalpha.
- PR #10411: New "accepts units" decorator
- PR #13403: FIX: remove idle_event
- PR #13069: 5 minor divisions when major ticks are 2.5 units apart
- PR #13402: Fix empty reshape2d
- PR #11683: Reuse axes_grid1's AxisDict in axisartist, instead of duplicating it.
- PR #12141: Let digits toggle axes nav only if they correspond to an existing axes.
- PR #9845: Add inaxes method to FigureCanvas to check whether point is in an axes.
- PR #13396: mpl_toolkits style fixes.
- PR #11497: Make CI fail if interactive toolkits can't be tested
- PR #11595: test doc rendering
- PR #13393: Deprecate Spine.is_frame_like.
- PR #13391: Remove colour specification from some examples
- PR #13386: Replace use of np.<ufunc> by operators (</&/|).
- PR #13389: Inherit more docstrings.
- PR #13387: Fix regression in docstring.dedent_interpd.
- PR #13383: Replace np.take by normal indexing.
- PR #13381: Avoid unneeded copies from flatten().
- PR #13354: Properly deprecate non-1D inputs to pie().
- PR #13379: Remove citation entry from FAQ.
- PR #13380: Minor simplifications to scatter3d.
- PR #13173: Decorator for deleting a parameter with a deprecation period.
- PR #8205: [MRG+1] plot_date() after axhline() doesn't rescale axes
- PR #11027: Specify custom tick space heuristic in MaxNLocator
- PR #13262: Shorten setupext and remove uninformative build log entries.
- PR #13377: Add private helper to internally suppress deprecations.
- PR #13376: Undeprecate case-insensitive "long" colornames.

- PR #13373: Deprecate axis3d.Axis.get_tick_positions.
- PR #13362: Kill the unused, private _get_pixel_distance_along_axis.
- PR #12772: Improve plot() docstring.
- PR #13359: DOC: change language a bit
- PR #13351: Fix: Log Colorbar minorticks_off reverted if ticks set
- PR #13356: More spelling fixes.
- PR #13125: Simplify and tighten the docstring handling API.
- PR #13346: Simplify parsing of tuple in C extension code.
- PR #13282: MAINT install of pinned vers for travis
- PR #13234: FIX: allow colorbar mappable norm to change and do right thing
- PR #13269: Rework a bit axes addition.
- PR #13330: Add Axis.get_inverted and Axis.set_inverted.
- PR #13117: Cleanup matplotlib.use
- PR #13335: Update and factor out Axis.get_tick_positions.
- PR #13324: Cleanup ScalarFormatter; preparatory to moving it to format_ticks.
- PR #13322: Update Axis docs
- PR #13342: Update some (mostly internal) docstrings in image.py.
- PR #11848: Country specific characters in Windows user folder name when locating .tfm-file
- PR #13309: bezier cleanups.
- PR #13334: Inherit some docstrings.
- PR #13332: Rewrite convert_to_string using std::string
- PR #13336: Update imshow docs.
- PR #13331: Try forcing font cache rebuild in flaky ttc test.
- PR #12105: API: make MaxNLocator trim out-of-view ticks before returning
- PR #13329: Pin flake8<3.7 to mitigate issues with flake8-per-file-ignores
- PR #13319: Deprecate dates.{str,bytes}pdate2num.
- PR #13320: Kill some private, unused functions in dates.py.
- PR #12909: Let Formatters format all ticks at once.
- PR #13313: Better explanation of ticks
- PR #13310: Replace *kw by *args.
- PR #13285: Defer checking of tex install to when it is actually used.

- PR #13128: Parameter-renaming decorator
- PR #13307: Spelling fixes.
- PR #13304: TST: deregister pandas
- PR #13300: Trivial bezier cleanups.
- PR #11664: FIX: clean up unit conversion unpacking of data, particularly for dates and pandas series
- PR #9639: Unify querying of executable versions
- PR #13224: numpydocify (some of) mpl_toolkits.
- PR #13301: Replace np.empty + ndarray.fill by np.full.
- PR #13229: Prevent exception when running animation on Agg backend.
- PR #13263: In imsave()'s Pillow-handled case, don't create a temporary figure.
- PR #13294: Simplify some calculations in polar.py.
- PR #13295: Kill some commented-out code.
- PR #13298: Add note about thread safety to FAQ.
- PR #13299: Don't emit a non-GUI warning when building the docs on Linux.
- PR #13297: Minor cleanup to OSX FAQ.
- PR #13283: Fix doc style in add_gridspec()
- PR #13129: ENH: add a user-friendly verbose interface
- PR #13279: Remove a useless catch_warnings() from example.
- PR #13268: Select RadioButtons by closest in position.
- PR #13271: Fix animation speed in double_pendulum example
- PR #13265: Allow turning off minor ticks on Colorbar with LogNorm
- PR #13260: Improve docs for format determination in savefig()/imsave().
- PR #12379: MAINT Use np.full when possible
- PR #12905: Add optional parameter use_default_template to rc_file()
- PR #13218: Fix checking of 'labels' argument to Sankey.add.
- PR #13256: DOC: reject MEP25 due to being stalled
- PR #13255: TST pandas support bar
- PR #13251: DEBUG-log font-matching results, and print failing logs on CI.
- PR #12818: Enh arbitrary scale
- PR #13187: FIX: bar mixed units, allow ValueError as well
- PR #13232: Fix incorrect kwarg being passed to TextPath.

- PR #13250: Replace safezip() by more informative error message in errorbar().
- PR #13239: Improve sankey logging.
- PR #13247: Simplify and optimize png writing in backend_pdf.
- PR #12455: Warn when "best" loc of legend is used with lots of data
- PR #13233: Remove warning in image_annotated_heatmap, and numpydocify it.
- PR #13248: Remove an unused local variable in backend_gtk3.
- PR #13249: Deprecate an unused "internal" API.
- PR #13243: Rewrite subplots_demo
- PR #13240: FIX: spelling error of local variable in category
- PR #13026: MNT: add a logging call if a categorical string array is all convertible
- PR #13225: Fix a warning in the doc build.
- PR #13227: Make color lowercase in example to avoid warning.
- PR #13217: numpydocify Sankey.add.
- PR #10209: Various backend cleanups.
- PR #13113: Globally cache single TexManager instances.
- PR #13213: Broadcast 'orientations' arg to Sankey.add.
- PR #13219: Fix some backend_bases docstrings.
- PR #13214: Reformat Sankey exceptions.
- PR #13211: Deprecate case-insensitive colors.
- PR #13210: Suppress a warning in the test suite.
- PR #13189: Remove cairo-based backends from backend fallback.
- PR #13207: Allow saving PNGs through Pillow instead of the builtin _png module.
- PR #13124: Simplify parsing of errorbar input.
- PR #13162: DOC: better argcheck bar
- PR #8531: Added compression option to save TIFF images
- PR #13094: Allow passing arguments to PIL.Image.save().
- PR #13202: Avoid private API in some examples.
- PR #13197: Cleanup the text of two mpl_toolkits examples.
- PR #13198: Cleanup SkewT example.
- PR #11914: Remove the system_monitor example.

- PR #13196: Deemphasize comment about extremely old Matplotlib versions in example.
- PR #13190: Show returncode when subprocess test fails
- PR #13163: Add explanatory comment to annotation box example
- PR #13104: Remove some more 1-tuples.
- PR #13105: Make GridSpec.update docstring match behavior.
- PR #13127: Deprecate add_subplot(<no positional args>) silently doing nothing.
- PR #13166: Simplify Text.get_usetex.
- PR #13188: Remove an outdated doc point regarding backend selection.
- PR #13107: Cleanup BboxBase docstrings.
- PR #13108: Capitalize some docstrings.
- PR #13115: Check for sphinx_copybutton when building the docs
- PR #13151: Update RadioButtons docs numpydoc style
- PR #13178: Remove :func: markup from mlab docstrings.
- PR #7461: [WIP] add matrix checking function for quiver input
- PR #13089: Ensure that arguments to quiver() are not matrices.
- PR #13179: Avoid calling a deprecated API in axis_artist.
- PR #13170: Don't try to find TeX-only fonts when layouting TeX text.
- PR #12957: Search also for user fonts on Windows (#12954)
- PR #12951: Make Text._get_layout simpler to follow.
- PR #11385: Add a get_zaxis method for 3d axes.
- PR #13172: Hyperlink DOIs to preferred resolver
- PR #13171: Document how to make colorbars "without" a ScalarMappable.
- PR #12903: FIX: (broken)bar(h) math before units
- PR #13167: Typos on subplot comments and example
- PR #13005: Improve error messages for unit conversion
- PR #13147: Extend joinstyle example
- PR #13165: Change doc string for Axes.arrow()
- PR #13155: Let ffmpeg report errors.
- PR #13149: Update errorbar limits example
- PR #13074: Move _windowing extension into _tkagg.
- PR #13146: Remove an outdated comment in backend_wx.

- PR #13126: FIX: minor log ticks overwrite
- PR #13148: Update example Step Demo
- PR #13138: API: Use class-based directive in sphinxext
- PR #11894: add `cache_frame_data` kwarg into `FuncAnimation`. fixes #8528.
- PR #13136: Small cleanups.
- PR #13140: Remove an "cannot show figure in agg" warning in test suite.
- PR #13134: Simplify color conversion backcompat shim.
- PR #13141: Unpin pytest (pytest-cov's latest release is compatible with it).
- PR #13133: Simplify the polys3d example.
- PR #12158: MNT: simplify valid tick logic
- PR #9867: Factor out common code between pdf and ps backends.
- PR #10111: Add set_data_3d and get_data_3d to Line3d
- PR #12245: Remove (some) features deprecated in mpl2.2
- PR #13119: Deprecate TextToPath.glyph_to_path.
- PR #13122: Pin pytest<4.1 to unbreak CI tests
- PR #13100: Restore the font cache on Travis.
- PR #12792: BUG: Ensure that distinct polygon collections are shaded identically
- PR #13070: cairo backend: default to pycairo
- PR #13114: BUG: calculate colorbar boundaries correctly from values
- PR #13111: Delete an unused private method.
- PR #10841: ENH: new date formatter
- PR #13093: Remove unused fontconfig conf file.
- PR #13063: Use default colour cycle in more examples
- PR #13103: Remove tight_bbox_test example.
- PR #13097: Replace 1-tuples by scalars where possible.
- PR #13027: Qt5 reset signals after non-interactive plotting
- PR #9787: Support (first font of) TTC files.
- PR #11780: ENH: Allow arbitrary coordinates for ConnectionPatch
- PR #12943: Update the font_table example.
- PR #13091: Improve MouseEvent str().
- PR #13095: Remove a duplicate attribute setting.
- PR #13090: Cleanup unused non-public imports.

- PR #13060: Move doc-requirements from root folder
- PR #13078: Convert streamplot to numpydoc
- PR #13088: Don't use deprecated np.random.random_integers.
- PR #13073: Drop pytest version check in setupext.py.
- PR #12933: Deprecate backend_pgf.LatexManagerFactory.
- PR #12969: Clarify the implementation of _process_plot_var_args.
- PR #12472: Make FontManager.defaultFont a property, to avoid hardcoding the prefix.
- PR #11806: Allow to not draw the labels on pie chart
- PR #11983: Simplify version checks for freetype and libpng.
- PR #13050: FIX: always eraseRect in Qt widget
- PR #13065: FIX: print out the correct ip address when starting webagg
- PR #13061: Make examples that load msft.csv robust against locale changes.
- PR #13042: cairo: remove the append_path() fast path
- PR #13058: pathlibify/cleanup triage_tests.py.
- PR #12995: Don't split creation of deprecation message and choice of warning class.
- PR #12998: Init MaxNLocator params only once
- PR #11691: Make Figure.frameon a thin wrapper for the patch visibility.
- PR #11735: Change {FigureCanvasAgg,RendererAgg}.buffer_rgba to return a memoryview.
- PR #12831: Reuse scale from sharing axis when calling cla().
- PR #12962: Deprecate setting the same property under two different aliases.
- PR #12973: Fix item check for pandas Series
- PR #13049: Add boxplot.flierprops.markeredgewidth rcParam
- PR #13048: Fix section names for numpydoc
- PR #10928: Simplify (quite a bit...) _preprocess_data
- PR #13039: Speed up Path.iter_segments()
- PR #12992: Adding rcParams['scatter.edgecolors'] defaulting to 'face'
- PR #13014: Drop pgi support for the GTK3 backend
- PR #12215: Cleanup initialization in text()
- PR #13029: Fix vertical alignment of text
- PR #12968: Simpler and stricter process_plot_format.

- PR #12989: Avoid spamming tests with warnings re: deprecation of pprint_val.
- PR #13032: fix typo in docstring in `axis_artist.py`
- PR #13025: MNT: add one more alias for tacaswell to mailmap
- PR #13010: Fix a format error in documenting_mpl.rst
- PR #12997: Add sphinx-copybutton to docs
- PR #12422: Scatter color: moving #10809 forward
- PR #12999: Format MaxNLocator with numpydoc
- PR #12991: Canonicalize weights extracted for AFM fonts.
- PR #12955: Cleanup cursor_demo.
- PR #12984: Cleanup GTK examples.
- PR #12986: Minor cleanup to double_pendulum example.
- PR #12959: Update the documentation of Cursor
- PR #12945: Correctly get weight & style hints from certain newer Microsoft fonts
- PR #12976: ENH: replace deprecated numpy header
- PR #12975: Fail-fast when trying to run tests with too-old pytest.
- PR #12970: Minor simplifications.
- PR #12974: Remove some checks for Py<3.6 in the test suite.
- PR #12779: Include scatter plots in Qt figure options editor.
- PR #12459: Improve formatting of imshow() cursor data when a colorbar exists.
- PR #12927: MAINT: Correctly handle empty lists in zip unpacking in mplot3d.art3d
- PR #12919: Suppress deprecation warning when testing drawstyle conflict
- PR #12956: Misc. cleanups.
- PR #12924: Deprecate public use of Formatter.pprint_val.
- PR #12947: Support ~ as nonbreaking space in mathtext.
- PR #12944: Fix the title of testing_api
- PR #12136: MAINT: Unify calculation of normal vectors from polygons
- PR #12880: More table documentation
- PR #12940: Avoid pyplot in showcase examples.
- PR #12935: os.PathLike exists on all supported Pythons now.
- PR #12936: Minor updates following bump to Py3.6+.
- PR #12932: Simplify argument checking in Table.__getitem__.

- PR #12930: Shorten an argument check.
- PR #12538: MNT: drop 3.5 testing for 3.1 branch
- PR #12868: Simplify use of Path._fast_from_codes_and_verts.
- PR #12300: API: Polar: allow flipped y/rlims....
- PR #12861: Don't use deprecated wx.NewId().
- PR #12908: Allow all valid hist.bins strings to be set in the rcparams
- PR #12902: Kill dead code in textpath.
- PR #12885: Improve margins in formlayout
- PR #12877: fooImage -> foo_image in testing/compare.py
- PR #12845: Deprecate silent dropping of unknown arguments to TextPath().
- PR #12852: Cleanup collections docs.
- PR #12888: Properly enable forward/backward buttons on GTK3
- PR #12865: Avoid 1-tick or 0-tick log-scaled axis.
- PR #12844: Remove unused, private _process_text_args.
- PR #12881: Fix string comparison
- PR #12863: FIX: translate timedeltas in _to_ordinalf
- PR #12640: Introduce MouseButton enum for MouseEvent.
- PR #12897: Reword a bit the contour docs.
- PR #12898: Validate rcParams["image.origin"].
- PR #12882: Write error messages to logger instead of stderr
- PR #12889: Deprecate public access to the vendored formlayout module.
- PR #12891: Add Azure Pipelines build badge
- PR #12883: MAINT Use list comprehension
- PR #12886: Properly enable forward/backward buttons on Qt
- PR #12858: Bump oldest supported numpy to 1.11.
- PR #12876: Fix a typo
- PR #12739: make Axes._parse_scatter_color_args static
- PR #12846: Deprecate Path.has_nonfinite.
- PR #12829: Remove unused variables
- PR #12872: Inline references to RendererPS in backend_ps.
- PR #12800: documenting dtype of hist counts
- PR #12842: Fix message in nbagg connection_info()

- PR #12855: Cleanup axes/_base.py.
- PR #12826: Minor code cleanup
- PR #12866: Simplify stride calculations in loglocator.
- PR #12867: Drop compat code for outdated MSVC.
- PR #12218: Improve table docs
- PR #12847: correctly format ticklabels when EngFormatter is used with usetex = True
- PR #12851: Keep Collections and Patches property aliases in sync.
- PR #12849: Update docstrings in path.py, and small cleanups.
- PR #12805: Don't insert spurious newlines by joining tex.preamble.
- PR #12827: Remove unused imports
- PR #12560: Add matplotlib.testing to the documentation
- PR #12821: MNT: remove debug from update_title_pos
- PR #12764: Cleanup Renderer/GraphicsContext docs.
- PR #12759: Warn on FreeType missing glyphs.
- PR #12799: Reword some colorbar docs.
- PR #12633: Added support for MacOSX backend for PyPy
- PR #12798: Replace assignments to array.shape by calls to reshape().
- PR #11851: Simpler check for whether a Framework Python build is being used.
- PR #12259: BUG: Fix face orientations of bar3d
- PR #12565: Make FontManager.score_weight less lenient.
- PR #12674: Allow "real" LaTeX code for pgf.preamble in matplotlibrc
- PR #12770: Simplify implementation of FontProperties.copy().
- PR #12753: MNT: remove _hold shims to support basemap + cartopy
- PR #12450: Attach a FigureCanvasBase by default to Figures.
- PR #12643: Allow unit input to FancyArrowPatch
- PR #12767: Make colorbars constructible with dataless ScalarMappables.
- PR #12526: Rename jquery files
- PR #12552: Update docs for writing image comparison tests.
- PR #12746: Use skipif, not xfail, for uncomparable image formats.
- PR #12747: Prefer log.warning("%s", ...) to log.warning("%s" % ...).
- PR #11753: FIX: Apply aspect before drawing starts

- PR #12749: Move toolmanager warning from logging to warning.
- PR #12598: Support Cn colors with n>=10.
- PR #12727: Reorder API docs: separate file per module
- PR #12738: Add unobtrusive depreaction note to the first line of the docstring.
- PR #11663: Refactor color parsing of Axes.scatter
- PR #12736: Move deprecation note to end of docstring
- PR #12704: Rename tkinter import from Tk to tk.
- PR #12715: Cleanup dviread.
- PR #12717: Delete some `if __name__ == "__main__"` clauses.
- PR #10575: FIX patch.update_from to also copy _original_edge/facecolor
- PR #12537: Improve error message on failing test_pyplot_up_to_date
- PR #12721: Make get_scale_docs() internal
- PR #12706: Extend sphinx Makefile to cleanup completely
- PR #12481: Warn if plot_surface Z values contain NaN
- PR #12685: Make ticks in demo_axes_rgb.py visible
- PR #12523: Run flake8 before pytest on travis
- PR #12691: DOC: Link to "How to make a PR" tutorials as badge and in contributing
- PR #11974: Make code match comment in sankey.
- PR #12440: Make arguments to @deprecated/warn_deprecated keyword-only.
- PR #12470: Update AutoDateFormatter with locator
- PR #12586: Improve linestyles example
- PR #12006: Replace warnings.warn with cbook._warn_external or logging.warning
- PR #12659: Add note that developer discussions are private
- PR #12543: Make rcsetup.py flak8 compliant
- PR #12642: Don't silence TypeErrors in fmt_{x,y}data.
- PR #12442: Deprecate passing drawstyle with linestyle as single string.
- PR #12625: Shorten some docstrings.
- PR #12627: Be a bit more stringent on invalid inputs.
- PR #12629: Fix issue with PyPy on macOS
- PR #10933: Remove "experimental" fontconfig font_manager backend.
- PR #12600: Minor style fixes.

- PR #12570: Fix mathtext tutorial for build with Sphinx 1.8.
- PR #12487: Update docs/tests for the deprecation of aname and label1On/label2On/etc.
- PR #12521: Improve docstring of draw_idle()
- PR #12574: Remove some unused imports
- PR #12568: Add note regarding builds of old Matplotlibs.
- PR #12547: Disable sticky edge accumulation if no autoscaling.
- PR #12546: Avoid quadratic behavior when accumulating stickies.
- PR #11789: endless looping GIFs with PillowWriter
- PR #12525: Fix some flake8 issues
- PR #12516: Don't handle impossible values for `align` in hist()
- PR #12500: Adjust the widths of the messages during the build.
- PR #12492: Simplify radar_chart example.
- PR #11984: Strip out pkg-config machinery for agg and libqhull.
- PR #12463: Document Artist.cursor_data() parameter
- PR #12482: Test slider orientation
- PR #12317: Always install mpl_toolkits.
- PR #12246: Be less tolerant of broken installs.
- PR #12477: Use N{MICRO SIGN} instead of N{GREEK SMALL LETTER MU} in EngFormatter.
- PR #12483: Kill FontManager.update_fonts.
- PR #12474: Throw ValueError when irregularly gridded data is passed to streamplot.
- PR #12466: np.fromstring -> np.frombuffer.
- PR #12369: Improved exception handling on animation failure
- PR #12460: Deprecate RendererBase.strip_math.
- PR #12453: Rollback erroneous commit to whats_new.rst from #10746
- PR #12452: Minor updates to the FAQ.
- PR #10746: Adjusted matplotlib.widgets.Slider to have optional vertical orientatation
- PR #12441: Get rid of a signed-compare warning.
- PR #12430: Deprecate Axes3D.plot_surface(shade=None)
- PR #12435: Fix numpydoc parameter formatting

- PR #12434: Clarify documentation for textprops keyword parameter of TextArea
- PR #12427: Document Artist.get_cursor_data
- PR #10322: Use np.hypot wherever possible.
- PR #10809: Fix for scatter not showing points with valid x/y but invalid color
- PR #12423: Minor simplifications to backend_svg.
- PR #10356: fix detecting which artist(s) the mouse is over
- PR #10268: Dvi caching
- PR #10238: Call kpsewhich with more arguments at one time
- PR #10236: Cache kpsewhich results persistently
- PR #4675: Deprecate color keyword argument in scatter
- PR #5054: Diverging norm
- PR #12416: Move font cache rebuild out of exception handler
- PR #4762: Traitlets
- PR #5414: WIP: New FreeType wrappers
- PR #3875: ENH: passing colors (and other optional keyword arguments) to violinplot()
- PR #1959: PS backend optionally jpeg-compresses the embedded images
- PR #11891: Group some print()s in backend_ps.
- PR #12165: Remove deprecated mlab code
- PR #12387: Update HTML animation as slider is dragged
- PR #12333: ENH: add colorbar method to axes
- PR #10088: Deprecate Tick.{gridOn,tick1On,label1On,...} in favor of set_visible.
- PR #12393: Deprecate to-days converters in matplotlib dates
- PR #11232: FIX: fix figure.set_dpi when pixel ratio not 1
- PR #12247: Machinery for deprecating properties.
- PR #12371: Move check for ImageMagick Windows path to bin_path().
- PR #12384: Cleanup axislines style.
- PR #9565: Stem performance boost
- PR #12368: Don't use stdlib private API in animation.py.
- PR #12351: dviread: find_tex_file: Ensure the encoding on windows
- PR #12372: Remove two examples.
- PR #12356: Fix stripping of CRLF on Windows.

- PR #12283: FIX: errorbar xywhere should return ndarray
- PR #12304: TST: Merge Qt tests into one file.
- PR #12340: Catch test deprecation warnings for mlab.demean
- PR #12296: Make FooConverter inherit from ConversionInterface in examples
- PR #12309: Deduplicate implementations of FooNorm.autoscale{,_None}
- PR #7716: [NF] Add 'truncate' and 'join' methods to colormaps.
- PR #12314: Deprecate `axis('normal')` in favor of `axis('auto')`.
- PR #12307: Clarify missing-property error message.
- PR #12260: Fix docs : change from issue #12191, remove "if 1:" blocks in examples
- PR #12253: Handle utf-8 output by kpathsea on Windows.
- PR #12292: TST: Modify the bar3d test to show three more angles
- PR #12284: Don't try to autoscale if no data present to autoscale to
- PR #12255: Deduplicate inherited docstrings.
- PR #12222: Remove extraneous if 1 statements in demo_axisline_style.py
- PR #12137: MAINT: Vectorize bar3d
- PR #12219: Merge OSXInstalledFonts into findSystemFonts.
- PR #12229: Less ACCEPTS, more numpydoc.
- PR #11621: TST: make E402 a universal flake8 ignore
- PR #12231: CI: Speed up Appveyor repository cloning
- PR #11661: Update blocking_input.py
- PR #12199: Allow disabling specific mouse actions in blocking_input
- PR #12210: Axes.tick_params() argument checking
- PR #12211: Fix typo
- PR #12200: Slightly clarify some invalid shape exceptions for image data.
- PR #12151: Don't pretend @deprecated applies to classmethods.
- PR #12190: Remove some unused variables and imports
- PR #12192: Exclude examples from lgtm analysis
- PR #12196: Give Carreau the ability to mention the backport bot.
- PR #12171: Remove internal warning due to zsort deprecation
- PR #12030: Speed up canvas redraw for GTK3Agg backend.
- PR #12156: Cleanup the GridSpec demos.

- PR #12144: Add explicit getters and setters for Annotation.anncoords.
- PR #12152: Use _warn_external for deprecations warnings.
- PR #12147: DOC: update the gh_stats code
- PR #12139: Unbreak build re: mplot3d style.
- PR #11367: Raise TypeError on unsupported kwargs of spy()
- PR #9990: Fix and document lightsource argument in mplot3d
- PR #12124: Correctly infer units from empty arrays
- PR #11994: Cleanup unused variables and imports
- PR #12122: MNT: re-add cbook import art3d
- PR #12086: FIX: make MaxNLocator only follow visible ticks for order of magnitude
- PR #12032: Remove unused imports
- PR #12093: Correct the removal of -Wstrict-prototypes from compiler flags.
- PR #12069: Style fixes for mplot3d.
- PR #11997: Cleanup some axes_grid1 examples
- PR #12098: Improve layout of HTML animation
- PR #12094: Fine-tune logging notes in contributing.rst.
- PR #12079: Clarifications to **im_show()** doc regarding *interpolation*='*none*'.
- PR #12068: More style fixes.
- PR #11499: FIX: layout for mixed descent multiline text objects
- PR #11921: FIX: allow reshape 2-D to return a bare 1-d list
- PR #12070: Avoid some uses of np.isscalar.
- PR #12067: DOC: make Line2D docstring definition easier to find
- PR #12054: More style fixes.
- PR #12066: fix indentation in docstring interpolation for spy.
- PR #11931: Remove separate autosummary_inher template.
- PR #12049: Make Poly3DCollection.set_zsort less lenient.
- PR #12050: Various cleanups.
- PR #12038: Modernize ArtistInspector a bit...
- PR #12033: DOC: formatting fixes to mplot3d
- PR #12051: Is bool
- PR #12045: Fix 999.9... edge case in ticker.EngFormatter for negative numbers

- PR #12044: Update doc on the *progressive* and *optimize* keywords in savefig
- PR #12061: Small refactor/simplification.
- PR #12060: INSTALL.rst fixes
- PR #12055: Fix invalid escape in docstring.
- PR #12026: whitespace(-mostly) style cleanup.
- PR #12043: Deprecate get_py2exe_datafiles.
- PR #12046: Make HTMLWriter constructor a bit more strict.
- PR #12034: Doc markup fixes.
- PR #11972: FIX: close mem leak for repeated draw
- PR #12024: Fix typos
- PR #11996: Minor javascript cleanup
- PR #11989: Remove support for ghostscript 8.60.
- PR #12004: Update acorr and xcorr docs to match numpy docs
- PR #11998: No clf() needed after creating a figure
- PR #12001: Do not use an explicit figum in plt.figure(1, ...) in simple cases
- PR #11999: Do not use an explicit fignum plt.figure(1) in simple cases
- PR #11995: Don't use bare except statements
- PR #11993: DOC: fixed typos
- PR #11992: Use pytest.warns instead of home-baked warnings capture.
- PR #11975: Derive plt.figlegend.__doc__ from Figure.legend.__doc__.
- PR #11980: Remove __version__numpy__; simplify dependencies check.
- PR #11982: Remove and old keyword documentation.
- PR #11981: Some extra typos
- PR #11979: Fix a couple of typos.
- PR #11959: cbook.iterable -> np.iterable.
- PR #11965: Move the removal of the -Wstrict-prototypes flag to setup.py.
- PR #11958: Remove unused code
- PR #11960: Make jpl_units a bit less painful to read.
- PR #11951: Improve Artist docstrings
- PR #11954: No need to define _log twice in matplotlib.dates.
- PR #11948: Minor fixes to docs and gitignore.
- PR #11777: Avoid incorrect warning in savefig

- PR #11942: Deprecate Artist.aname and Axes.aname
- PR #11935: Remove ginput demo example
- PR #11939: Improve alias signatures
- PR #11940: Do not use aliases of properties in internal code
- PR #11941: Fix test_large_subscript_title()
- PR #11938: More docstring cleanup of Line2D.
- PR #11920: Add LGTM.com code quality badge
- PR #11922: Improve docstrings of Line2D
- PR #11924: Minor formatting update on alias docstrings
- PR #11926: Minor fix to ginput_demo.
- PR #11912: BLD: update PR template for flake8
- PR #11909: Simplify linestyle and fillstyle reference docs.
- PR #11502: FIX: move title(s) up if subscripts hang too low.
- PR #11906: fix format of bar_of_pie example
- PR #11741: Factor out common code between Patch.draw and FancyArrowPatch.draw.
- PR #11784: Argument checking for grid()
- PR #11888: Factor out a subprocess log-and-check helper.
- PR #11740: Deprecate support for 3rd-party backends without set_hatch_color.
- PR #11884: Deprecate the tk_window_focus function.
- PR #11689: Don't cache the renderer on the Axes instance.
- PR #11698: For property, use decorator or lambdas.
- PR #11872: Make all builtin cmaps picklable.
- PR #11870: More style fixes.
- PR #11873: Remove mention of deprecated/removed methods from mlab's docstring.
- PR #11869: Style fixes.
- PR #11874: Remove some remnants of Py2-handling in test_rcparams.
- PR #11865: example file for making a bar of pie chart
- PR #11868: mathtext.py style fixes.
- PR #11854: Accept anything that's not a directory for $MATPLOTLIBRC.
- PR #11589: WIP ENH secondary axes:
- PR #8449: Including Additional Metadata using the SVG Backend

- PR #11465: ENH: optimize Collection non-affine transform to call transform once

Issues (161):

- #4001: Qt5 Backend: dblclick is always False on 'mouse_release_event'
- #14152: qt_compat.py performing wrong test for PyQt5
- #10875: Annotation.contains and FancyArrow.contains return incorrect values
- #458: JPG quality keyword in savefig
- #4354: scatter not showing valid x/y points with invalid color
- #14113: scatter could not raise when colors are provided but position data are empty
- #14003: numpydoc 0.9 breaks doc build
- #14054: ticks sometimes disappear when zooming interactively
- #10189: The data decorator does not integrate well with numpydoc
- #14034: pyplot plot raises ValueError when plotting NaN against datetime dates
- #14039: bar plot yerr lines/caps should respect zorder
- #14042: dynamic_image.py + saving animation broken
- #14013: osx backend not usable with ipython/jupyter from conda?
- #13993: Tests files installed by default?
- #13991: MaxNLocator.default_params deprecation may break Cartopy
- #5045: Axes.grid() not honoring specified "zorder" kwarg
- #4371: LaTeX and PGF preambles do not allow commas
- #13982: hist() no longer respects range=... when density=True
- #13963: Dataless colorbars break when updated
- #10381: Issue when setting scatter color in separate method call
- #13618: Minor ticklabels are missing at positions of major ticks.
- #13880: Adding documentation for Text.fontfamily default, set_fontfamily(None)?
- #13865: Appveyor broken
- #8636: plt.hist chooses improper range when using string-based bin options
- #7300: weird mathtext doc markup
- #8862: Replace mathcircled by textcircled
- #13759: DOC: matplotlib.patches.Arc
- #13785: Imshow gives values out of the extent

- #13786: Cannot import matplotlib.animation
- #13561: Progress of animation.save (for long animations)
- #13735: title doesn't move for ticklables....
- #12175: Example link near markevery in the "What's new in 3.0" page is malformed/broken
- #13713: Boxplot xlim not correctly calculated
- #11070: Add a "density" kwarg to hist2d
- #11337: Cannot plot fully masked array against datetimes
- #10165: Adapt stem plot
- #10976: ENH: secondary axis for a x or y scale.
- #10763: Cairo in 2.2.0 not working for new backends
- #9737: setupext should not explicitly add /usr/{,local/}include to the include path
- #11217: Crash on zero-length FancyArrow
- #13623: do not cause warning in seaborn
- #13480: Segfault on help('modules') command when matplotlib is installed
- #13604: legend's framealpha kwarg does not apply when writing to an eps file
- #12311: 'off' vs. False bug
- #10237: Setting an alpha value to a Poly3DCollection
- #11781: fill_between interpolation & nan issue
- #1077: 3d plots with aspect='equal'
- #11761: Still naming inconsistency in API on axes limits
- #11623: Regression: "TypeError: Period('2000-12-31', 'D') is not a string" when a Series with date index was plotted
- #12655: auto-ticks do not handle values near bounds gracefully
- #13487: labelpad is not the spacing between the axis and the label
- #13540: Docs for matplotlib.pyplot.specgram() reference an unsupported mode setting
- #8997: Proposal: Grid arrangement by number of plots
- #6928: Cannot run `setup.py build` with numpy master
- #12697: Axes are drawn at wrong positions
- #13478: FuncAnimation: interactive zoom/pan with blitting does not work
- #11575: Setting axis ticks in log scale produces duplicate tick labels.

- #13464: set_rlim(bottom=...) no longer works
- #12628: Write canonical example of how to use Matplotlib inside a webserver
- #10022: boxplot: positions used to take Int64Index
- #11647: Disable buttons in ginput
- #12987: issues parsing AFM fonts
- #12667: Colorbar ticks....
- #13137: Travis for Python 3.7 sometimes fails due to missing font
- #7969: Stem is slow and will crash if I try to close the window
- #13002: Hist color kwarg broken for multiple empty datasets
- #5581: [mpl_toolkits] Things drawn on parasite axes don't fire pick events
- #13417: Secondary axis doesn't resize properly
- #8120: Inconsistent inset_axes position between show(), savefig(format='png') and savefig(format='pdf')
- #8947: Different result, slower runtime of heatmap between 2.0.0 and 2.0.1
- #13264: Use of logging in matplotlib
- #11602: animation error
- #12925: Python pandas datetime plot xticks in unexpected location
- #11025: AxesGrid ticks missing on x-axis
- #10974: Examples not shown in API docs for many methods.
- #13392: boxplot broken for empty inputs
- #12345: Need more tests for units and errorbar
- #10361: FigureCanvas.draw() with tight_layout () needs to be called twice with Matplotlib 2.1.0
- #11376: Temporary styling ignores color cycle
- #11546: import time
- #13286: AttributeError: 'float' object has no attribute 'deg2rad'
- #11508: bi-directional perceptually flat colormaps in matplotlib?
- #12918: Mac shows an icon in the dock when using matplotlib.pyplot.
- #13339: Log Colorbar minorticks_off reverted if ticks set...
- #13228: MPL 3 + Colorbar + PowerNorm bug
- #13096: Matplotlib.get_backend()/matplotlib.use() cause NSException with Anaconda
- #7712: Number of ticks for dates still gives overlapping labels

- #9978: General poor default formatting of datetimes on plot x-axis
- #13253: imsave outputs JPEG with wrong dimension
- #11391: Use data argument for scatter plotting timestamps from pandas
- #13145: widgets.RadioButtons: select by closest in position
- #13267: "double-pendulum" example's speed not correct / varying
- #13257: Allow turning off minorticks for Colorbar with LogNorm?
- #13237: Sankey basic gallery example is not rendered properly.
- #12836: matplotlib.rc_file resets to default template before updating rcparams
- #13186: ax.bar throws when x axis is pandas datetime
- #5397: Expose compression and filter PNG options through savefig
- #13142: Cannot plot bar graph with dates: "TypeError: ufunc subtract cannot use operands with types dtype('<M8[ns]') and dtype('float64')"
- #8530: Feature request: TIFF LZW compression support in savefig()
- #13139: font family ['serif'] not found. Falling back to DejaVu Sans
- #1558: Graceful handling of a numpy matrix
- #12954: Fonts installed in the user directory are not detected (Windows 1809)
- #3644: Feature Request: manually set colorbar without mappable
- #12862: broken_barh appears not to work with datetime/timedelta objects
- #11290: ax.bar doesn't work correctly when width is a timedelta64 object
- #13156: DOC: matplotlib.pyplot.arrow
- #12990: Unclear error message for `plt.xticks(names)`
- #12769: Failing to save an animated graph with matplotlib.animation
- #13112: LogNorm colorbar prints double tick labels after set_ticks()
- #13132: BUG: matplotlib.sphinxext.plot_directive uses old function-based API
- #8528: Funcanimation memory leak?
- #8914: line3D set_data only takes in x and y data
- #8768: One one tick in a log-scale axis
- #13121: Tests fail with pytest 4.1
- #13098: Likely incorrect code(?) in colorbar.py
- #12562: Clean up unused imports
- #12106: plt.plot does not plot anything with named arguments
- #5145: Python [Error 17]No usable Temporary file name found

- #13012: qt5agg image quality changes when window is out of focus
- #13055: 127.0.0.1 hardcoded in webagg backend server
- #12971: Pandas Series not supported as data kwarg
- #13022: boxplot not showing symbols with seaborn style sheet
- #13028: Bad rotation_mode/center_baseline combination even if rotation=0
- #12745: Sphinx copy button for code block
- #12801: scatter() should not drop data points at nonfinite coordinates
- #12358: Dropping support for Py3.5 and numpy 1.10
- #12994: Axes range with set_xticks with categoricals
- #12993: Semantics of set_xticks for categoricals
- #12946: ~ in mathrm leads to Unknown symbol: mathrm
- #10704: Add documentation for set_rlim
- #11202: Using of ax.set_ylim() for polar plot leads to "posx and posy should be finite values" error
- #12859: DeprecationWarning: NewId() is deprecated in wxPython.
- #12817: Multiple places where Type Errors on cbook.warn_deprecated will happen
- #12308: #12253 FIX: Handle utf-8 output by kpathsea on Windows -- possibly causing issues
- #12804: Usetex produces preamble with one character per line
- #12808: Issue with minor tick spacing in colorbar with custom Normalize class
- #12138: Faces of Axes3d.bar3d are not oriented correctly
- #12591: Adding FancyArrowPatch with datetime coordinates fails
- #11139: "make clean" doesn't remove all the build doc files
- #11908: Improve linestyle documentation
- #10643: Most warnings calls do not set the stacklevel
- #12532: Incorrect rendering of math symbols
- #11787: Looping gifs with PillowWriter
- #9205: after the animation encoder (e.g. ffmpeg) fails, the animation framework itself fails internally in various ways while trying to report the error
- #11154: Unexpected behavior for Axes3D.plot_surface(shade=None)
- #12121: Documentation of TextArea's fontprops keyword argument is misleading
- #12191: "if 1:" blocks in examples

- #12107: warnings re: deprecated pytest API with pytest 3.8
- #12010: Popover over plot is very slow
- #12118: Scatter: empty np.arrays with non-numeric dtypes cause TypeError
- #12072: `MaxNLocator` changes the scientific notation exponent with different number of tick labels
- #11795: Un-align animations created with to_jshtml()?
- #10201: Available fonts are ignored by font_manager
- #12065: Keyword *interpolation* behaving improperly while saving to SVG with **savefig()**
- #11498: Test layout with big descenders and multiple lines inconsistent.
- #11468: Layout managers have problems with titles containing MathText
- #11899: Histogram of list of datetimes
- #11956: apparent memory leak with live plotting
- #11587: Missing filled contours when using contourf
- #11716: errorbar pickling fails when specifying y error bars
- #11557: Hoping add a drawing function 'patch' in matplotlib

7.1.10 GitHub Stats for Matplotlib 3.0.2

GitHub stats for 2018/09/18 - 2018/11/09 (tag: v3.0.0)

These lists are automatically generated, and may be incomplete or contain duplicates.

We closed 170 issues and merged 224 pull requests.

The following 49 authors contributed 460 commits.

- Abhinuv Nitin Pitale
- Alon Hershenhorn
- Andras Deak
- Ankur Dedania
- Antony Lee
- Anubhav Shrimal
- Ayappan P
- azure-pipelines[bot]
- Ben Root
- Colin

- Colin Carroll
- Daniele Nicolodi
- David Haberthür
- David Stansby
- Dmitry Mottl
- Elan Ernest
- Elliott Sales de Andrade
- Eric Wieser
- esvhd
- Galen Lynch
- hannah
- Ildar Akhmetgaleev
- ImportanceOfBeingErnest
- Jody Klymak
- Joel Wanner
- Kai Muehlbauer
- Kevin Rose
- Kyle Sunden
- Marcel Martin
- Matthias Bussonnier
- MeeseeksMachine
- Michael Jancsy
- Nelle Varoquaux
- Nick Papior
- Nikita Kniazev
- Paul Hobson
- pharshalp
- Rasmus Diederichsen
- Ryan May
- saksmito
- Takafumi Arakaki
- teresy

- Thomas A Caswell
- thoo
- Tim Hoffmann
- Tobias Megies
- Tyler Makaro
- Will Handley
- Yuxin Wu

GitHub issues and pull requests:

Pull Requests (224):

- PR #12785: Use level kwargs in irregular contour example
- PR #12767: Make colorbars constructible with dataless ScalarMappables.
- PR #12775: Add note to errorbar function about sign of errors
- PR #12776: Fix typo in example (on-borad -> on-board).
- PR #12771: Do not rely on external stack frame to exist
- PR #12526: Rename jquery files
- PR #12552: Update docs for writing image comparison tests.
- PR #12746: Use skipif, not xfail, for uncomparable image formats.
- PR #12747: Prefer log.warning("%s", ...) to log.warning("%s" % ...).
- PR #11753: FIX: Apply aspect before drawing starts
- PR #12749: Move toolmanager warning from logging to warning.
- PR #12708: Run flake8 in a separate travis environment
- PR #12737: Improve docstring of Arc
- PR #12598: Support Cn colors with n>=10.
- PR #12670: FIX: add setter for hold to un-break basemap
- PR #12693: Workaround Text3D breaking tight_layout()
- PR #12727: Reorder API docs: separate file per module
- PR #12738: Add unobtrusive depreaction note to the first line of the docstring.
- PR #12740: DOC: constrained layout guide (fix: Spacing with colorbars)
- PR #11663: Refactor color parsing of Axes.scatter
- PR #12736: Move deprecation note to end of docstring
- PR #12704: Rename tkinter import from Tk to tk.
- PR #12730: MNT: merge ignore lines in .flake8

- PR #12707: Fix tk error when closing first pyplot figure
- PR #12715: Cleanup dviread.
- PR #12717: Delete some `if __name__ == "__main__"` clauses.
- PR #12726: Fix test_non_gui_warning for Azure (and mplcairo).
- PR #12720: Improve docs on Axes scales
- PR #12537: Improve error message on failing test_pyplot_up_to_date
- PR #12721: Make get_scale_docs() internal
- PR #12617: Set up CI with Azure Pipelines
- PR #12673: Fix for _axes.scatter() array index out of bound error
- PR #12676: Doc: document textpath module
- PR #12705: Improve docs on Axes limits and direction
- PR #12706: Extend sphinx Makefile to cleanup completely
- PR #12481: Warn if plot_surface Z values contain NaN
- PR #12709: Correctly remove nans when drawing paths with pycairo.
- PR #12685: Make ticks in demo_axes_rgb.py visible
- PR #12691: DOC: Link to "How to make a PR" tutorials as badge and in contributing
- PR #12684: Change ipython block to code-block
- PR #11974: Make code match comment in sankey.
- PR #12440: Make arguments to @deprecated/warn_deprecated keyword-only.
- PR #12683: TST: mark test_constrainedlayout.py::test_colorbar_location as flaky
- PR #12686: Remove deprecation warnings in tests
- PR #12470: Update AutoDateFormatter with locator
- PR #12656: FIX: fix error in colorbar.get_ticks not having valid data
- PR #12586: Improve linestyles example
- PR #12006: Added stacklevel=2 to all warnings.warn calls (issue 10643)
- PR #12651: FIX: ignore non-finite bbox
- PR #12653: Don't warn when accessing deprecated properties from the class.
- PR #12608: ENH: allow matplotlib.use after getbackend
- PR #12658: Do not warn-depreacted when iterating over rcParams
- PR #12635: FIX: allow non bbox_extra_artists calls
- PR #12659: Add note that developer discussions are private

- PR #12543: Make rcsetup.py flak8 compliant
- PR #12642: Don't silence TypeErrors in fmt_{x,y}data.
- PR #11667: DOC: update doc requirement
- PR #12442: Deprecate passing drawstyle with linestyle as single string.
- PR #12625: Shorten some docstrings.
- PR #12627: Be a bit more stringent on invalid inputs.
- PR #12561: Properly css-style exceptions in the documentation
- PR #12629: Fix issue with PyPy on macOS
- PR #10933: Remove "experimental" fontconfig font_manager backend.
- PR #12630: Fix RcParams.__len__
- PR #12285: FIX: Don't apply tight_layout if axes collapse
- PR #12548: undef _XOPEN_SOURCE breaks the build in AIX
- PR #12615: Fix travis OSX build
- PR #12600: Minor style fixes.
- PR #12607: STY: fix whitespace and escaping
- PR #12603: FIX: don't import macosx to check if eventloop running
- PR #12599: Fix formatting of docstring
- PR #12569: Don't confuse uintptr_t and Py_ssize_t.
- PR #12572: Fix singleton hist labels
- PR #12581: Fix hist() error message
- PR #12570: Fix mathtext tutorial for build with Sphinx 1.8.
- PR #12487: Update docs/tests for the deprecation of aname and label1On/label2On/etc.
- PR #12521: Improve docstring of draw_idle()
- PR #12573: BUG: mplot3d: Don't crash if azim or elev are non-integral
- PR #12574: Remove some unused imports
- PR #12568: Add note regarding builds of old Matplotlibs.
- PR #12555: Clarify horizontalalignment and verticalalignment in suptitle
- PR #12547: Disable sticky edge accumulation if no autoscaling.
- PR #12546: Avoid quadratic behavior when accumulating stickies.
- PR #12159: FIX: colorbar re-check norm before draw for autolabels
- PR #12501: Rectified plot error

- PR #11789: endless looping GIFs with PillowWriter
- PR #12525: Fix some flake8 issues
- PR #12431: FIX: allow single-string color for scatter
- PR #12216: Doc: Fix search for sphinx >=1.8
- PR #12461: FIX: make add_lines work with new colorbar
- PR #12241: FIX: make unused spines invisible
- PR #12516: Don't handle impossible values for `align` in hist()
- PR #12504: DOC: clarify min supported version wording
- PR #12507: FIX: make minor ticks formatted with science formatter as well
- PR #12500: Adjust the widths of the messages during the build.
- PR #12492: Simplify radar_chart example.
- PR #12478: MAINT: NumPy deprecates asscalar in 1.16
- PR #12363: FIX: errors in get_position changes
- PR #12495: Fix duplicate condition in pathpatch3d example
- PR #11984: Strip out pkg-config machinery for agg and libqhull.
- PR #12463: Document Artist.cursor_data() parameter
- PR #12489: Fix typo in documentation of ylim
- PR #12482: Test slider orientation
- PR #12317: Always install mpl_toolkits.
- PR #12246: Be less tolerant of broken installs.
- PR #12477: Use N{MICRO SIGN} instead of N{GREEK SMALL LETTER MU} in EngFormatter.
- PR #12483: Kill FontManager.update_fonts.
- PR #12448: Don't error if some font directories are not readable.
- PR #12474: Throw ValueError when irregularly gridded data is passed to streamplot.
- PR #12469: Clarify documentation of offsetbox.AnchoredText's prop kw argument
- PR #12468: Fix `set_ylim` unit handling
- PR #12466: np.fromstring -> np.frombuffer.
- PR #12369: Improved exception handling on animation failure
- PR #12460: Deprecate RendererBase.strip_math.
- PR #12457: Fix tutorial typos.

- PR #12453: Rollback erroneous commit to whats_new.rst from #10746
- PR #12452: Minor updates to the FAQ.
- PR #10746: Adjusted matplotlib.widgets.Slider to have optional vertical orientation
- PR #12441: Get rid of a signed-compare warning.
- PR #12430: Deprecate Axes3D.plot_surface(shade=None)
- PR #12435: Fix numpydoc parameter formatting
- PR #12434: Clarify documentation for textprops keyword parameter of TextArea
- PR #12427: Document Artist.get_cursor_data
- PR #12277: FIX: datetime64 now recognized if in a list
- PR #10322: Use np.hypot wherever possible.
- PR #12423: Minor simplifications to backend_svg.
- PR #12293: Make pyplot more tolerant wrt. 3rd-party subclasses.
- PR #12360: Replace axes_grid by axes_grid1 in test
- PR #10356: fix detecting which artist(s) the mouse is over
- PR #12416: Move font cache rebuild out of exception handler
- PR #11891: Group some print()s in backend_ps.
- PR #12165: Remove deprecated mlab code
- PR #12394: DOC: fix CL tutorial to give same output from saved file and example
- PR #12387: Update HTML animation as slider is dragged
- PR #12408: Don't crash on invalid registry font entries on Windows.
- PR #10088: Deprecate Tick.{gridOn,tick1On,label1On,...} in favor of set_visible.
- PR #12149: Mathtext tutorial fixes
- PR #12393: Deprecate to-days converters in matplotlib dates
- PR #12257: Document standard backends in matplotlib.use()
- PR #12383: Revert change of parameter name in annotate()
- PR #12385: CI: Added Appveyor Python 3.7 build
- PR #12247: Machinery for deprecating properties.
- PR #12371: Move check for ImageMagick Windows path to bin_path().
- PR #12384: Cleanup axislines style.
- PR #12353: Doc: clarify default parameters in scatter docs
- PR #12366: TST: Update test images for new Ghostscript.

- PR #11648: FIX: colorbar placement in constrained layout
- PR #12368: Don't use stdlib private API in animation.py.
- PR #12351: dviread: find_tex_file: Ensure the encoding on windows
- PR #12244: Merge barchart examples.
- PR #12372: Remove two examples.
- PR #12214: Improve docstring of Annotation
- PR #12347: DOC: add_child_axes to axes_api.rst
- PR #12304: TST: Merge Qt tests into one file.
- PR #12321: maint: setupext.py for freetype had a Catch case for missing ft2build.h
- PR #12340: Catch test deprecation warnings for mlab.demean
- PR #12334: Improve selection of inset indicator connectors.
- PR #12316: Fix some warnings from Travis
- PR #12268: FIX: remove unnecessary `self` in `super_`-calls, fixes #12265
- PR #12212: font_manager: Fixed problems with Path(...).suffix
- PR #12326: fixed minor spelling error in docstring
- PR #12296: Make FooConverter inherit from ConversionInterface in examples
- PR #12322: Fix the docs build.
- PR #12319: Fix Travis 3.6 builds
- PR #12309: Deduplicate implementations of FooNorm.autoscale{,_None}
- PR #12314: Deprecate `axis('normal')` in favor of `axis('auto')`.
- PR #12313: BUG: Fix typo in view_limits() for MultipleLocator
- PR #12307: Clarify missing-property error message.
- PR #12274: MNT: put back `_hold` as read-only attribute on AxesBase
- PR #12260: Fix docs : change from issue #12191, remove "if 1:" blocks in examples
- PR #12163: TST: Defer loading Qt framework until test is run.
- PR #12253: Handle utf-8 output by kpathsea on Windows.
- PR #12301: Ghostscript 9.0 requirement revisited
- PR #12294: Fix expand_dims warnings in triinterpolate
- PR #12292: TST: Modify the bar3d test to show three more angles
- PR #12297: Remove some pytest parameterising warnings
- PR #12261: FIX: parasite axis2 demo

- PR #12278: Document inheriting docstrings
- PR #12262: Simplify empty-rasterized pdf test.
- PR #12269: Add some param docs to BlockingInput methods
- PR #12272: Fix `contrained` to `constrained`
- PR #12255: Deduplicate inherited docstrings.
- PR #12254: Improve docstrings of Animations
- PR #12258: Fix CSS for module-level data
- PR #12222: Remove extraneous if 1 statements in demo_axisline_style.py
- PR #12137: MAINT: Vectorize bar3d
- PR #12219: Merge OSXInstalledFonts into findSystemFonts.
- PR #12229: Less ACCEPTS, more numpydoc.
- PR #12209: Doc: Sort named colors example by palette
- PR #12237: Use (float, float) as parameter type for 2D positions in docstrings
- PR #12238: Typo in docs
- PR #12236: Make boilerplate-generated pyplot.py flake8 compliant
- PR #12231: CI: Speed up Appveyor repository cloning
- PR #12228: Fix trivial typo in docs.
- PR #12227: Use (float, float) as parameter type for 2D positions
- PR #12199: Allow disabling specific mouse actions in blocking_input
- PR #12213: Change win32InstalledFonts return value
- PR #12207: FIX: dont' check for interactive framework if none required
- PR #11688: Don't draw axis (spines, ticks, labels) twice when using parasite axes.
- PR #12210: Axes.tick_params() argument checking
- PR #12211: Fix typo
- PR #12200: Slightly clarify some invalid shape exceptions for image data.
- PR #12151: Don't pretend @deprecated applies to classmethods.
- PR #12190: Remove some unused variables and imports
- PR #12186: DOC: fix API note about get_tightbbox
- PR #12203: Document legend's slowness when "best" location is used
- PR #12192: Exclude examples from lgtm analysis
- PR #12196: Give Carreau the ability to mention the backport bot.

- PR #12187: DOC: Update INSTALL.rst
- PR #12164: Fix Annotation.contains.
- PR #12177: FIX: remove cwd from mac font path search
- PR #12182: Fix Flash of Unstyled Content by removing remaining Flipcause integration
- PR #12184: DOC: update "Previous What's New" for 2.2 with reference to cividis paper
- PR #12183: Doc: Don't use Sphinx 1.8
- PR #12171: Remove internal warning due to zsort deprecation
- PR #12166: Document preference order for backend auto selection
- PR #12154: Avoid triggering deprecation warnings with pytest 3.8.
- PR #12030: Speed up canvas redraw for GTK3Agg backend.
- PR #12157: Properly declare the interactive framework for the qt4foo backends.
- PR #12156: Cleanup the GridSpec demos.
- PR #12144: Add explicit getters and setters for Annotation.anncoords.
- PR #12152: Use _warn_external for deprecations warnings.
- PR #12148: BLD: pragmatic fix for building basic_unit example on py37
- PR #12147: DOC: update the gh_stats code

Issues (170):

- #12699: Annotations get cropped out of figures saved with bbox_inches='tight'
- #9217: Weirdness with inline figure DPI settings in Jupyter Notebook
- #4853: %matplotlib notebook creates much bigger figures than %matplotlib inline
- #12780: Vague/misleading exception message in scatter()
- #10239: Weird interaction with Tkinter
- #10045: subplots_adjust() breaks layout of tick labels
- #12765: Matplotlib draws incorrect color
- #11800: Gridspec tutorial
- #12757: up the figure
- #12724: Importing pyplot steals focus on macOS
- #12669: fixing _hold on cartopy broke basemap
- #12687: Plotting text on 3d axes before tight_layout() breaks tight_layout()
- #12734: Wishlist: functionally linked twin axes

- #12576: RcParams is fundamentally broken
- #12641: `_axes.py.scatter()` array index out of bound / calling from `seaborn`
- #12703: Error when closing first of several pyplot figures in TkAgg
- #12728: Deprecation Warnings
- #4124: Provide canonical examples of mpl in web frameworks
- #10574: Default color after setting alptha to Patch in legened
- #12702: couldn't find or load Qt platform plugin "windows" in "".
- #11139: "make clean" doesn't remove all the build doc files
- #12701: semilogy with NaN prevents display of Title (cairo backend)
- #12696: Process finished with exit code -1 due to matplotlib configuration
- #12692: matplotlib.plot.show always blocks the execution of python script
- #12433: Travis error is MacOS image tolerance of 0.005 for `test_constrained_layout.py::test_colorbar_location`
- #10017: unicode_literals considered harmful
- #12682: using AxesImage.set_clim() shrinks the colorbar
- #12620: Overlapping 3D objects
- #12680: matplotlib ui in thread still blocked
- #11908: Improve linestyle documentation
- #12650: Deprecation warnings when calling help(matplotlib)
- #10643: Most warnings calls do not set the stacklevel
- #12671: make_axes_locatable breaks with matplotlib 3.0
- #12664: plt.scatter crashes because overwrites the colors to an empty list
- #12188: matplotlib 3 pyplot on MacOS bounces rocket icon in dock
- #12648: Regression when calling annotate with nan values for the position
- #12362: In 3.0.0 backend cannot be set if 'get_backend()' is run first
- #12649: Over-verbose deprecation warning about examples.directory
- #12661: In version 3.0.0 make_axes_locatable + colorbar does not produce expected result
- #12634: axes_grid1 axes have no keyword argument 'bbox_extra_artists'
- #12654: Broken 'Developer Discussions' link
- #12657: With v3.0.0 mpl_toolkits.axes_grid1.make_axes_locatable().append_axes breaks in Jupyter

- #12645: Markers are offset when 'facecolor' or 'edgecolor' are set to 'none' when plotting data
- #12644: Memory leak with plt.plot in Jupyter Notebooks?
- #12632: Do we need input hooks macosx?
- #12535: AIX Support - Do not undef _XOPEN_SOURCE
- #12626: AttributeError: module 'matplotlib' has no attribute 'artist'
- #11034: Doc Typo: matplotlib.axes.Axes.get_yticklabels / Axis.get_ticklabels
- #12624: make_axes_locatable : Colorbar in the middle instead of bottom while saving a pdf, png.
- #11094: can not use GUI backends inside django request handlers
- #12613: transiently linked interactivity of unshared pair of axes generated with make_axes_locatable
- #12578: macOS builds are broken
- #12612: gui backends do not work inside of flask request handlers
- #12611: Matplotlib 3.0.0 Likely bug TypeError: stackplot() got multiple values for argument 'x'
- #12610: matplotlibrc causes import to fail 3.0.0 (didn't crash 2.y.z series)
- #12601: Can't import matplotlib
- #12597: Please soon add Chinese language support!! It's to difficult for new people handle character
- #12590: Matplotlib pypi distribution lacks packages for Python 2.7
- #3869: Numeric labels do not work with plt.hist
- #12580: Incorrect hist error message with bad color size
- #12100: document where to get nightly wheels
- #7205: Converting docstrings to numpydoc
- #12564: Saving plot as PNG file prunes tick labels
- #12161: Problems of using sharex options with lines plots and colormesh with colorbar
- #12256: tight_layout for plot with non-clipped screen-unit items causes issues on zoom
- #12545: Program quit unormally without reporting error
- #12532: Incorrect rendering of math symbols
- #12567: Calling pyplot.show() with TkAgg backend on x86 machine raises OverflowError.

- #12571: cannot install because Fatal Python error: initfsencoding: Unable to get the locale encoding
- #12566: Problem installing Version 1.3.1 -> missing pkg-config freetype and libagg
- #12556: Matplotlib 3.0.0 import hangs in clean environment
- #12197: Weird behaviour of suptitle() when horizontalalignment is not 'center'
- #12550: colorbar resizes in animation
- #12155: Incorrect placement of Colorbar ticks using LogNorm
- #11787: Looping gifs with PillowWriter
- #12533: Plotting with alpha=0 with rasterized=True causes ValueError on saving to pdf
- #12438: Scatter doesn't accept a list of strings as color spec.
- #12429: scatter() does not accept gray strings anymore
- #12499: run my code failed after i Import pylab failed, python version is 3.6.6
- #12458: add_lines misses lines for matplotlib.colorbar.ColorbarBase
- #12239: 3d axes are collapsed by tight_layout
- #12414: Function to draw angle between two lines
- #12488: inconsistent colorbar tick labels for LogNorm
- #12515: pyplot.step broken in 3.0.0?
- #12355: Error for bbox_inches='tight' in savefig with make_axes_locatable
- #12505: ImageGrid in 3.0
- #12502: How can I put the ticks of logarithmic coordinate in the axes?
- #12496: Maplotlib Can't Plot a Dataset
- #12486: rotate label of legend ?
- #12291: Importing pyplot crashes on macOS due to missing fontlist-v300.json and then Permission denied: '/opt/local/share/fonts'
- #12480: "close_event" for nbagg/notebook backend
- #12467: Documentation of AnchoredText's prop keyword argument is misleading
- #12288: New function signatures in pyplot break Cartopy
- #12445: Error on colorbar
- #8760: Traceback from animation.MovieWriter.saving method is confusing because it provides no useful information

- #9205: after the animation encoder (e.g. ffmpeg) fails, the animation framework itself fails internally in various ways while trying to report the error
- #12357: Unclear error when saving Animation using FFMpeg
- #12454: Formatting numerical legend
- #9636: matplotlib crashes upon window resize
- #11473: Continuous plotting cause memory leak 20-50kb/sec
- #12018: No image pop-up or display for plt.imshow() and plt.show()
- #11583: How to draw parallelepiped with real size scaling?
- #12446: Polar Contour - float() argument must be a string or a number, not 'AxesParasiteParasiteAuxTrans'
- #12444: Issues with gridspec/tight_layout in matplotlib version 2.2.3
- #11154: Unexpected behavior for Axes3D.plot_surface(shade=None)
- #12409: Calling savefig() multiple times causes crash of Spyder IDE / IPython Kernel dying.
- #9799: FigureCanvasTkAgg - "buffer is of wrong type" error during blit
- #12439: FileNotFoundError for font_manager
- #12437: matplotlib-mac
- #12121: Documentation of TextArea's fontprops keyword argument is misleading
- #12279: Axes.format_cursor_data lacks documentation and seems unused
- #12428: Simple plot spacing bug: ylabel gets wrongfully removed from plot
- #11190: Images in the docs are too large.
- #12271: error with errorbar with datetime64
- #12405: plt.stackplot() does not work with 3.0.0
- #12282: `Axes.imshow` tooltip does not get updated when another call to `Axes.imshow` is made
- #12420: How to remove Rectangle Selector from figure?
- #12391: Constrained Layout tutorial needs some cleanup....
- #12406: Bug with font finding, and here is my fix as well.
- #9051: ParasiteAxes over plotting
- #12325: Annotation change from "s" to "text" in 3.0- documentation
- #12397: plt.show( ) not working (can't get figures to display in external window) when using jupyter QTconsole
- #12396: Defining arrowprops in draggable annotation disables the pick_event

- #12389: Setting row edge color of matplotlib table
- #12376: The output figure file is strange: there is a lot of blank area on the output figure.
- #11641: constrained_layout and colorbar for a subset of axes
- #12373: Unexpected outcome with matplotlib.pyplot.pcolor()
- #12370: ImageGrid bug when using inline backend
- #12364: pdf image generated by matplotlib with semi transparent lines missing in Word on Windows.
- #12352: TeX rendering broken on master with windows
- #12354: Too many levels of symbolic links
- #12323: indicate_inset_zoom sometimes draws incorrect connector lines
- #12341: Figures not rendering in docker
- #12335: Matplotlib plt.Rectangle Incoherent Results
- #12265: ParasiteAxesAuxTrans pcolor/pcolormesh and contour/contourf broken
- #12337: AttributeError: module 'matplotlib.pyplot' has no attribute 'hold'
- #11673: Inconsistent font settings when changing style context
- #11693: The rcParams setting for figure.figsize does not change when run from another notebook
- #11725: New mode between non-interactive and interactive?
- #12134: tight_layout flips images when making plots without displaying them
- #12310: plot fails with datetime64[ns] timezone aware objects (for example datetime64[ns, UTC+00:00])
- #12191: "if 1:" blocks in examples
- #11288: FR: Figure.subplots add optional SubplotSpec parameter
- #12298: c and cmap for plot
- #12286: Sample code given in Matplotlib's site does not work.
- #11955: UnicodeDecodeError on importing pyplot in python2
- #12208: parasite axis2 demo now crashes with log x-axis
- #8871: Error when using quantities when plotting errorbars
- #6658: literature reference for 'viridis' colormap
- #6789: Tutorial pyplot_scales.py crashes when used with plt.tight_layout()
- #6922: imshow does not immediately update shared axes
- #11879: Unable to change filename when saving from figure window

- #12225: In histogram, bars whose count is larger than 2**31 sometimes become negative
- #1461: DOC: keyword arguments to plt.axes, plt.subpot, and fig.add_subplot
- #12173: Cannot import pyplot
- #12217: Python will suddenly not plot anymore
- #12120: Default legend behavior (loc='best') very slow for large amounts of data.
- #12176: import pyplot on MacOS without font cache will search entire subtree of current dir
- #12146: fix pdf docs
- #12160: MacOS: Cannot import name 'format_exc'
- #12169: Cannot install 3.0.0 "python setup.py egg_info" failed (freetype & png)
- #12168: pip install v3.0.0 'failed with exit status 1181'
- #12107: warnings re: deprecated pytest API with pytest 3.8
- #12162: https://matplotlib.org/users/beginner.html is outdated
- #12010: Popover over plot is very slow
- #6739: Make matplotlib fail more gracefully in headless environments
- #3679: Runtime detection for default backend
- #11340: matplotlib fails to install from source with intel compiler
- #11838: docs do not build on py3.7 due to small change in python handling of -m
- #12115: Plot in JS Animation has larger margin than "normal" PNG plot

CHAPTER

EIGHT

PREVIOUS WHAT'S NEW

8.1 What's new in Matplotlib 3.2

For a list of all of the issues and pull requests since the last revision, see the *GitHub Stats*.

Table of Contents

8.1.1 Unit converters recognize subclasses

Unit converters now also handle instances of subclasses of the class they have been registered for.

8.1.2 `imsave` accepts metadata and PIL options

imsave has gained support for the `metadata` and `pil_kwargs` parameters. These parameters behave similarly as for the *Figure.savefig()* method.

8.1.3 `cbook.normalize_kwargs`

cbook.normalize_kwargs now presents a convenient interface to normalize artist properties (e.g., from "lw" to "linewidth"):

```
>>> cbook.normalize_kwargs({"lw": 1}, Line2D)
{"linewidth": 1}
```

The first argument is the mapping to be normalized, and the second argument can be an artist class or an artist instance (it can also be a mapping in a specific format; see the function's docstring for details).

8.1.4 `FontProperties` accepts `os.PathLike`

The *fname* argument to *FontProperties* can now be an `os.PathLike`, e.g.

```
>>> FontProperties(fname=pathlib.Path("/path/to/font.ttf"))
```

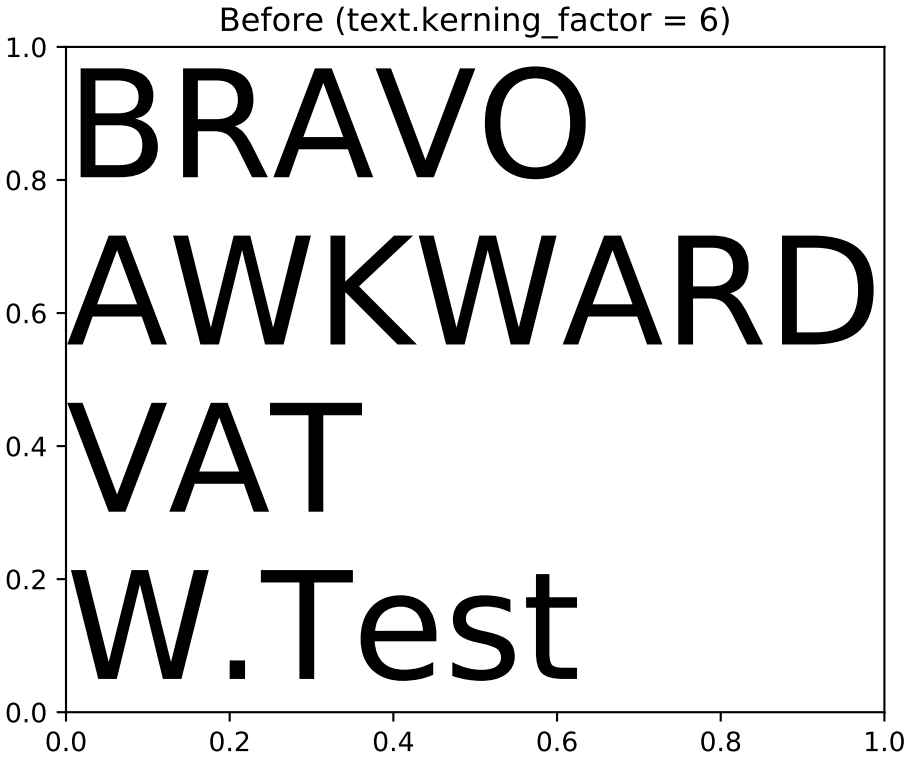
8.1.5 Gouraud-shading alpha channel in PDF backend

The pdf backend now supports an alpha channel in Gouraud-shaded triangle meshes.

8.1.6 Kerning adjustments now use correct values

Due to an error in how kerning adjustments were applied, previous versions of Matplotlib would under-correct kerning. This version will now correctly apply kerning (for fonts supported by FreeType). To restore the old behavior (e.g., for test images), you may set `rcParams["text.kerning_factor"]` (default: `0`) to 6 (instead of 0). Other values have undefined behavior.

Note how the spacing between characters is uniform between their bounding boxes (above). With corrected kerning (below), slanted characters (e.g., AV or VA) will be spaced closer together, as well as various other character pairs, depending on font support (e.g., T and e, or the period after the W).



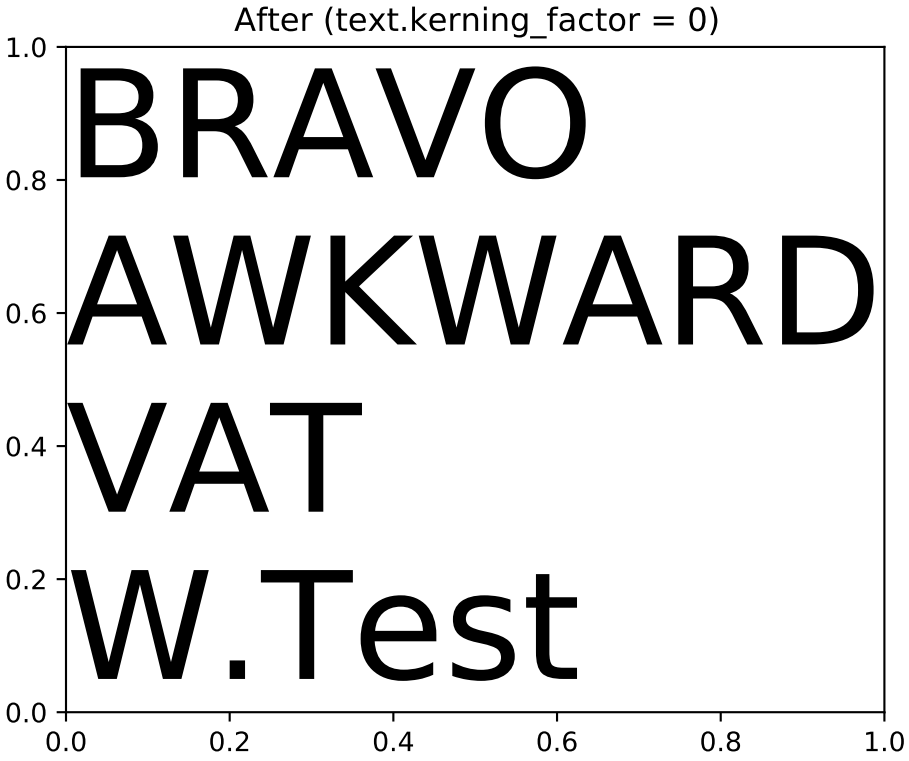
8.1.7 bar3d lightsource shading

bar3d() now supports lighting from different angles when the *shade* parameter is `True`, which can be configured using the `lightsource` parameter.

8.1.8 Shifting errorbars

Previously, *errorbar()* accepted a keyword argument *errorevery* such that the command `plt.errorbar(x, y, yerr, errorevery=6)` would add error bars to datapoints `x[::6], y[::6]`.

errorbar() now also accepts a tuple for *errorevery* such that `plt.errorbar(x, y, yerr, errorevery=(start, N))` adds error bars to points `x[start::N], y[start::N]`.



8.1.9 Improvements in Logit scale ticker and formatter

Introduced in version 1.5, the logit scale didn't have an appropriate ticker and formatter. Previously, the location of ticks was not zoom dependent, too many labels were displayed causing overlapping which broke readability, and label formatting did not adapt to precision.

Starting from this version, the logit locator has nearly the same behavior as the locator for the log scale or the linear scale, depending on used zoom. The number of ticks is controlled. Some minor labels are displayed adaptively as sublabels in log scale. Formatting is adapted for probabilities and the precision adapts to the scale.

8.1.10 rcParams for axes title location and color

Two new rcParams have been added: `rcParams["axes.titlelocation"]` (default: `'center'`) denotes the default axes title alignment, and `rcParams["axes.titlecolor"]` (default: `'auto'`) the default axes title color.

Valid values for `axes.titlelocation` are: left, center, and right. Valid values for `axes.titlecolor` are: auto or a color. Setting it to auto will fall back to previous behaviour, which is using the color in `text.color`.

8.1.11 3-digit and 4-digit hex colors

Colors can now be specified using 3-digit or 4-digit hex colors, shorthand for the colors obtained by duplicating each character, e.g. `#123` is equivalent to `#112233` and `#123a` is equivalent to `#112233aa`.

8.1.12 Added support for RGB(A) images in pcolorfast

`Axes.pcolorfast` now accepts 3D images (RGB or RGBA) arrays.

8.2 What's new in Matplotlib 3.1

For a list of all of the issues and pull requests since the last revision, see the *GitHub Stats*.

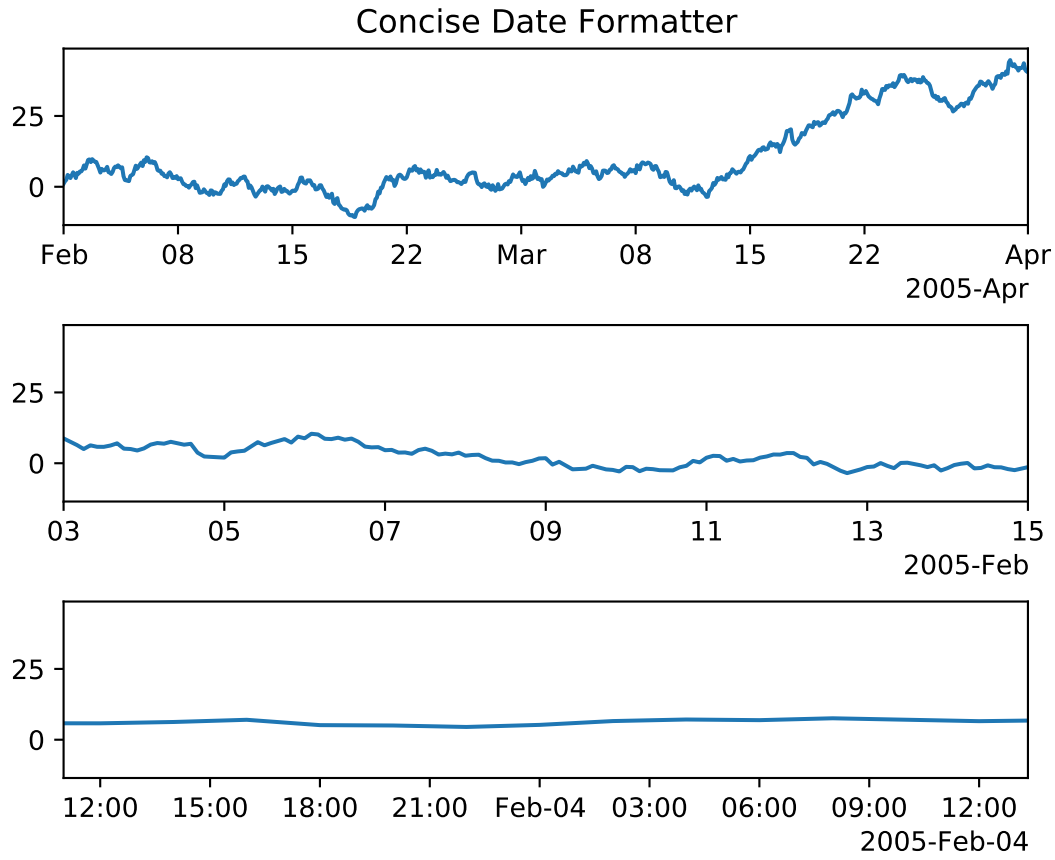
Table of Contents

    * *mplot3d Line3D now allows {set,get}_data_3d*
    * *`Axes3D.voxels` now shades the resulting voxels*
  - *Axis and Ticks*
    * *Added `Axis.get_inverted` and `Axis.set_inverted`*
    * *Adjust default minor tick spacing*
    * `EngFormatter` *now accepts* usetex, useMathText *as keyword only arguments*
  - *Animation and Interactivity*
    * *Support for forward/backward mouse buttons*
    * progress_callback *argument to* `save()`
    * *Add `cache_frame_data` keyword-only argument into* `animation.FuncAnimation`
    * *Endless Looping GIFs with PillowWriter*
    * *Adjusted* `matplotlib.widgets.Slider` *to have vertical orientation*
    * *Improved formatting of image values under cursor when a colorbar is present*
    * *MouseEvent button attribute is now an IntEnum*
  - *Configuration, Install, and Development*
    * *The MATPLOTLIBRC environment variable can now point to any "file" path*
    * *Allow LaTeX code `pgf.preamble` and `text.latex.preamble` in MATPLOTLIBRC file*
    * *New logging API*

8.2.1 New Features

`ConciseDateFormatter`

The automatic date formatter used by default can be quite verbose. A new formatter can be accessed that tries to make the tick labels appropriately concise.



Secondary x/y Axis support

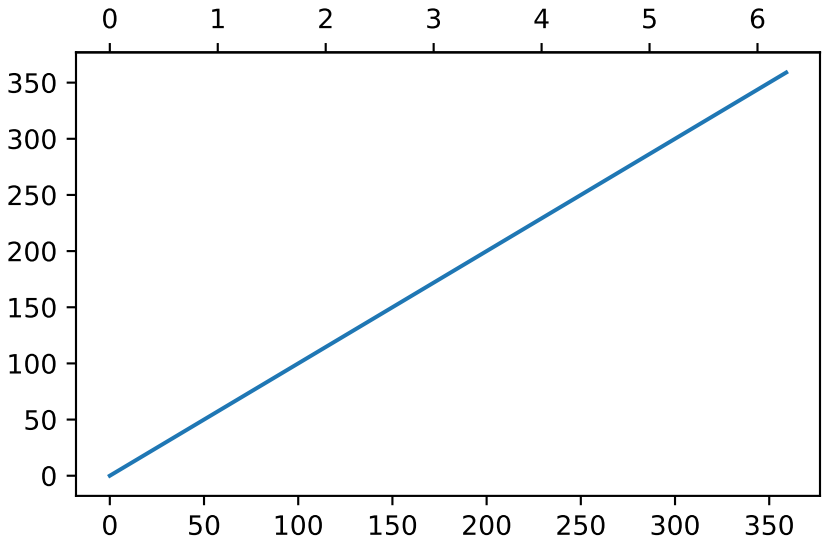
A new method provides the ability to add a second axis to an existing axes via *Axes.secondary_xaxis* and *Axes.secondary_yaxis*. See /gallery/subplots_axes_and_figures/secondary_axis for examples.

`FuncScale` for arbitrary axes scales

A new *FuncScale* class was added (and *FuncTransform*) to allow the user to have arbitrary scale transformations without having to write a new subclass of *ScaleBase*. This can be accessed by:

```
ax.set_yscale('function', functions=(forward, inverse))
```

where `forward` and `inverse` are callables that return the scale transform and its inverse. See the last example in /gallery/scales/scales.



Legend for scatter

A new method for creating legends for scatter plots has been introduced. Previously, in order to obtain a legend for a *scatter()* plot, one could either plot several scatters, each with an individual label, or create proxy artists to show in the legend manually. Now, *PathCollection* provides a method *legend_elements()* to obtain the handles and labels for a scatter plot in an automated way. This makes creating a legend for a scatter plot as easy as

An example can be found in automatedlegendcreation.

Matplotlib no longer requires framework app build on MacOSX backend

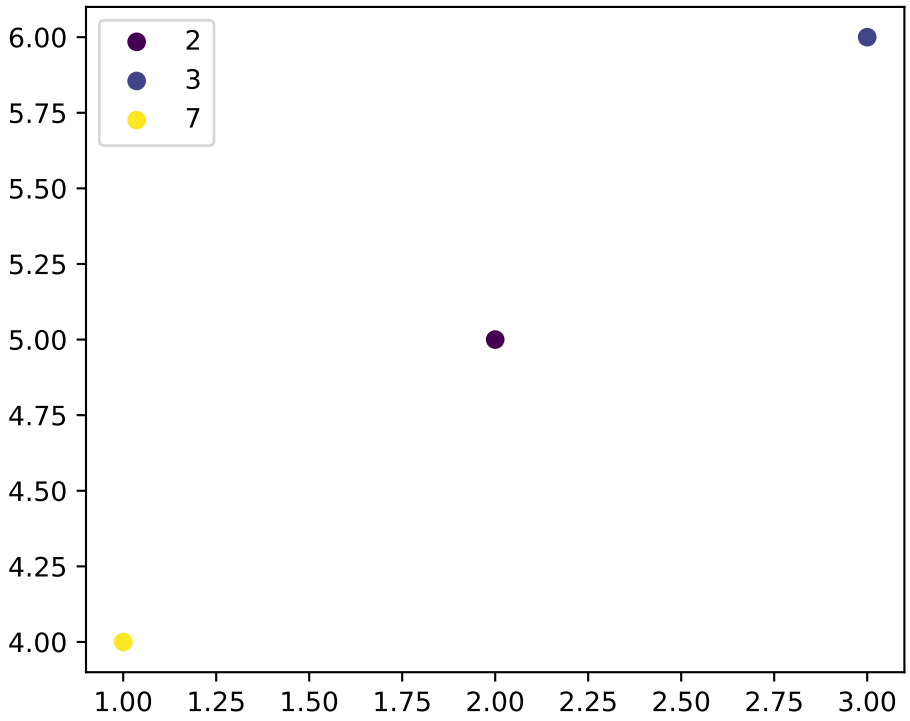
Previous versions of matplotlib required a Framework build of python to work. The app type was updated to no longer require this, so the MacOSX backend should work with non-framework python.

This also adds support for the MacOSX backend for PyPy3.

8.2.2 Figure, FigureCanvas, and Backends

Figure.frameon is now a direct proxy for the Figure patch visibility state

Accessing `Figure.frameon` (including via `get_frameon` and `set_frameon` now directly forwards to the visibility of the underlying Rectangle artist (`Figure.patch.get_frameon`, `Figure.patch.set_frameon`).



pil_kwargs argument added to savefig

Matplotlib uses Pillow to handle saving to the JPEG and TIFF formats. The *savefig()* function gained a *pil_kwargs* keyword argument, which can be used to forward arguments to Pillow's `pillow.Image.save()`.

The *pil_kwargs* argument can also be used when saving to PNG. In that case, Matplotlib also uses Pillow's `pillow.Image.save()` instead of going through its own builtin PNG support.

Add `inaxes` method to `FigureCanvasBase`

The *FigureCanvasBase* class has now an *inaxes* method to check whether a point is in an axes and returns the topmost axes, else None.

cairo backend defaults to pycairo instead of cairocffi

This leads to faster import/runtime performance in some cases. The backend will fall back to cairocffi in case pycairo isn't available.

8.2.3 Axes and Artists

axes_grid1 and axisartist Axes no longer draw spines twice

Previously, spines of *axes_grid1* and *axisartist* Axes would be drawn twice, leading to a "bold" appearance. This is no longer the case.

Return type of ArtistInspector.get_aliases changed

ArtistInspector.get_aliases previously returned the set of aliases as `{fullname: {alias1: None, alias2: None, ...}}`. The dict-to-None mapping was used to simulate a set in earlier versions of Python. It has now been replaced by a set, i.e. `{fullname: {alias1, alias2, ...}}`.

This value is also stored in `ArtistInspector.aliasd`, which has likewise changed.

`ConnectionPatch` accepts arbitrary transforms

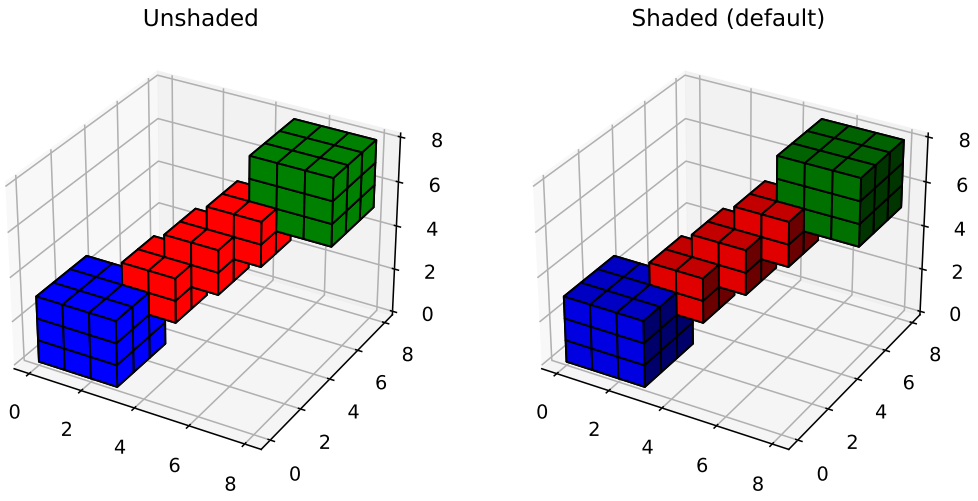
Alternatively to strings like `"data"` or `"axes fraction"`, `ConnectionPatch` now accepts any *Transform* as input for the *coordsA* and *coordsB* arguments. This allows to draw lines between points defined in different user defined coordinate systems. Also see the Connect Simple01 example.

mplot3d Line3D now allows {set,get}_data_3d

Lines created with the 3d projection in mplot3d can now access the data using *get_data_3d()* which returns a tuple of array_likes containing the (x, y, z) data. The equivalent *set_data_3d* can be used to modify the data of an existing Line3D.

`Axes3D.voxels` now shades the resulting voxels

The *Axes3D.voxels* method now takes a *shade* parameter that defaults to `True`. This shades faces based on their orientation, behaving just like the matching parameters to *plot_trisurf()* and *bar3d()*. The plot below shows how this affects the output.



8.2.4 Axis and Ticks

Added `Axis.get_inverted` and `Axis.set_inverted`

The `Axis.get_inverted` and `Axis.set_inverted` methods query and set whether the axis uses "inverted" orientation (i.e. increasing to the left for the x-axis and to the bottom for the y-axis).

They perform tasks similar to *Axes.xaxis_inverted*, *Axes.yaxis_inverted*, *Axes.invert_xaxis*, and *Axes.invert_yaxis*, with the specific difference that `Axis.set_inverted` makes it easier to set the inversion of an axis regardless of whether it had previously been inverted before.

Adjust default minor tick spacing

Default minor tick spacing was changed from 0.625 to 0.5 for major ticks spaced 2.5 units apart.

`EngFormatter` now accepts *usetex*, *useMathText* as keyword only arguments

A public API has been added to *EngFormatter* to control how the numbers in the ticklabels will be rendered. By default, *useMathText* evaluates to `rcParams["axes.formatter.use_mathtext'"]` and *usetex* evaluates to `rcParams["'text.usetex'"]`.

If either is `True` then the numbers will be encapsulated by `$` signs. When using `TeX` this implies that the numbers will be shown in TeX's math font. When using mathtext, the `$` signs around numbers will ensure Unicode rendering (as implied by mathtext). This will make sure that the minus signs in the ticks are rendered as the Unicode minus (U+2212) when using mathtext (without relying on the *fix_minus* method).

8.2.5 Animation and Interactivity

Support for forward/backward mouse buttons

Figure managers now support a `button_press` event for mouse buttons, similar to the `key_press` events. This allows binding actions to mouse buttons (see *MouseButton*) The first application of this mechanism is support of forward/backward mouse buttons in figures created with the Qt5 backend.

progress_callback argument to `save()`

The method *Animation.save* gained an optional *progress_callback* argument to notify the saving progress.

Add `cache_frame_data` keyword-only argument into `animation.FuncAnimation`

matplotlib.animation.FuncAnimation has been caching frame data by default; however, this caching is not ideal in certain cases e.g. When *FuncAnimation* needs to be only drawn(not saved) interactively and memory required by frame data is quite large. By adding *cache_frame_data* keyword-only argument, users can now disable this caching; thereby, this new argument provides a fix for issue #8528.

Endless Looping GIFs with PillowWriter

We acknowledge that most people want to watch a GIF more than once. Saving an animation as a GIF with PillowWriter now produces an endless looping GIF.

Adjusted `matplotlib.widgets.Slider` to have vertical orientation

The *matplotlib.widgets.Slider* widget now takes an optional argument *orientation* which indicates the direction (`'horizontal'` or `'vertical'`) that the slider should take.

Improved formatting of image values under cursor when a colorbar is present

When a colorbar is present, its formatter is now used to format the image values under the mouse cursor in the status bar. For example, for an image displaying the values 10,000 and 10,001, the statusbar will now (using default settings) display the values as `10000` and `10001`), whereas both values were previously displayed as `1e+04`.

MouseEvent button attribute is now an IntEnum

The `button` attribute of *MouseEvent* instances can take the values None, 1 (left button), 2 (middle button), 3 (right button), "up" (scroll), and "down" (scroll). For better legibility, the 1, 2, and 3 values are now represented using the `enum.IntEnum` class *matplotlib.backend_bases.MouseButton*, with the values *MouseButton.LEFT* (`== 1`), *MouseButton.MIDDLE* (`== 2`), and *MouseButton.RIGHT* (`== 3`).

8.2.6 Configuration, Install, and Development

The MATPLOTLIBRC environment variable can now point to any "file" path

This includes device files; in particular, on Unix systems, one can set `MATPLOTLIBRC` to `/dev/null` to ignore the user's matplotlibrc file and fall back to Matplotlib's defaults.

As a reminder, if `MATPLOTLIBRC` points to a directory, Matplotlib will try to load the matplotlibrc file from `$MATPLOTLIBRC/matplotlibrc`.

Allow LaTeX code `pgf.preamble` and `text.latex.preamble` in MATPLOTLIBRC file

Previously, the rc file keys `rcParams["pgf.preamble"]` (default: `''`) and `rcParams["text.latex.preamble"]` (default: `''`) were parsed using commas as separators. This would break valid LaTeX code, such as:

```
\usepackage[protrusion=true, expansion=false]{microtype}
```

The parsing has been modified to pass the complete line to the LaTeX system, keeping all commas. Passing a list of strings from within a Python script still works as it used to.

New logging API

`matplotlib.set_loglevel` / `pyplot.set_loglevel` can be called to display more (or less) detailed logging output.

8.3 New in Matplotlib 3.0

8.3.1 Improved default backend selection

The default backend no longer must be set as part of the build process. Instead, at run time, the builtin backends are tried in sequence until one of them imports.

Headless Linux servers (identified by the DISPLAY environment variable not being defined) will not select a GUI backend.

8.3.2 Cyclic colormaps

Two new colormaps named 'twilight' and 'twilight_shifted' have been added. These colormaps start and end on the same color, and have two symmetric halves with equal lightness, but diverging color. Since they wrap around, they are a good choice for cyclic data such as phase angles, compass directions, or time of day. Like *viridis* and *cividis*, *twilight* is perceptually uniform and colorblind friendly.

8.3.3 Ability to scale axis by a fixed order of magnitude

To scale an axis by a fixed order of magnitude, set the *scilimits* argument of *Axes.ticklabel_format* to the same (non-zero) lower and upper limits. Say to scale the y axis by a million (1e6), use

```
ax.ticklabel_format(style='sci', scilimits=(6, 6), axis='y')
```

The behavior of `scilimits=(0, 0)` is unchanged. With this setting, Matplotlib will adjust the order of magnitude depending on the axis values, rather than keeping it fixed. Previously, setting `scilimits=(m, m)` was equivalent to setting `scilimits=(0, 0)`.

8.3.4 Add `AnchoredDirectionArrows` feature to mpl_toolkits

A new mpl_toolkits class *AnchoredDirectionArrows* draws a pair of orthogonal arrows to indicate directions on a 2D plot. A minimal working example takes in the transformation object for the coordinate system (typically ax.transAxes), and arrow labels. There are several optional parameters that can be used to alter layout. For example, the arrow pairs can be rotated and the color can be changed. By default the labels and arrows have the same color, but the class may also pass arguments for customizing arrow and text layout, these are passed to *matplotlib.textpath.TextPath* and *matplotlib.patches.FancyArrowPatch*. Location, length and width for both arrow tail and head can be adjusted, the direction arrows and labels can have a frame. Padding and separation parameters can be adjusted.

8.3.5 Add `minorticks_on()/off()` methods for colorbar

A new method `colorbar.Colobar.minorticks_on()` has been added to correctly display minor ticks on a colorbar. This method doesn't allow the minor ticks to extend into the regions beyond vmin and vmax when the *extend* keyword argument (used while creating the colorbar) is set to 'both', 'max' or 'min'. A complementary method `colorbar.Colobar.minorticks_off()` has also been added to remove the minor ticks on the colorbar.

8.3.6 Colorbar ticks can now be automatic

The number of ticks placed on colorbars was previously appropriate for a large colorbar, but looked bad if the colorbar was made smaller (i.e. via the *shrink* keyword argument). This has been changed so that the number of ticks is now responsive to how large the colorbar is.

8.3.7 Don't automatically rename duplicate file names

Previously, when saving a figure to a file using the GUI's save dialog box, if the default filename (based on the figure window title) already existed on disk, Matplotlib would append a suffix (e.g. `Figure_1-1.png`), preventing the dialog from prompting to overwrite the file. This behaviour has been removed. Now if the file name exists on disk, the user is prompted whether or not to overwrite it. This eliminates guesswork, and allows intentional overwriting, especially when the figure name has been manually set using `figure.Figure.canvas.set_window_title()`.

8.3.8 Legend now has a *title_fontsize* keyword argument (and rcParam)

The title for a *Figure.legend* and *Axes.legend* can now have its font size set via the *title_fontsize* keyword argument. There is also a new `rcParams["legend.title_fontsize"]` (default: `None`). Both default to `None`, which means the legend title will have the same font size as the axes default font size (*not* the legend font size, set by the *fontsize* keyword argument or `rcParams["legend.fontsize"]` (default: `'medium'`)).

8.3.9 Support for axes.prop_cycle property *markevery* in rcParams

The Matplotlib `rcParams` settings object now supports configuration of the attribute `axes.prop_cycle` with cyclers using the `markevery` Line2D object property. An example of this feature is provided at /gallery/lines_bars_and_markers/markevery_prop_cycle.

8.3.10 Multi-page PDF support for pgf backend

The pgf backend now also supports multi-page PDF files.

```
from matplotlib.backends.backend_pgf import PdfPages
import matplotlib.pyplot as plt

with PdfPages('multipage.pdf') as pdf:
    # page 1
    plt.plot([2, 1, 3])
    pdf.savefig()

    # page 2
```

(continues on next page)

(continued from previous page)

```
    plt.cla()
    plt.plot([3, 1, 2])
    pdf.savefig()
```

8.3.11 Pie charts are now circular by default

We acknowledge that the majority of people do not like egg-shaped pies. Therefore, an axes to which a pie chart is plotted will be set to have equal aspect ratio by default. This ensures that the pie appears circular independent on the axes size or units. To revert to the previous behaviour set the axes' aspect ratio to automatic by using `ax.set_aspect("auto")` or `plt.axis("auto")`.

8.3.12 Add `ax.get_gridspec` to `SubplotBase`

New method *`SubplotBase.get_gridspec`* is added so that users can easily get the gridspec that went into making an axes:

```
import matplotlib.pyplot as plt

fig, axs = plt.subplots(3, 2)
gs = axs[0, -1].get_gridspec()

# remove the last column
for ax in axs[:,-1].flatten():
  ax.remove()

# make a subplot in last column that spans rows.
ax = fig.add_subplot(gs[:, -1])
plt.show()
```

8.3.13 Axes titles will no longer overlap xaxis

Previously an axes title had to be moved manually if an xaxis overlapped (usually when the xaxis was put on the top of the axes). Now, the title will be automatically moved above the xaxis and its decorators (including the xlabel) if they are at the top.

If desired, the title can still be placed manually. There is a slight kludge; the algorithm checks if the y-position of the title is 1.0 (the default), and moves if it is. If the user places the title in the default location (i.e. `ax.title.set_position(0.5, 1.0)`), the title will still be moved above the xaxis. If the user wants to avoid this, they can specify a number that is close (i.e. `ax.title.set_position(0.5, 1.01)`) and the title will not be moved via this algorithm.

8.3.14 New convenience methods for GridSpec

There are new convenience methods for *gridspec.GridSpec* and *gridspec.GridSpecFromSubplotSpec*. Instead of the former we can now call *Figure.add_gridspec* and for the latter *SubplotSpec.subgridspec*.

```
import matplotlib.pyplot as plt

fig = plt.figure()
gs0 = fig.add_gridspec(3, 1)
ax1 = fig.add_subplot(gs0[0])
ax2 = fig.add_subplot(gs0[1])
gssub = gs0[2].subgridspec(1, 3)
for i in range(3):
    fig.add_subplot(gssub[0, i])
```

8.3.15 Figure has an `add_artist` method

A method *add_artist* has been added to the *Figure* class, which allows artists to be added directly to a figure. E.g.

```
circ = plt.Circle((.7, .5), .05)
fig.add_artist(circ)
```

In case the added artist has no transform set previously, it will be set to the figure transform (`fig.transFigure`). This new method may be useful for adding artists to figures without axes or to easily position static elements in figure coordinates.

8.3.16 `:math:` directive renamed to `:mathmpl:`

The `:math:` rst role provided by `matplotlib.sphinxext.mathmpl` has been renamed to `:mathmpl:` to avoid conflicting with the `:math:` role that Sphinx 1.8 provides by default. (`:mathmpl:` uses Matplotlib to render math expressions to images embedded in html, whereas Sphinx uses MathJax.)

When using Sphinx<1.8, both names (`:math:` and `:mathmpl:`) remain available for backwards-compatibility.

8.4 New in Matplotlib 2.2

8.4.1 Constrained Layout Manager

Warning: Constrained Layout is **experimental**. The behaviour and API are subject to change, or the whole functionality may be removed without a deprecation period.

A new method to automatically decide spacing between subplots and their organizing `GridSpec` instances has been added. It is meant to replace the venerable `tight_layout` method. It is invoked via a new `constrained_layout=True` kwarg to *Figure* or *subplots*.

There are new *rcParams* for this package, and spacing can be more finely tuned with the new *set_constrained_layout_pads*.

Features include:

- Automatic spacing for subplots with a fixed-size padding in inches around subplots and all their decorators, and space between as a fraction of subplot size between subplots.
- Spacing for *suptitle*, and colorbars that are attached to more than one axes.
- Nested *GridSpec* layouts using *GridSpecFromSubplotSpec*.

For more details and capabilities please see the new tutorial: *Constrained Layout Guide*

Note the new API to access this:

New `plt.figure` and `plt.subplots` kwarg: `constrained_layout`

figure() and *subplots()* can now be called with `constrained_layout=True` kwarg to enable constrained_layout.

New `ax.set_position` behaviour

Axes.set_position now makes the specified axis no longer responsive to `constrained_layout`, consistent with the idea that the user wants to place an axis manually.

Internally, this means that old `ax.set_position` calls *inside* the library are changed to private `ax._set_position` calls so that `constrained_layout` will still work with these axes.

New `figure` kwarg for `GridSpec`

In order to facilitate `constrained_layout`, `GridSpec` now accepts a `figure` keyword. This is backwards compatible, in that not supplying this will simply cause `constrained_layout` to not operate on the subplots orgainzed by this `GridSpec` instance. Routines that use `GridSpec` (e.g. `fig.subplots`) have been modified to pass the figure to `GridSpec`.

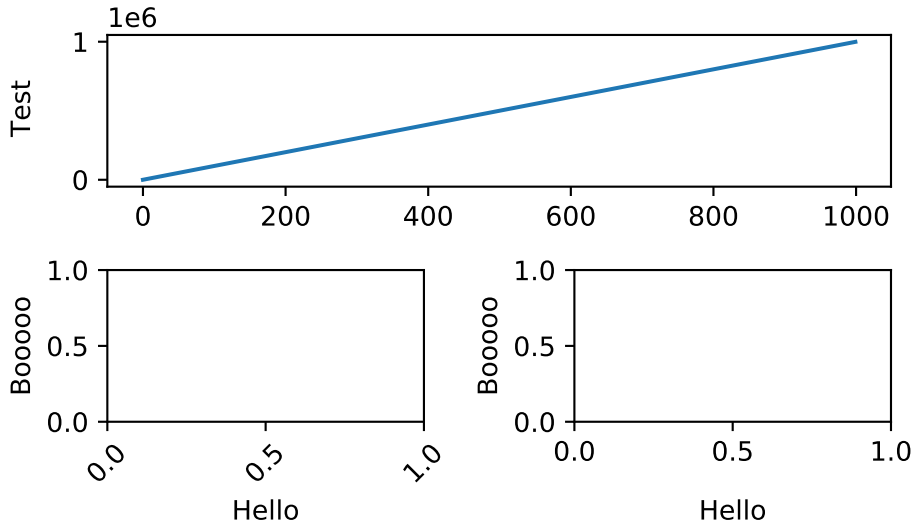
8.4.2 xlabels and ylabels can now be automatically aligned

Subplot axes `ylabels` can be misaligned horizontally if the tick labels are very different widths. The same can happen to `xlabels` if the ticklabels are rotated on one subplot (for instance). The new methods on the *Figure* class: *Figure.align_xlabels* and *Figure.align_ylabels* will now align these labels horizontally or vertically. If the user only wants to align some axes, a list of axes can be passed. If no list is passed, the algorithm looks at all the labels on the figure.

Only labels that have the same subplot locations are aligned. i.e. the ylabels are aligned only if the subplots are in the same column of the subplot layout.

Alignment is persistent and automatic after these are called.

A convenience wrapper *Figure.align_labels* calls both functions at once.

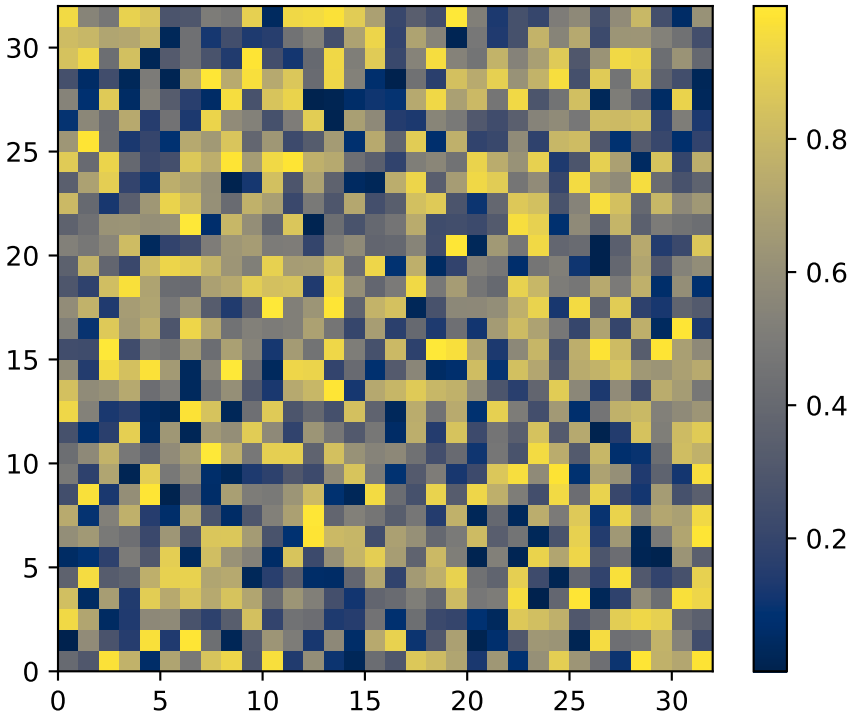


8.4.3 Axes legends now included in tight_bbox

Legends created via `ax.legend` can sometimes overspill the limits of the axis. Tools like `fig.tight_layout()` and `fig.savefig(bbox_inches='tight')` would clip these legends. A change was made to include them in the `tight` calculations.

8.4.4 Cividis colormap

A new dark blue/yellow colormap named 'cividis' was added. Like viridis, cividis is perceptually uniform and colorblind friendly. However, cividis also goes a step further: not only is it usable by colorblind users, it should actually look effectively identical to colorblind and non-colorblind users. For more details see Nuñez J, Anderton C, and Renslow R: "Optimizing colormaps with consideration for color vision deficiency to enable accurate interpretation of scientific data".



8.4.5 New style colorblind-friendly color cycle

A new style defining a color cycle has been added, tableau-colorblind10, to provide another option for colorblind-friendly plots. A demonstration of this new style can be found in the reference of style sheets. To load this color cycle in place of the default one:

```
import matplotlib.pyplot as plt
plt.style.use('tableau-colorblind10')
```

8.4.6 Support for numpy.datetime64

Matplotlib has supported `datetime.datetime` dates for a long time in *matplotlib.dates*. We now support `numpy.datetime64` dates as well. Anywhere that `datetime.datetime` could be used, `numpy.datetime64` can be used. eg:

```
time = np.arange('2005-02-01', '2005-02-02', dtype='datetime64[h]')
plt.plot(time)
```

8.4.7 Writing animations with Pillow

It is now possible to use Pillow as an animation writer. Supported output formats are currently gif (Pillow>=3.4) and webp (Pillow>=5.0). Use e.g. as

```
from __future__ import division

from matplotlib import pyplot as plt
from matplotlib.animation import FuncAnimation, PillowWriter

fig, ax = plt.subplots()
line, = plt.plot([0, 1])

def animate(i):
   line.set_ydata([0, i / 20])
   return [line]

anim = FuncAnimation(fig, animate, 20, blit=True)
anim.save("movie.gif", writer=PillowWriter(fps=24))
plt.show()
```

8.4.8 Slider UI widget can snap to discrete values

The slider UI widget can take the optional argument *valstep*. Doing so forces the slider to take on only discrete values, starting from *valmin* and counting up to *valmax* with steps of size *valstep*.

If *closedmax==True*, then the slider will snap to *valmax* as well.

8.4.9 `capstyle` and `joinstyle` attributes added to `Collection`

The *`Collection`* class now has customizable `capstyle` and `joinstyle` attributes. This allows the user for example to set the `capstyle` of errorbars.

8.4.10 *pad* kwarg added to ax.set_title

The method *`Axes.set_title`* now has a *pad* kwarg, that specifies the distance from the top of an axes to where the title is drawn. The units of *pad* is points, and the default is the value of the (already-existing) `rcParams["axes.titlepad"]` (default: `6.0`).

8.4.11 Comparison of 2 colors in Matplotlib

As the colors in Matplotlib can be specified with a wide variety of ways, the *`matplotlib.colors.same_color`* method has been added which checks if two *`colors`* are the same.

8.4.12 Autoscaling a polar plot snaps to the origin

Setting the limits automatically in a polar plot now snaps the radial limit to zero if the automatic limit is nearby. This means plotting from zero doesn't automatically scale to include small negative values on the radial axis.

The limits can still be set manually in the usual way using *`set_ylim`*.

8.4.13 PathLike support

On Python 3.6+, *`savefig`*, *`imsave`*, *`imread`*, and animation writers now accept `os.PathLike`s as input.

8.4.14 `Axes.tick_params` can set gridline properties

`Tick` objects hold gridlines as well as the tick mark and its label. *`Axis.set_tick_params`*, *`Axes.tick_params`* and *`pyplot.tick_params`* now have keyword arguments 'grid_color', 'grid_alpha', 'grid_linewidth', and 'grid_linestyle' for overriding the defaults in *`rcParams`*: 'grid.color', etc.

8.4.15 `Axes.imshow` clips RGB values to the valid range

When *`Axes.imshow`* is passed an RGB or RGBA value with out-of-range values, it now logs a warning and clips them to the valid range. The old behaviour, wrapping back in to the range, often hid outliers and made interpreting RGB images unreliable.

8.4.16 Properties in `matplotlibrc` to place xaxis and yaxis tick labels

Introducing four new boolean properties in `matplotlibrc` for default positions of xaxis and yaxis tick labels, namely, `rcParams["xtick.labeltop"]` (default: `False`), `rcParams["xtick.labelbottom"]` (default: `True`), `rcParams["ytick.labelright"]` (default: `False`) and `rcParams["ytick.labelleft"]` (default: `True`). These can also be changed in rcParams.

8.4.17 PGI bindings for gtk3

The GTK3 backends can now use PGI instead of PyGObject. PGI is a fairly incomplete binding for GObject, thus its use is not recommended; its main benefit is its availability on Travis (thus allowing CI testing for the gtk3agg and gtk3cairo backends).

The binding selection rules are as follows: - if `gi` has already been imported, use it; else - if `pgi` has already been imported, use it; else - if `gi` can be imported, use it; else - if `pgi` can be imported, use it; else - error out.

Thus, to force usage of PGI when both bindings are installed, import it first.

8.4.18 Cairo rendering for Qt, WX, and Tk canvases

The new `Qt4Cairo`, `Qt5Cairo`, `WXCairo`, and `TkCairo` backends allow Qt, Wx, and Tk canvases to use Cairo rendering instead of Agg.

8.4.19 Added support for QT in new ToolManager

Now it is possible to use the ToolManager with Qt5 For example

import matplotlib

matplotlib.use('QT5AGG') matplotlib.rcParams['toolbar'] = 'toolmanager' import matplotlib.pyplot as plt

plt.plot([1,2,3]) plt.show()

Treat the new Tool classes experimental for now, the API will likely change and perhaps the rcParam as well

The main example /gallery/user_interfaces/toolmanager_sgskip shows more details, just adjust the header to use QT instead of GTK3

8.4.20 TkAgg backend reworked to support PyPy

PyPy can now plot using the TkAgg backend, supported on PyPy 5.9 and greater (both PyPy for python 2.7 and PyPy for python 3.5).

8.4.21 Python logging library used for debug output

Matplotlib has in the past (sporadically) used an internal verbose-output reporter. This version converts those calls to using the standard python `logging` library.

Support for the old *rcParams* `verbose.level` and `verbose.fileo` is dropped.

The command-line options `--verbose-helpful` and `--verbose-debug` are still accepted, but deprecated. They are now equivalent to setting `logging.INFO` and `logging.DEBUG`.

The logger's root name is `matplotlib` and can be accessed from programs as:

```
import logging
mlog = logging.getLogger('matplotlib')
```

Instructions for basic usage are in *Troubleshooting* and for developers in *Contributing*.

8.4.22 Improved `repr` for `Transforms`

*Transform*s now indent their `repr`s in a more legible manner:

```
In [1]: l, = plt.plot([]); l.get_transform()
Out[1]:
CompositeGenericTransform(
   TransformWrapper(
      BlendedAffine2D(
```

(continues on next page)

(continued from previous page)

```
           IdentityTransform(),
           IdentityTransform())),
   CompositeGenericTransform(
      BboxTransformFrom(
           TransformedBbox(
              Bbox(x0=-0.05500000000000001, y0=-0.05500000000000001, x1=0.
↪05500000000000001, y1=0.05500000000000001),
              TransformWrapper(
                 BlendedAffine2D(
                       IdentityTransform(),
                       IdentityTransform()))),
      BboxTransformTo(
           TransformedBbox(
              Bbox(x0=0.125, y0=0.10999999999999999, x1=0.9, y1=0.88),
              BboxTransformTo(
                 TransformedBbox(
                       Bbox(x0=0.0, y0=0.0, x1=6.4, y1=4.8),
                       Affine2D(
                          [[ 100.    0.    0.]
                          [   0.  100.    0.]
                          [   0.    0.    1.]]))))))))
```

8.5 New in Matplotlib 2.1.0

8.5.1 Documentation

The examples have been migrated to use sphinx gallery. This allows better mixing of prose and code in the examples, provides links to download the examples as both a Python script and a Jupyter notebook, and improves the thumbnail galleries. The examples have been re-organized into *Tutorials* and a gallery.

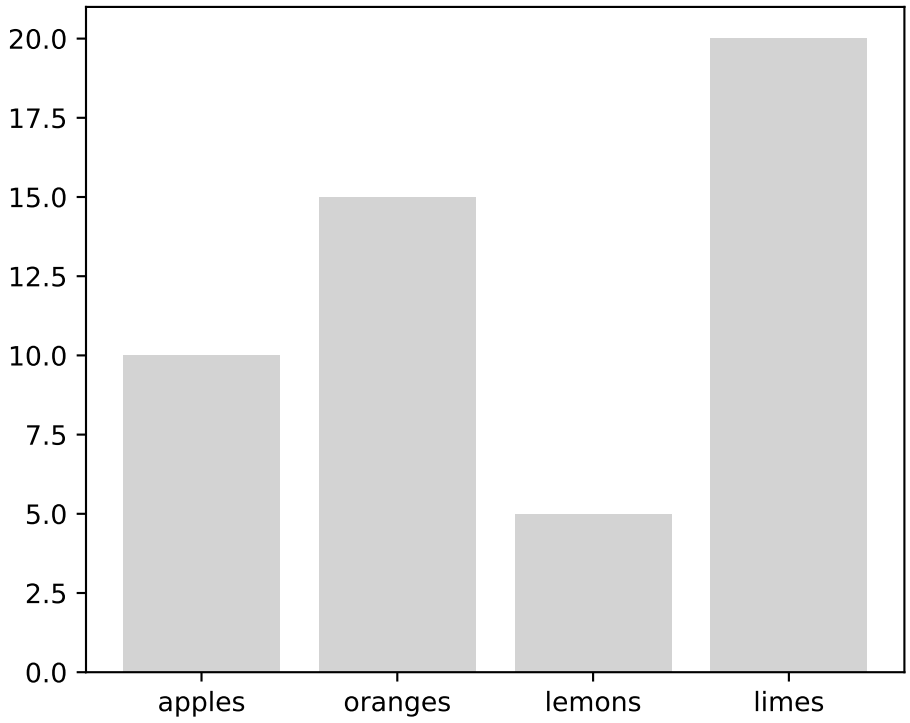
Many docstrings and examples have been clarified and improved.

8.5.2 New features

String categorical values

All plotting functions now support string categorical values as input. For example:

```
data = {'apples': 10, 'oranges': 15, 'lemons': 5, 'limes': 20}
fig, ax = plt.subplots()
ax.bar(data.keys(), data.values(), color='lightgray')
```



Interactive JS widgets for animation

Jake Vanderplas' JSAnimation package has been merged into Matplotlib. This adds to Matplotlib the *HTMLWriter* class for generating a JavaScript HTML animation, suitable for the IPython notebook. This can be activated by default by setting the `animation.html` rc parameter to `jshtml`. One can also call the *to_jshtml* method to manually convert an animation. This can be displayed using IPython's `HTML` display class:

```
from IPython.display import HTML
HTML(animation.to_jshtml())
```

The *HTMLWriter* class can also be used to generate an HTML file by asking for the `html` writer.

Enhancements to polar plot

The polar axes transforms have been greatly re-factored to allow for more customization of view limits and tick labelling. Additional options for view limits allow for creating an annulus, a sector, or some combination of the two.

The *set_rorigin()* method may be used to provide an offset to the minimum plotting radius, producing an annulus.

The *set_theta_zero_location()* method now has an optional `offset` argument. This argument may be used to further specify the zero location based on the given anchor point.

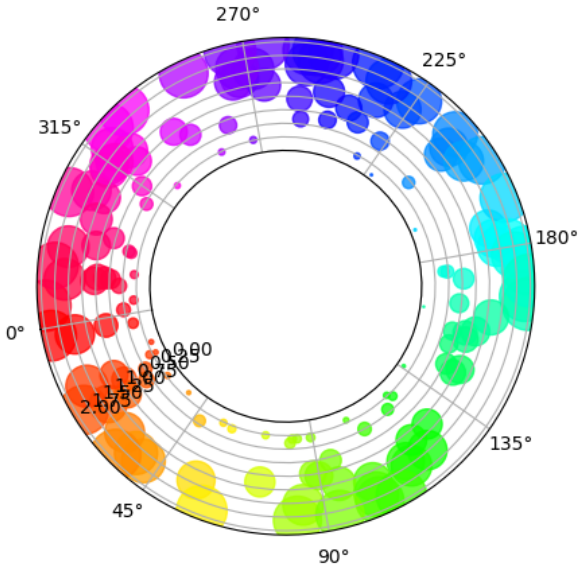


Fig. 1: Polar Offset Demo

The *set_thetamin()* and *set_thetamax()* methods may be used to limit the range of angles plotted, producing sectors of a circle.

Previous releases allowed plots containing negative radii for which the negative values are simply used as labels, and the real radius is shifted by the configured minimum. This release also allows negative radii to be used for grids and ticks, which were previously silently ignored.

Radial ticks have been modified to be parallel to the circular grid line, and angular ticks have been modified to be parallel to the grid line. It may also be useful to rotate tick *labels* to match the boundary. Calling `ax.tick_params(rotation='auto')` will enable the new behavior: radial tick labels will be parallel to the circular grid line, and angular tick labels will be perpendicular to the grid line (i.e., parallel to the outer boundary). Additionally, tick labels now obey the padding settings that previously only worked on Cartesian plots. Consequently, the `frac` argument to *PolarAxes.set_thetagrids* is no longer applied. Tick padding can be modified with the `pad` argu-

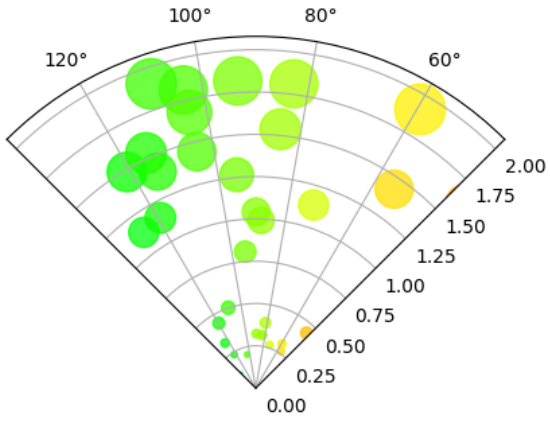


Fig. 2: Polar Sector Demo

ment to *Axes.tick_params* or *Axis.set_tick_params*.

`Figure` class now has `subplots` method

The *Figure* class now has a *subplots()* method which behaves the same as *pyplot.subplots()* but on an existing figure.

Metadata savefig keyword argument

savefig() now accepts `metadata` as a keyword argument. It can be used to store key/value pairs in the image metadata.

- 'png' with Agg backend
- 'pdf' with PDF backend (see *writeInfoDict()* for a list of supported keywords)
- 'eps' and 'ps' with PS backend (only 'Creator' key is accepted)

```
plt.savefig('test.png', metadata={'Software': 'My awesome software'})
```

Busy Cursor

The interactive GUI backends will now change the cursor to busy when Matplotlib is rendering the canvas.

PolygonSelector

A *PolygonSelector* class has been added to *matplotlib.widgets*. See /gallery/widgets/polygon_selector_demo for details.

Added `matplotlib.ticker.PercentFormatter`

The new *PercentFormatter* formatter has some nice features like being able to convert from arbitrary data scales to percents, a customizable percent symbol and either automatic or manual control over the decimal points.

Reproducible PS, PDF and SVG output

The `SOURCE_DATE_EPOCH` environment variable can now be used to set the timestamp value in the PS and PDF outputs. See source date epoch.

Alternatively, calling `savefig` with `metadata={'CreationDate': None}` will omit the timestamp altogether for the PDF backend.

The reproducibility of the output from the PS and PDF backends has so far been tested using various plot elements but only default values of options such as `{ps, pdf}.fonttype` that can affect the output at a low level, and not with the mathtext or usetex features. When Matplotlib calls external tools (such as PS distillers or LaTeX) their versions need to be kept constant for reproducibility, and they may add sources of nondeterminism outside the control of Matplotlib.

For SVG output, the `svg.hashsalt` rc parameter has been added in an earlier release. This parameter changes some random identifiers in the SVG file to be deterministic. The downside of this setting is that if more than one file is generated using deterministic identifiers and they end up as parts of one larger document, the identifiers can collide and cause the different parts to affect each other.

These features are now enabled in the tests for the PDF and SVG backends, so most test output files (but not all of them) are now deterministic.

Orthographic projection for mplot3d

Axes3D now accepts `proj_type` keyword argument and has a method *set_proj_type()*. The default option is `'persp'` as before, and supplying `'ortho'` enables orthographic view.

Compare the z-axis which is vertical in orthographic view, but slightly skewed in the perspective view.

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure(figsize=(4, 6))
ax1 = fig.add_subplot(2, 1, 1, projection='3d')
ax1.set_proj_type('persp')
ax1.set_title('Perspective (default)')

ax2 = fig.add_subplot(2, 1, 2, projection='3d')
ax2.set_proj_type('ortho')
ax2.set_title('Orthographic')

plt.show()
```

`voxels` function for mplot3d

Axes3D now has a *voxels* method, for visualizing boolean 3D data. Uses could include plotting a sparse 3D heat map, or visualizing a volumetric model.

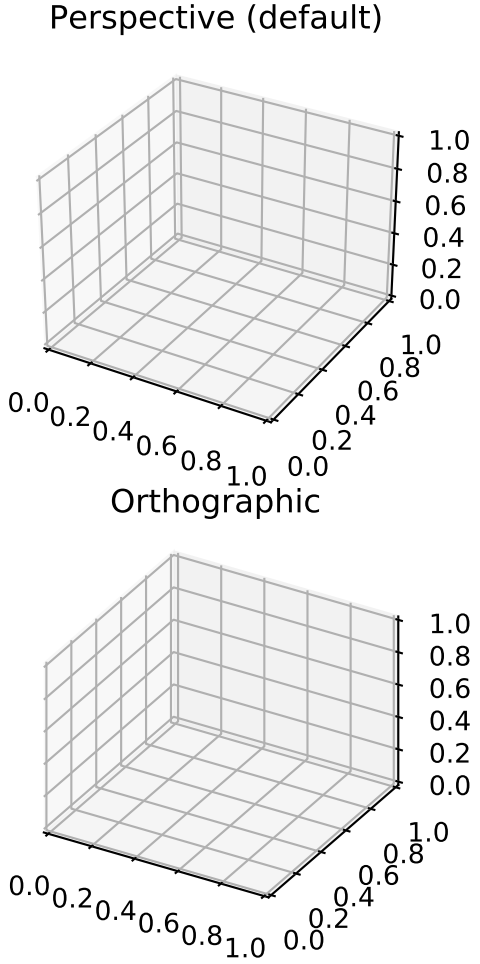
8.5.3 Improvements

CheckButtons widget `get_status` function

A *get_status()* method has been added to the *matplotlib.widgets.CheckButtons* class. This `get_status` method allows user to query the status (True/False) of all of the buttons in the `CheckButtons` object.

Add `fill_bar` argument to `AnchoredSizeBar`

The `mpl_toolkits` class *AnchoredSizeBar* now has an additional `fill_bar` argument, which makes the size bar a solid rectangle instead of just drawing the border of the rectangle. The default is `None`, and whether or not the bar will be filled by default depends on the value of `size_vertical`. If `size_vertical` is nonzero, `fill_bar` will be set to `True`. If `size_vertical` is zero then `fill_bar` will be set to `False`. If you wish to override this default behavior, set `fill_bar` to `True` or `False` to unconditionally always or never use a filled patch rectangle for the size bar.


Perspective (default)
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
Orthographic
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0

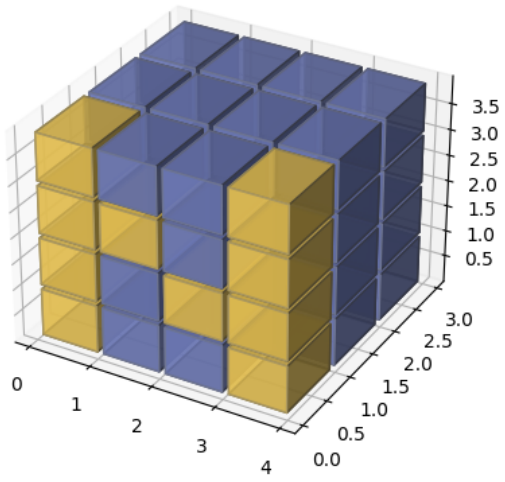
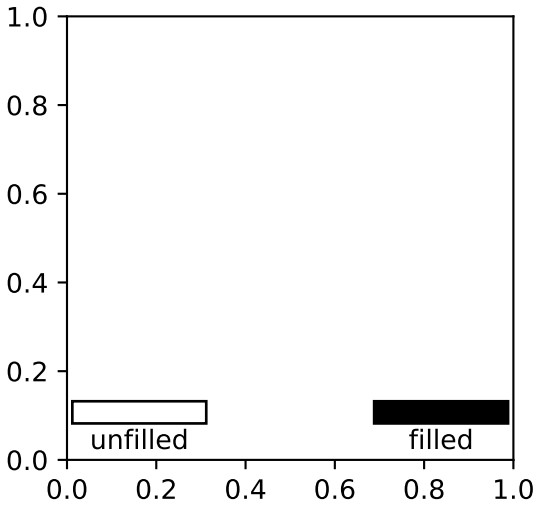


Fig. 3: Voxel Demo

```
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1.anchored_artists import AnchoredSizeBar

fig, ax = plt.subplots(figsize=(3, 3))

bar0 = AnchoredSizeBar(ax.transData, 0.3, 'unfilled', loc='lower left',
                       frameon=False, size_vertical=0.05, fill_bar=False)
ax.add_artist(bar0)
bar1 = AnchoredSizeBar(ax.transData, 0.3, 'filled', loc='lower right',
                       frameon=False, size_vertical=0.05, fill_bar=True)
ax.add_artist(bar1)

plt.show()
```

Annotation can use a default arrow style

Annotations now use the default arrow style when setting `arrowprops={}`, rather than no arrow (the new behavior actually matches the documentation).



Barbs and Quiver Support Dates

When using the *quiver()* and *barbs()* plotting methods, it is now possible to pass dates, just like for other methods like *plot()*. This also allows these functions to handle values that need unit-conversion applied.

Hexbin default line color

The default `linecolor` keyword argument for *hexbin()* is now `'face'`, and supplying `'none'` now prevents lines from being drawn around the hexagons.

Figure.legend() can be called without arguments

Calling *Figure.legend()* can now be done with no arguments. In this case a legend will be created that contains all the artists on all the axes contained within the figure.

Multiple legend keys for legend entries

A legend entry can now contain more than one legend key. The extended *HandlerTuple* class now accepts two parameters: `ndivide` divides the legend area in the specified number of sections; `pad` changes the padding between the legend keys.

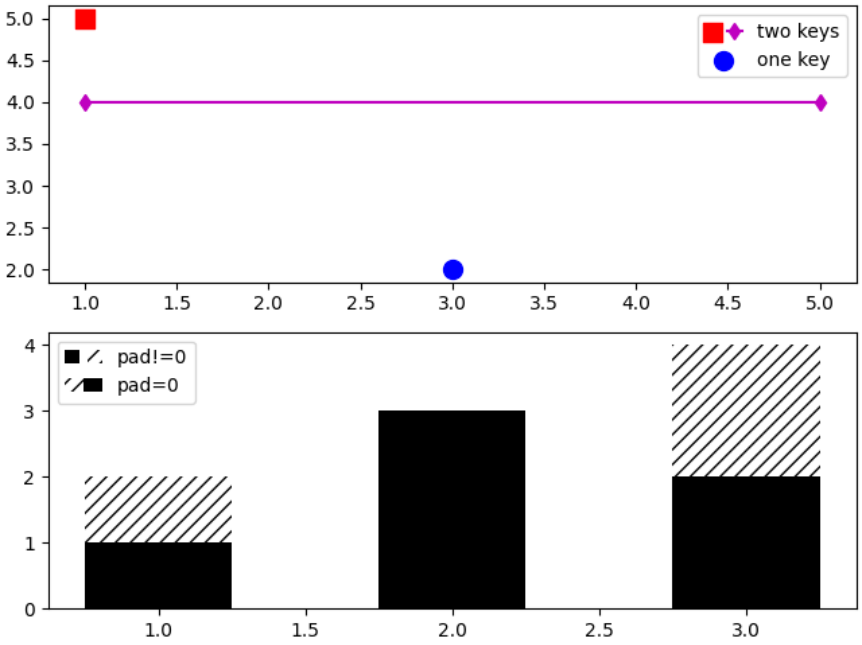


Fig. 4: Multiple Legend Keys

New parameter *clear* for `figure()`

When the pyplot's function *figure()* is called with a `num` parameter, a new window is only created if no existing window with the same value exists. A new bool parameter *clear* was added for explicitly clearing its existing contents. This is particularly useful when utilized in interactive sessions. Since *subplots()* also accepts keyword arguments from *figure()*, it can also be used there:

```
import matplotlib.pyplot as plt

fig0 = plt.figure(num=1)
fig0.suptitle("A fancy plot")
print("fig0.texts: ", [t.get_text() for t in fig0.texts])

fig1 = plt.figure(num=1, clear=False)  # do not clear contents of window
fig1.text(0.5, 0.5, "Really fancy!")
print("fig0 is fig1: ",  fig0 is fig1)
print("fig1.texts: ", [t.get_text() for t in fig1.texts])

fig2, ax2 = plt.subplots(2, 1, num=1, clear=True)  # clear contents
print("fig0 is fig2: ",  fig0 is fig2)
print("fig2.texts: ", [t.get_text() for t in fig2.texts])

# The output:
# fig0.texts:  ['A fancy plot']
# fig0 is fig1:  True
# fig1.texts:  ['A fancy plot', 'Really fancy!']
# fig0 is fig2:  True
```

(continues on next page)

(continued from previous page)

```
# fig2.texts:  []
```

Specify minimum value to format as scalar for `LogFormatterMathtext`

`LogFormatterMathtext` now includes the option to specify a minimum value exponent to format as a scalar (i.e., 0.001 instead of 10^{-3}).

New quiverkey angle keyword argument

Plotting a *`quiverkey()`* now admits the `angle` keyword argument, which sets the angle at which to draw the key arrow.

Colormap reversed method

The methods *`matplotlib.colors.LinearSegmentedColormap.reversed()`* and *`matplotlib.colors.ListedColormap.reversed()`* return a reversed instance of the Colormap. This implements a way for any Colormap to be reversed.

`artist.setp` (and `pyplot.setp`) accept a *file* argument

The argument is keyword-only. It allows an output file other than `sys.stdout` to be specified. It works exactly like the *file* argument to `print`.

`streamplot` streamline generation more configurable

The starting point, direction, and length of the stream lines can now be configured. This allows to follow the vector field for a longer time and can enhance the visibility of the flow pattern in some use cases.

`Axis.set_tick_params` now responds to `rotation`

Bulk setting of tick label rotation is now possible via *`tick_params()`* using the `rotation` keyword.

```
ax.tick_params(which='both', rotation=90)
```

Ticklabels are turned off instead of being invisible

Internally, the *Tick*'s `label1On()` attribute is now used to hide tick labels instead of setting the visibility on the tick label objects. This improves overall performance and fixes some issues. As a consequence, in case those labels ought to be shown, *tick_params()* needs to be used, e.g.

```
ax.tick_params(labelbottom=True)
```

Shading in 3D bar plots

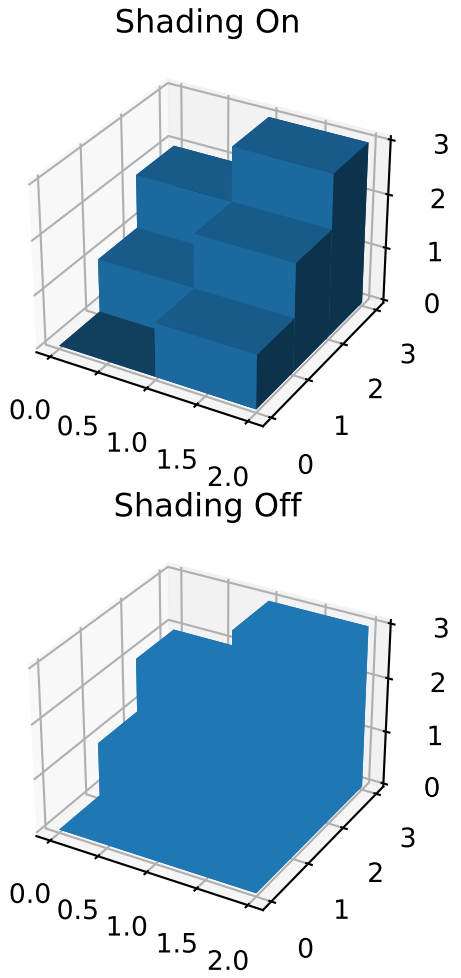
A new `shade` parameter has been added the 3D *bar* plotting method. The default behavior remains to shade the bars, but now users have the option of setting `shade` to `False`.

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

x = np.arange(2)
y = np.arange(3)
x2d, y2d = np.meshgrid(x, y)
x, y = x2d.ravel(), y2d.ravel()
z = np.zeros_like(x)
dz = x + y

fig = plt.figure(figsize=(4, 6))
ax1 = fig.add_subplot(2, 1, 1, projection='3d')
ax1.bar3d(x, y, z, 1, 1, dz, shade=True)
ax1.set_title('Shading On')

ax2 = fig.add_subplot(2, 1, 2, projection='3d')
ax2.bar3d(x, y, z, 1, 1, dz, shade=False)
ax2.set_title('Shading Off')

plt.show()
```

New `which` Parameter for `autofmt_xdate`

A `which` parameter now exists for the method *autofmt_xdate()*. This allows a user to format `major`, `minor` or `both` tick labels selectively. The default behavior will rotate and align the `major` tick labels.

```
fig.autofmt_xdate(bottom=0.2, rotation=30, ha='right', which='minor')
```


Shading On
3
2
1
0
3
2
1
0
0.0
0.5
1.0
1.5
2.0
Shading Off
3
2
1
0
3
2
1
0
0.0
0.5
1.0
1.5
2.0

New Figure Parameter for `subplot2grid`

A `fig` parameter now exists for the function *subplot2grid()*. This allows a user to specify the figure where the subplots will be created. If `fig` is `None` (default) then the method will use the current figure retrieved by *gcf()*.

```
subplot2grid(shape, loc, rowspan=1, colspan=1, fig=myfig)
```

Interpolation in `fill_betweenx`

The `interpolate` parameter now exists for the method *fill_betweenx()*. This allows a user to interpolate the data and fill the areas in the crossover points, similarly to *fill_between()*.

New keyword argument `sep` for EngFormatter

A new `sep` keyword argument has been added to *EngFormatter* and provides a means to define the string that will be used between the value and its unit. The default string is `" "`, which preserves the former behavior. Additionally, the separator is now present between the value and its unit even in the absence of SI prefix. There was formerly a bug that was causing strings like `"3.14V"` to be returned instead of the expected `"3.14 V"` (with the default behavior).

Extend `MATPLOTLIBRC` behavior

The environmental variable can now specify the full file path or the path to a directory containing a `matplotlibrc` file.

`density` kwarg to hist

The *hist()* method now prefers `density` to `normed` to control if the histogram should be normalized, following a change upstream to NumPy. This will reduce confusion as the behavior has always been that the integral of the histogram is 1 (rather than sum or maximum value).

8.5.4 Internals

New TransformedPatchPath caching object

A newly added *TransformedPatchPath* provides a means to transform a *Patch* into a *Path* via a *Transform* while caching the resulting path. If neither the patch nor the transform have changed, a cached copy of the path is returned.

This class differs from the older *TransformedPath* in that it is able to refresh itself based on the underlying patch while the older class uses an immutable path.

Abstract base class for movie writers

The new *AbstractMovieWriter* class defines the API required by a class that is to be used as the `writer` in the *matplotlib.animation.Animation.save()* method. The existing *MovieWriter* class now derives from the new abstract base class.

Stricter validation of line style rcParams

The validation of rcParams that are related to line styles (`lines.linestyle`, `boxplot.*.linestyle`, `grid.linestyle` and `contour.negative_linestyle`) now effectively checks that the values are valid line styles. Strings like `'dashed'` or `'--'` are accepted, as well as even-length sequences of on-off ink like `[1, 1.65]`. In this latter case, the offset value is handled internally and should *not* be provided by the user.

The new validation scheme replaces the former one used for the `contour.negative_linestyle` rcParams, that was limited to `'solid'` and `'dashed'` line styles.

The validation is case-insensitive. The following are now valid:

```
grid.linestyle             : (1, 3)   # loosely dotted grid lines
contour.negative_linestyle : dashdot  # previously only solid or dashed
```

pytest

The automated tests have been switched from nose to pytest.

8.5.5 Performance

Path simplification updates

Line simplification controlled by the `path.simplify` and `path.simplify_threshold` parameters has been improved. You should notice better rendering performance when plotting large amounts of data (as long as the above parameters are set accordingly). Only the line segment portion of paths will be simplified -- if you are also drawing markers and experiencing problems with rendering speed, you should consider using the `markevery` option to *plot*. See the *Performance* section in the usage tutorial for more information.

The simplification works by iteratively merging line segments into a single vector until the next line segment's perpendicular distance to the vector (measured in display-coordinate space) is greater than the `path.simplify_threshold` parameter. Thus, higher values of `path.simplify_threshold` result in quicker rendering times. If you are plotting just to explore data and not for publication quality, pixel perfect plots, then a value of `1.0` can be safely used. If you want to make sure your plot reflects your data *exactly*, then you should set `path.simplify` to false and/or `path.simplify_threshold` to `0`. Matplotlib currently defaults to a conservative value of `1/9`, smaller values are unlikely to cause any visible differences in your plots.

Implement intersects_bbox in c++

intersects_bbox() has been implemented in c++ which improves the performance of automatically placing the legend.

8.6 New in matplotlib 2.0

Note: matplotlib 2.0 supports Python 2.7, and 3.4+

8.6.1 Default style changes

The major changes in v2.0 are related to overhauling the default styles.

Changes to the default style

The most important changes in matplotlib 2.0 are the changes to the default style.

While it is impossible to select the best default for all cases, these are designed to work well in the most common cases.

A 'classic' style sheet is provided so reverting to the 1.x default values is a single line of python

```
import matplotlib.style
import matplotlib as mpl
mpl.style.use('classic')
```

See *The matplotlibrc file* for details about how to persistently and selectively revert many of these changes.

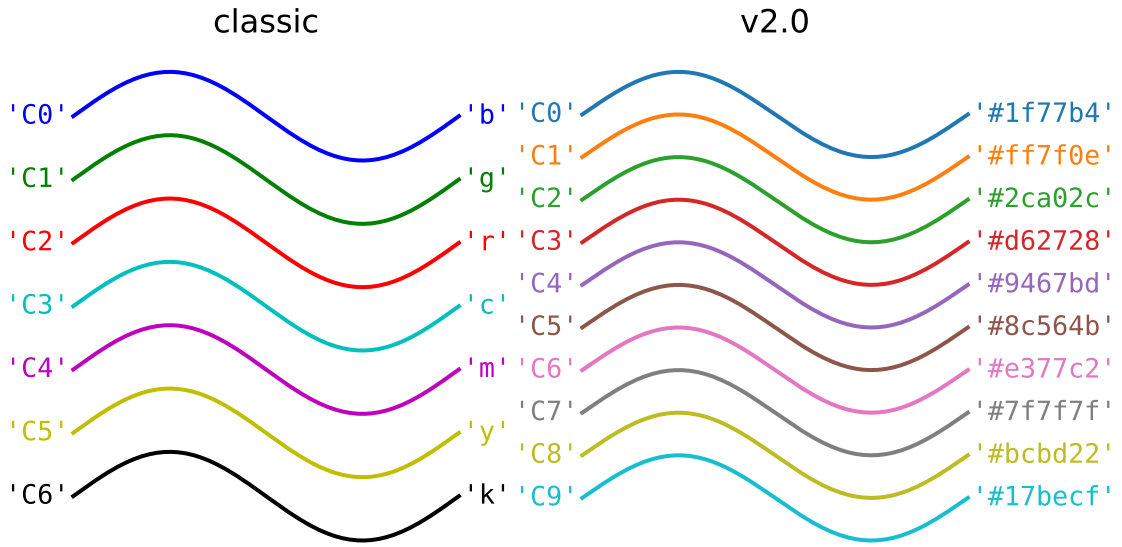
Table of Contents

- *Legends*
- *Image*
  - *Interpolation*
  - *Colormapping pipeline*
  - *Shading*
- *Plot layout*
  - *Auto limits*
  - *Z-order*
  - *Ticks*
  - *Tick label formatting*
- *mplot3d*

Colors, color cycles, and color maps

Colors in default property cycle

The colors in the default property cycle have been changed from `['b', 'g', 'r', 'c', 'm', 'y', 'k']` to the category10 color palette used by Vega and d3 originally developed at Tableau.



In addition to changing the colors, an additional method to specify colors was added. Previously, the default colors were the single character short-hand notations for red, green, blue, cyan, magenta, yellow, and black. This made them easy to type and usable in the abbreviated style string in `plot`, however the

new default colors are only specified via hex values. To access these colors outside of the property cycling the notation for colors `'CN'`, where `N` takes values 0-9, was added to denote the first 10 colors in `rcParams["axes.prop_cycle"]` (default: `cycler('color', ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728', '#9467bd', '#8c564b', '#e377c2', '#7f7f7f', '#bcbd22', '#17becf'])`). See *Specifying Colors* for more details.

To restore the old color cycle use

```
from cycler import cycler
mpl.rcParams['axes.prop_cycle'] = cycler(color='bgrcmyk')
```

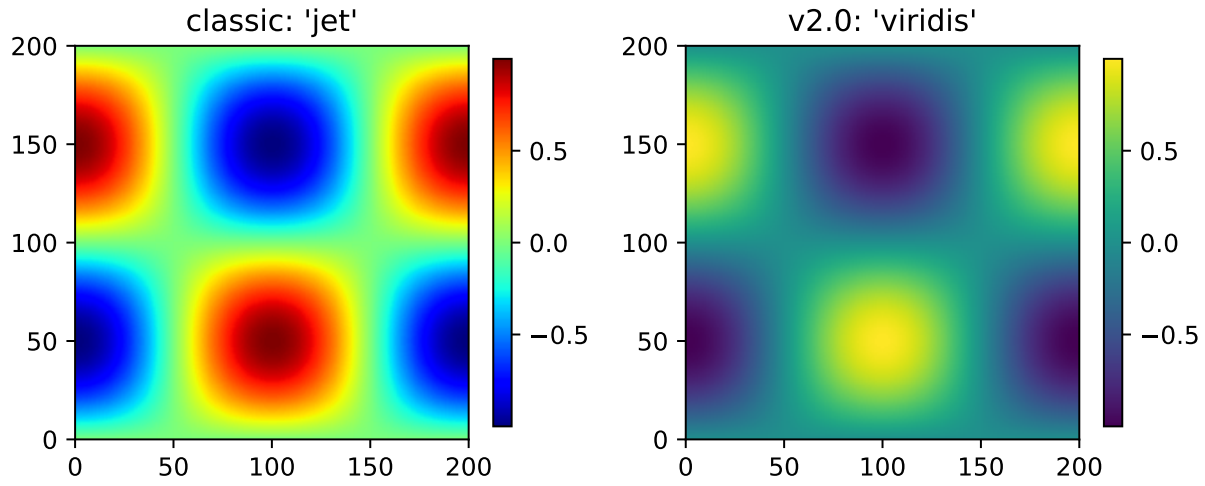
or set

```
axes.prop_cycle    : cycler('color', 'bgrcmyk')
```

in your `matplotlibrc` file.

Colormap

The new default color map used by *matplotlib.cm.ScalarMappable* instances is 'viridis' (aka option D).



For an introduction to color theory and how 'viridis' was generated watch Nathaniel Smith and Stéfan van der Walt's talk from SciPy2015. See here for many more details about the other alternatives and the tools used to create the color map. For details on all of the color maps available in matplotlib see *Choosing Colormaps in Matplotlib*.

The previous default can be restored using

```
mpl.rcParams['image.cmap'] = 'jet'
```

or setting

```
image.cmap    : 'jet'
```

in your `matplotlibrc` file; however this is strongly discouraged.

Interactive figures

The default interactive figure background color has changed from grey to white, which matches the default background color used when saving.

The previous defaults can be restored by

```
mpl.rcParams['figure.facecolor'] = '0.75'
```

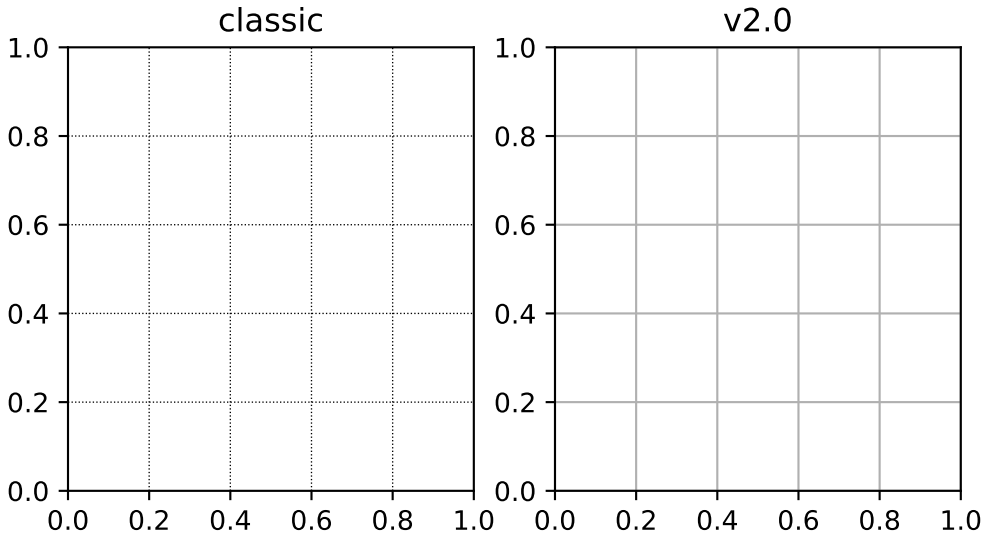
or by setting

```
figure.facecolor : '0.75'
```

in your `matplotlibrc` file.

Grid lines

The default style of grid lines was changed from black dashed lines to thicker solid light grey lines.



The previous default can be restored by using:

```
mpl.rcParams['grid.color'] = 'k'
mpl.rcParams['grid.linestyle'] = ':'
mpl.rcParams['grid.linewidth'] = 0.5
```

or by setting:

```
grid.color       :   k       # grid color
grid.linestyle   :   :       # dotted
grid.linewidth   :   0.5     # in points
```

in your `matplotlibrc` file.

Figure size, font size, and screen dpi

The default dpi used for on-screen display was changed from 80 dpi to 100 dpi, the same as the default dpi for saving files. Due to this change, the on-screen display is now more what-you-see-is-what-you-get for saved files. To keep the figure the same size in terms of pixels, in order to maintain approximately the same size on the screen, the default figure size was reduced from 8x6 inches to 6.4x4.8 inches. As a consequence of this the default font sizes used for the title, tick labels, and axes labels were reduced to maintain their size relative to the overall size of the figure. By default the dpi of the saved image is now the dpi of the *Figure* instance being saved.

This will have consequences if you are trying to match text in a figure directly with external text.

The previous defaults can be restored by

```
mpl.rcParams['figure.figsize'] = [8.0, 6.0]
mpl.rcParams['figure.dpi'] = 80
mpl.rcParams['savefig.dpi'] = 100

mpl.rcParams['font.size'] = 12
mpl.rcParams['legend.fontsize'] = 'large'
mpl.rcParams['figure.titlesize'] = 'medium'
```

or by setting:

```
figure.figsize   : [8.0, 6.0]
figure.dpi       : 80
savefig.dpi      : 100

font.size        : 12.0
legend.fontsize  : 'large'
figure.titlesize : 'medium'
```

In your `matplotlibrc` file.

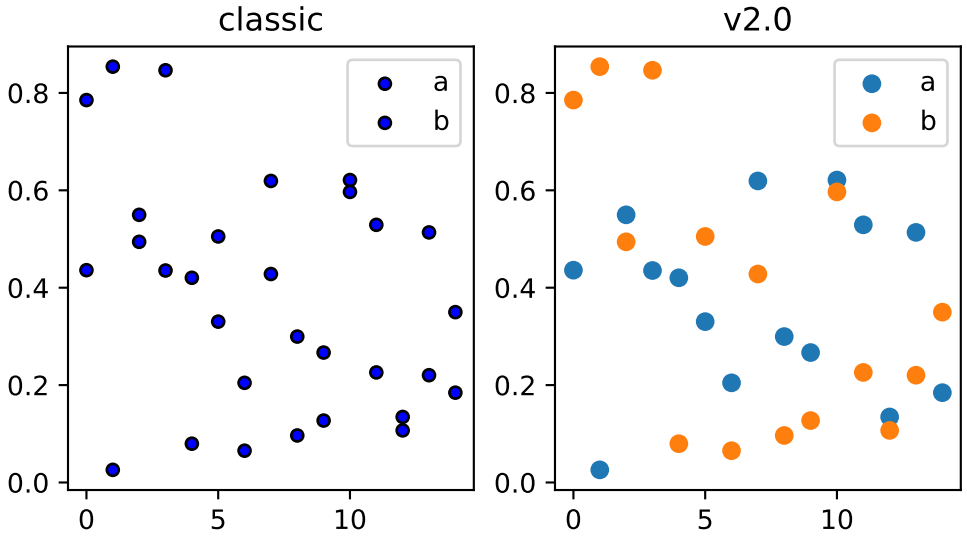
In addition, the `forward` kwarg to `set_size_inches` now defaults to `True` to improve the interactive experience. Backend canvases that adjust the size of their bound *matplotlib.figure.Figure* must pass `forward=False` to avoid circular behavior. This default is not configurable.

Plotting functions

scatter

The following changes were made to the default behavior of *scatter*

- The default size of the elements in a scatter plot is now based on `rcParams["lines.markersize"]` (default: `6.0`) so it is consistent with `plot(X, Y, 'o')`. The old value was 20, and the new value is 36 (6^2).
- Scatter markers no longer have a black edge.
- If the color of the markers is not specified it will follow the property cycle, pulling from the 'patches' cycle on the `Axes`.



The classic default behavior of *scatter* can only be recovered through `mpl.style.use('classic')`. The marker size can be recovered via

```
mpl.rcParam['lines.markersize'] = np.sqrt(20)
```

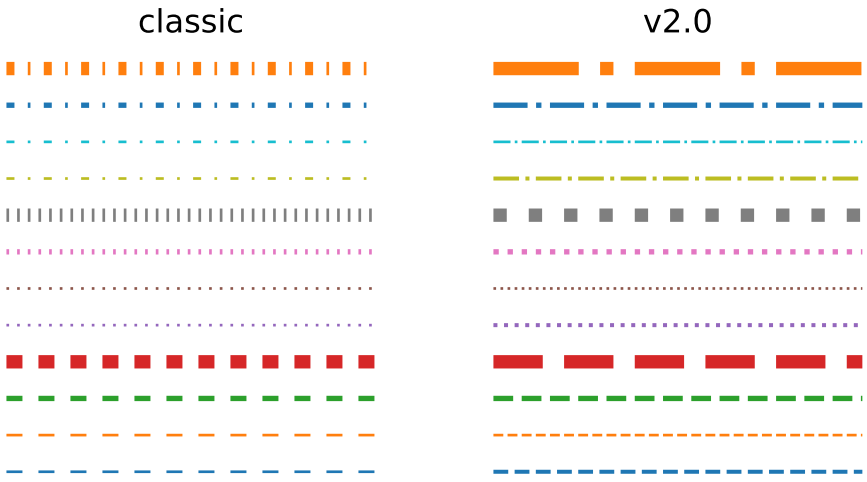
however, this will also affect the default marker size of *plot*. To recover the classic behavior on a per-call basis pass the following kwargs:

```
classic_kwargs = {'s': 20, 'edgecolors': 'k', 'c': 'b'}
```

`plot`

The following changes were made to the default behavior of *plot*

- the default linewidth increased from 1 to 1.5
- the dash patterns associated with `'--'`, `':'`, and `'-.'` have changed
- the dash patterns now scale with line width



The previous defaults can be restored by setting:

```
mpl.rcParams['lines.linewidth'] = 1.0
mpl.rcParams['lines.dashed_pattern'] = [6, 6]
mpl.rcParams['lines.dashdot_pattern'] = [3, 5, 1, 5]
mpl.rcParams['lines.dotted_pattern'] = [1, 3]
mpl.rcParams['lines.scale_dashes'] = False
```

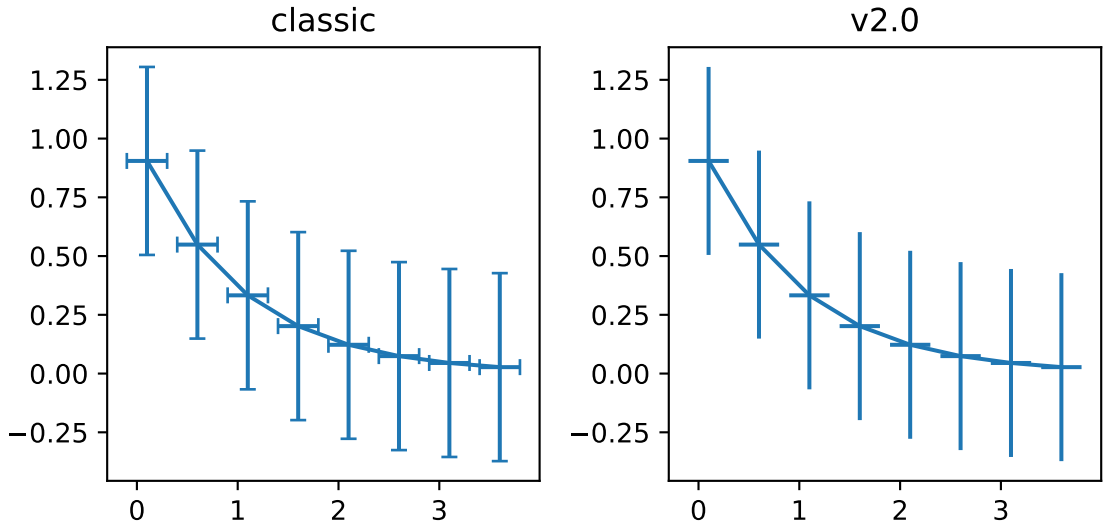
or by setting:

```
lines.linewidth   : 1.0
lines.dashed_pattern : 6, 6
lines.dashdot_pattern : 3, 5, 1, 5
lines.dotted_pattern : 1, 3
lines.scale_dashes: False
```

in your `matplotlibrc` file.

errorbar

By default, caps on the ends of errorbars are not present.



This also changes the return value of *errorbar()* as the list of 'caplines' will be empty by default.

The previous defaults can be restored by setting:

```
mpl.rcParams['errorbar.capsize'] = 3
```

or by setting

```
errorbar.capsize : 3
```

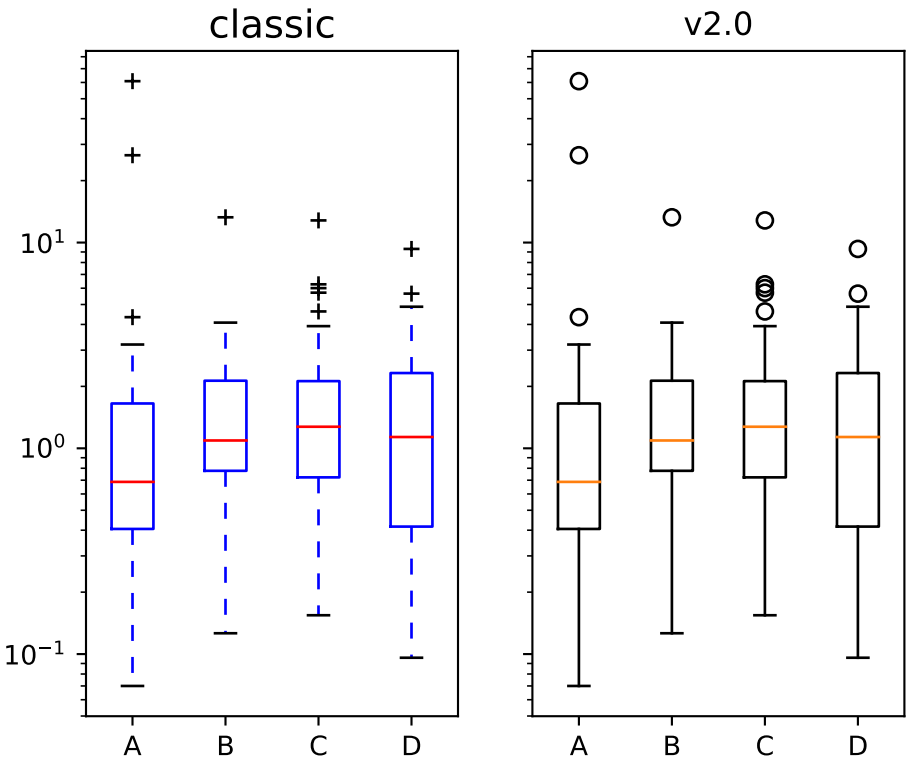
in your `matplotlibrc` file.

boxplot

Previously, boxplots were composed of a mish-mash of styles that were, for better for worse, inherited from Matlab. Most of the elements were blue, but the medians were red. The fliers (outliers) were black plus-symbols ('+') and the whiskers were dashed lines, which created ambiguity if the (solid and black) caps were not drawn.

For the new defaults, everything is black except for the median and mean lines (if drawn), which are set to the first two elements of the current color cycle. Also, the default flier markers are now hollow circles, which maintain the ability of the plus-symbols to overlap without obscuring data too much.

The previous defaults can be restored by setting:




```
mpl.rcParams['boxplot.flierprops.color'] = 'k'
mpl.rcParams['boxplot.flierprops.marker'] = '+'
mpl.rcParams['boxplot.flierprops.markerfacecolor'] = 'none'
mpl.rcParams['boxplot.flierprops.markeredgecolor'] = 'k'
mpl.rcParams['boxplot.boxprops.color'] = 'b'
mpl.rcParams['boxplot.whiskerprops.color'] = 'b'
mpl.rcParams['boxplot.whiskerprops.linestyle'] = '--'
mpl.rcParams['boxplot.medianprops.color'] = 'r'
mpl.rcParams['boxplot.meanprops.color'] = 'r'
mpl.rcParams['boxplot.meanprops.marker'] = '^'
mpl.rcParams['boxplot.meanprops.markerfacecolor'] = 'r'
mpl.rcParams['boxplot.meanprops.markeredgecolor'] = 'k'
mpl.rcParams['boxplot.meanprops.markersize'] = 6
mpl.rcParams['boxplot.meanprops.linestyle'] = '--'
mpl.rcParams['boxplot.meanprops.linewidth'] = 1.0
```

or by setting:

```
boxplot.flierprops.color:           'k'
boxplot.flierprops.marker:          '+'
boxplot.flierprops.markerfacecolor: 'none'
boxplot.flierprops.markeredgecolor: 'k'
boxplot.boxprops.color:             'b'
```

(continues on next page)

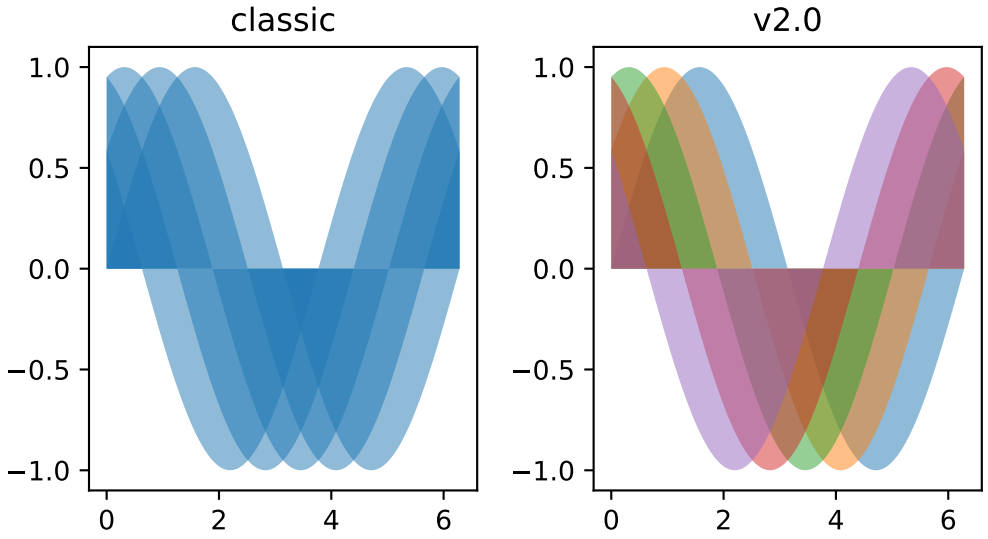
(continued from previous page)

```
boxplot.whiskerprops.color:          'b'
boxplot.whiskerprops.linestyle:      '--'
boxplot.medianprops.color:           'r'
boxplot.meanprops.color:             'r'
boxplot.meanprops.marker:            '^'
boxplot.meanprops.markerfacecolor:   'r'
boxplot.meanprops.markeredgecolor:   'k'
boxplot.meanprops.markersize:         6
boxplot.meanprops.linestyle:          '--'
boxplot.meanprops.linewidth:          1.0
```

in your `matplotlibrc` file.

`fill_between` and `fill_betweenx`

`fill_between` and *`fill_betweenx`* both follow the patch color cycle.

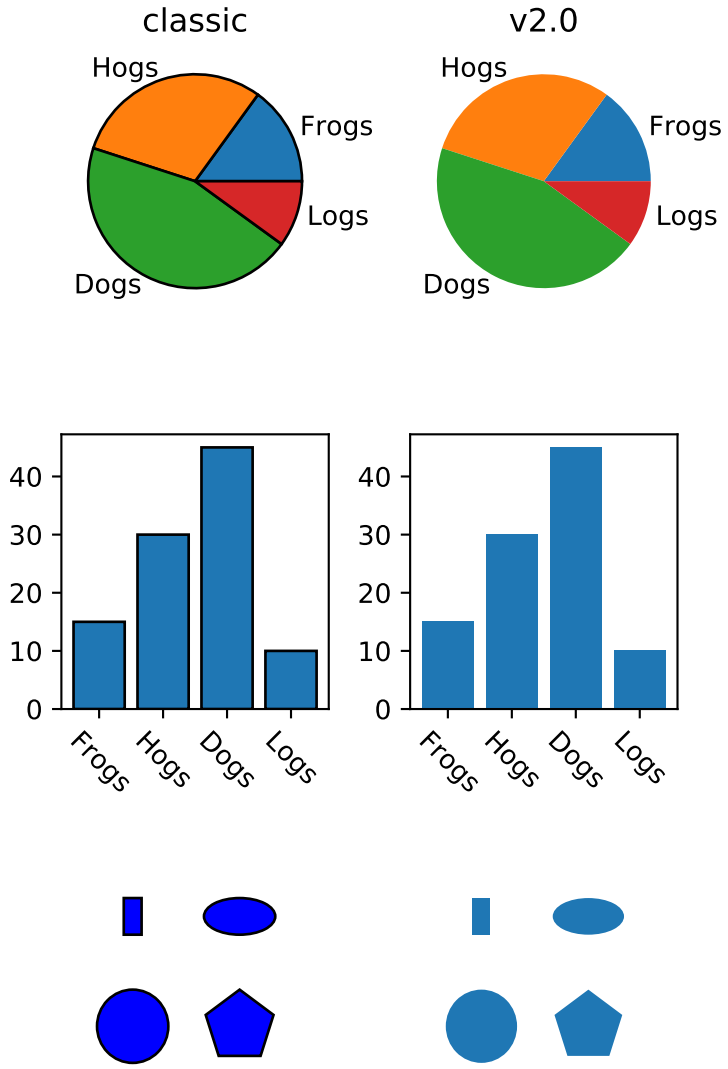


If the facecolor is set via the `facecolors` or `color` keyword argument, then the color is not cycled.

To restore the previous behavior, explicitly pass the keyword argument `facecolors='C0'` to the method call.

Patch edges and color

Most artists drawn with a patch (`~matplotlib.axes.Axes.bar`, `~matplotlib.axes.Axes.pie`, etc) no longer have a black edge by default. The default face color is now `'C0'` instead of `'b'`.



The previous defaults can be restored by setting:

```
mpl.rcParams['patch.force_edgecolor'] = True
mpl.rcParams['patch.facecolor'] = 'b'
```

or by setting:

```
patch.facecolor        : b
patch.force_edgecolor  : True
```

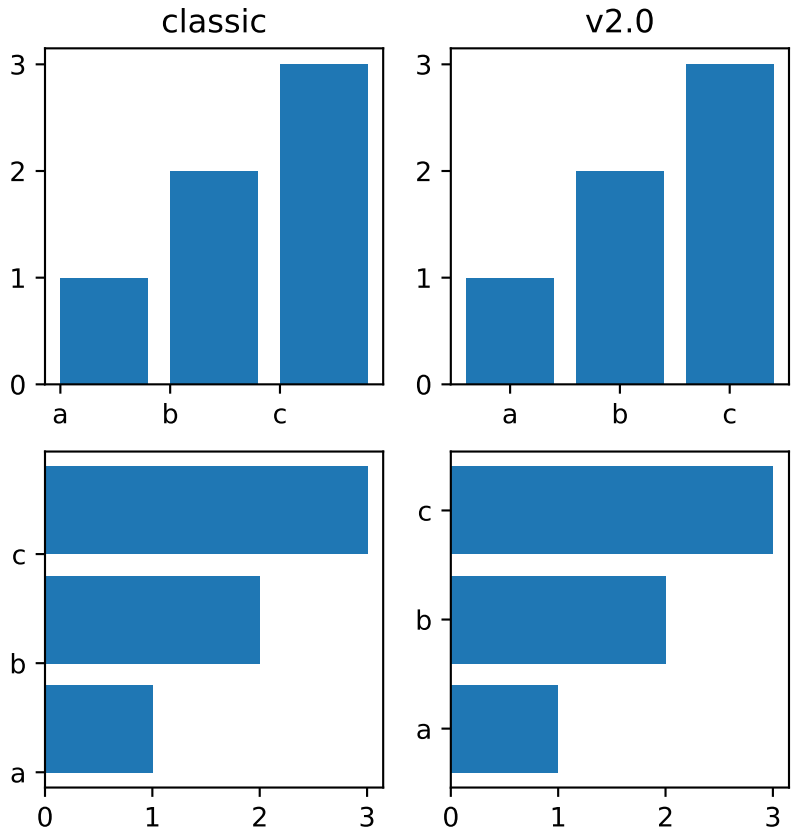
in your `matplotlibrc` file.

`hexbin`

The default value of the `linecolor` kwarg for `hexbin` has changed from `'none'` to `'face'`. If 'none' is now supplied, no line edges are drawn around the hexagons.

`bar` and `barh`

The default value of the `align` kwarg for both `bar` and `barh` is changed from `'edge'` to `'center'`.



To restore the previous behavior explicitly pass the keyword argument `align='edge'` to the method call.

Hatching

The color of the lines in the hatch is now determined by

- If an edge color is explicitly set, use that for the hatch color
- If the edge color is not explicitly set, use `rcParams["hatch.color"]` (default: `'black'`) which is looked up at artist creation time.

The width of the lines in a hatch pattern is now configurable by the rcParams `rcParams["hatch.linewidth"]` (default: `1.0`), which defaults to 1 point. The old behavior for the line width was different depending on backend:

- PDF: 0.1 pt
- SVG: 1.0 pt
- PS: 1 px
- Agg: 1 px

The old line width behavior can not be restored across all backends simultaneously, but can be restored for a single backend by setting:

```
mpl.rcParams['hatch.linewidth'] = 0.1  # previous pdf hatch linewidth
mpl.rcParams['hatch.linewidth'] = 1.0  # previous svg hatch linewidth
```

The behavior of the PS and Agg backends was DPI dependent, thus:

```
mpl.rcParams['figure.dpi'] = dpi
mpl.rcParams['savefig.dpi'] = dpi  # or leave as default 'figure'
mpl.rcParams['hatch.linewidth'] = 1.0 / dpi  # previous ps and Agg hatch linewidth
```

There is no direct API level control of the hatch color or linewidth.

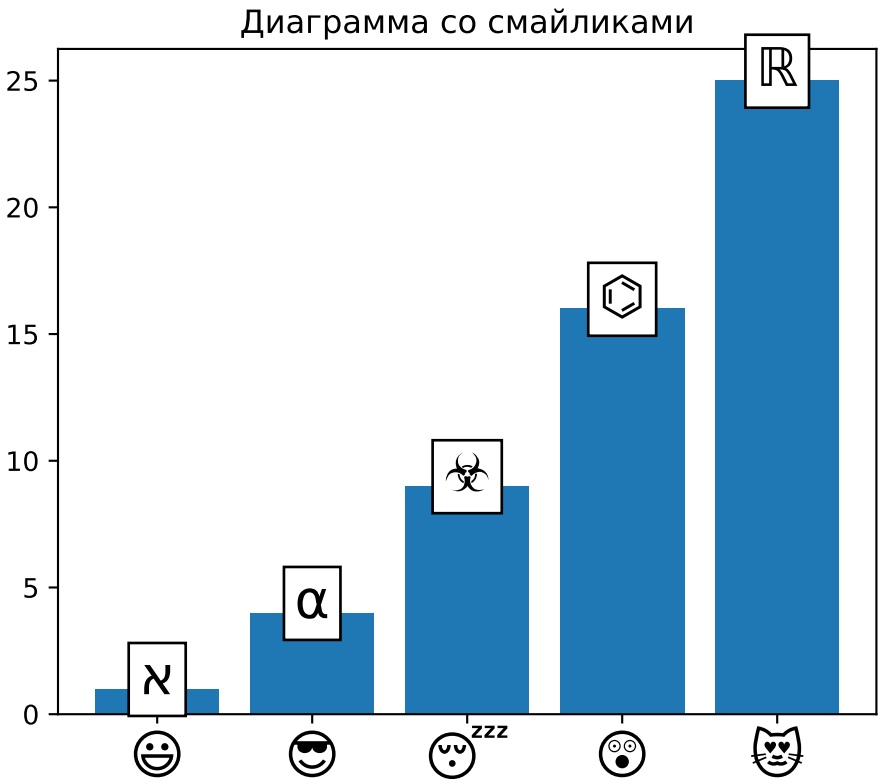
Hatching patterns are now rendered at a consistent density, regardless of DPI. Formerly, high DPI figures would be more dense than the default, and low DPI figures would be less dense. This old behavior cannot be directly restored, but the density may be increased by repeating the hatch specifier.

Fonts

Normal text

The default font has changed from "Bitstream Vera Sans" to "DejaVu Sans". DejaVu Sans has additional international and math characters, but otherwise has the same appearance as Bitstream Vera Sans. Latin, Greek, Cyrillic, Armenian, Georgian, Hebrew, and Arabic are all supported (but right-to-left rendering is still not handled by matplotlib). In addition, DejaVu contains a sub-set of emoji symbols.

See the DejaVu Sans PDF sample for full coverage.



Math text

The default math font when using the built-in math rendering engine (mathtext) has changed from "Computer Modern" (i.e. LaTeX-like) to "DejaVu Sans". This change has no effect if the TeX backend is used (i.e. `text.usetex` is `True`).

To revert to the old behavior set the:

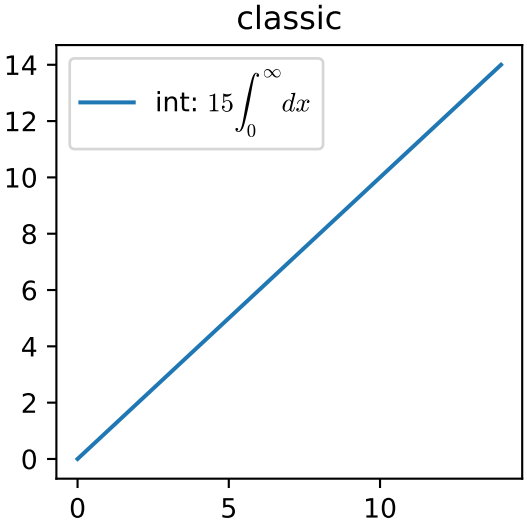
```
mpl.rcParams['mathtext.fontset'] = 'cm'
mpl.rcParams['mathtext.rm'] = 'serif'
```

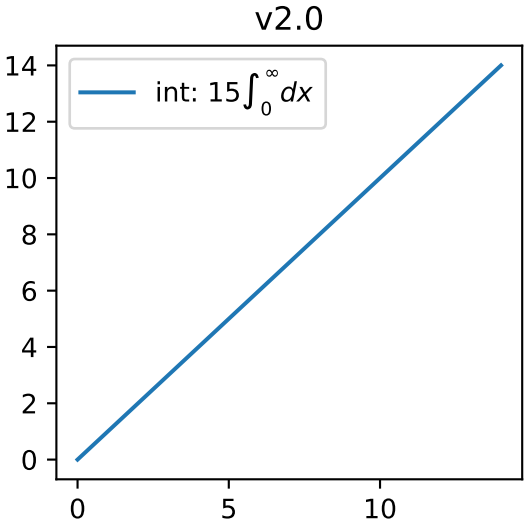
or set:

```
mathtext.fontset: cm
mathtext.rm : serif
```

in your `matplotlibrc` file.

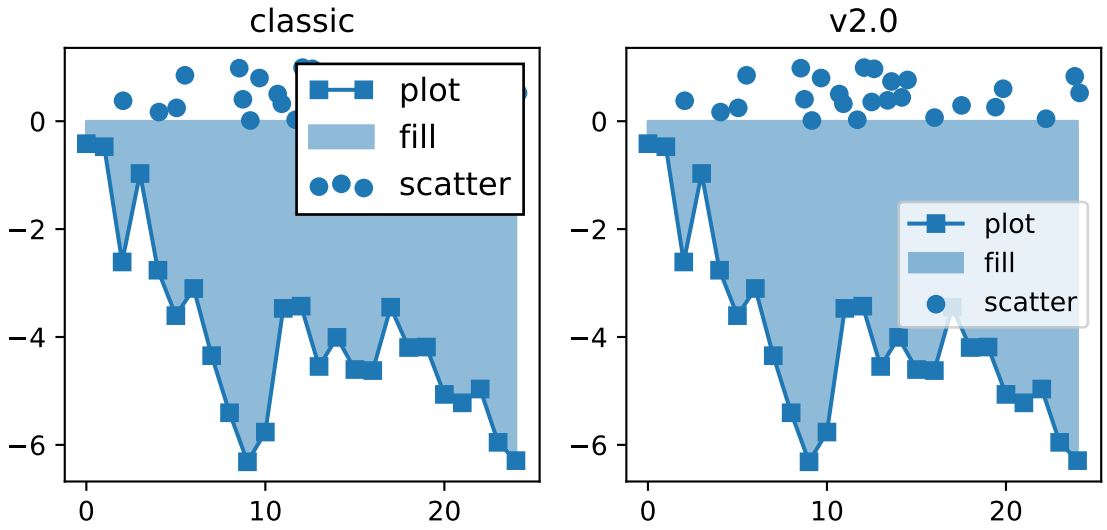
This `rcParam` is consulted when the text is drawn, not when the artist is created. Thus all mathtext on a given `canvas` will use the same fontset.


classic
int: $15\int_0^\infty dx$
0
2
4
6
8
10
12
14
0
5
10


v2.0
int: $15\int_0^\infty dx$
0
2
4
6
8
10
12
14
0
5
10

Legends

- By default, the number of points displayed in a legend is now 1.
- The default legend location is `'best'`, so the legend will be automatically placed in a location to minimize overlap with data.
- The legend defaults now include rounded corners, a lighter boundary, and partially transparent boundary and background.



The previous defaults can be restored by setting:

```
mpl.rcParams['legend.fancybox'] = False
mpl.rcParams['legend.loc'] = 'upper right'
mpl.rcParams['legend.numpoints'] = 2
mpl.rcParams['legend.fontsize'] = 'large'
mpl.rcParams['legend.framealpha'] = None
mpl.rcParams['legend.scatterpoints'] = 3
mpl.rcParams['legend.edgecolor'] = 'inherit'
```

or by setting:

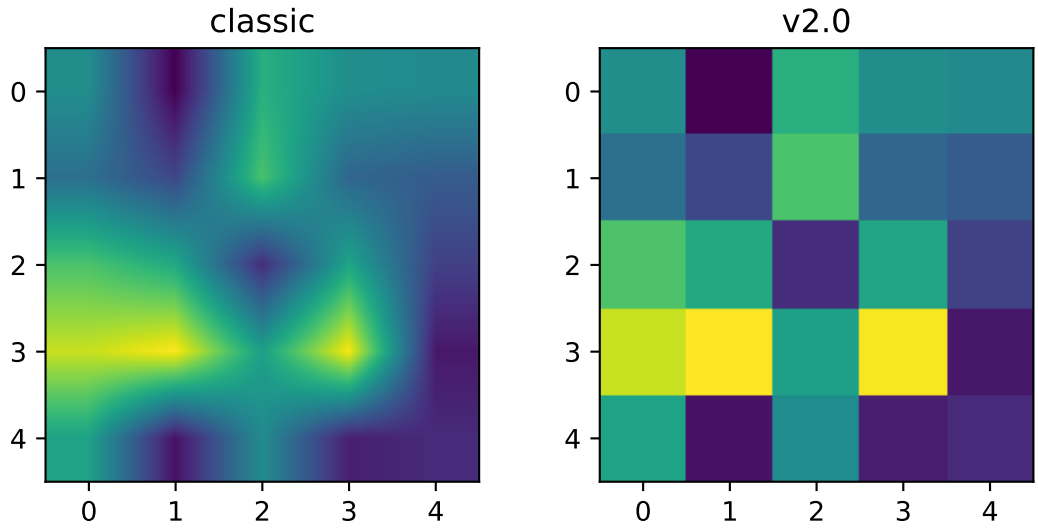
```
legend.fancybox      : False
legend.loc           : upper right
legend.numpoints     : 2      # the number of points in the legend line
legend.fontsize      : large
legend.framealpha    : None    # opacity of legend frame
legend.scatterpoints : 3 # number of scatter points
legend.edgecolor     : inherit   # legend edge color ('inherit'
                                 # means it uses axes.edgecolor)
```

in your `matplotlibrc` file.

Image

Interpolation

The default interpolation method for *imshow* is now `'nearest'` and by default it resamples the data (both up and down sampling) before color mapping.



To restore the previous behavior set:

```
mpl.rcParams['image.interpolation'] = 'bilinear'
mpl.rcParams['image.resample'] = False
```

or set:

```
image.interpolation  : bilinear  # see help(imshow) for options
image.resample  : False
```

in your `matplotlibrc` file.

Colormapping pipeline

Previously, the input data was normalized, then color mapped, and then resampled to the resolution required for the screen. This meant that the final resampling was being done in color space. Because the color maps are not generally linear in RGB space, colors not in the color map may appear in the final image. This bug was addressed by an almost complete overhaul of the image handling code.

The input data is now normalized, then resampled to the correct resolution (in normalized dataspace), and then color mapped to RGB space. This ensures that only colors from the color map appear in the final image. (If your viewer subsequently resamples the image, the artifact may reappear.)

The previous behavior cannot be restored.

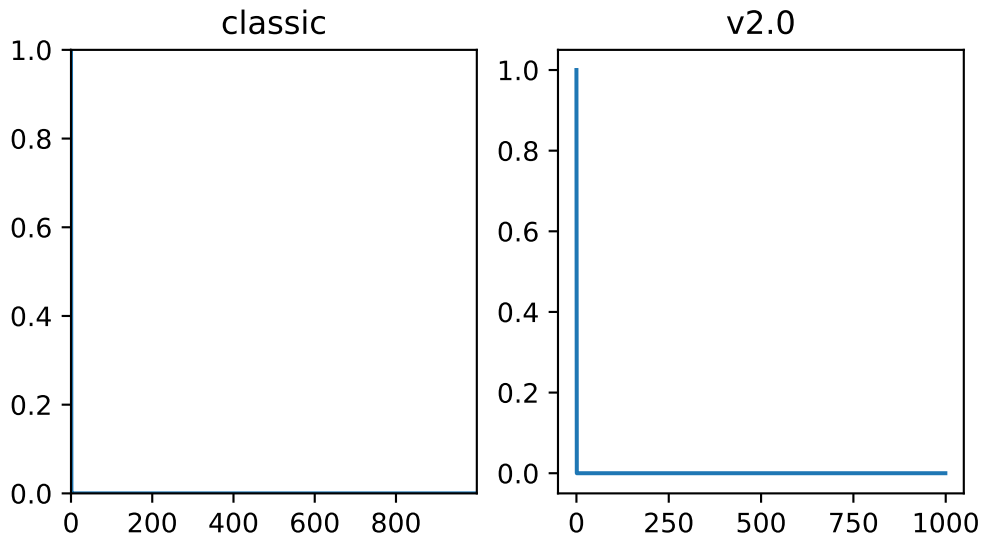
Shading

- The default shading mode for light source shading, in `matplotlib.colors.LightSource.shade`, is now `overlay`. Formerly, it was `hsv`.

Plot layout

Auto limits

The previous auto-scaling behavior was to find 'nice' round numbers as view limits that enclosed the data limits, but this could produce bad plots if the data happened to fall on a vertical or horizontal line near the chosen 'round number' limit. The new default sets the view limits to 5% wider than the data range.



The size of the padding in the x and y directions is controlled by the `'axes.xmargin'` and `'axes.ymargin'` rcParams respectively. Whether the view limits should be 'round numbers' is controlled by `rcParams["axes.autolimit_mode"]` (default: `'data'`). In the original `'round_number'` mode, the view limits coincide with ticks.

The previous default can be restored by using:

```
mpl.rcParams['axes.autolimit_mode'] = 'round_numbers'
mpl.rcParams['axes.xmargin'] = 0
mpl.rcParams['axes.ymargin'] = 0
```

or setting:

```
axes.autolimit_mode: round_numbers
axes.xmargin: 0
axes.ymargin: 0
```

in your `matplotlibrc` file.

Z-order

- Ticks and grids are now plotted above solid elements such as filled contours, but below lines. To return to the previous behavior of plotting ticks and grids above lines, set `rcParams['axes.axisbelow'] = False`.

Ticks

Direction

To reduce the collision of tick marks with data, the default ticks now point outward by default. In addition, ticks are now drawn only on the bottom and left spines to prevent a porcupine appearance, and for a cleaner separation between subplots.

To restore the previous behavior set:

```
mpl.rcParams['xtick.direction'] = 'in'
mpl.rcParams['ytick.direction'] = 'in'
mpl.rcParams['xtick.top'] = True
mpl.rcParams['ytick.right'] = True
```

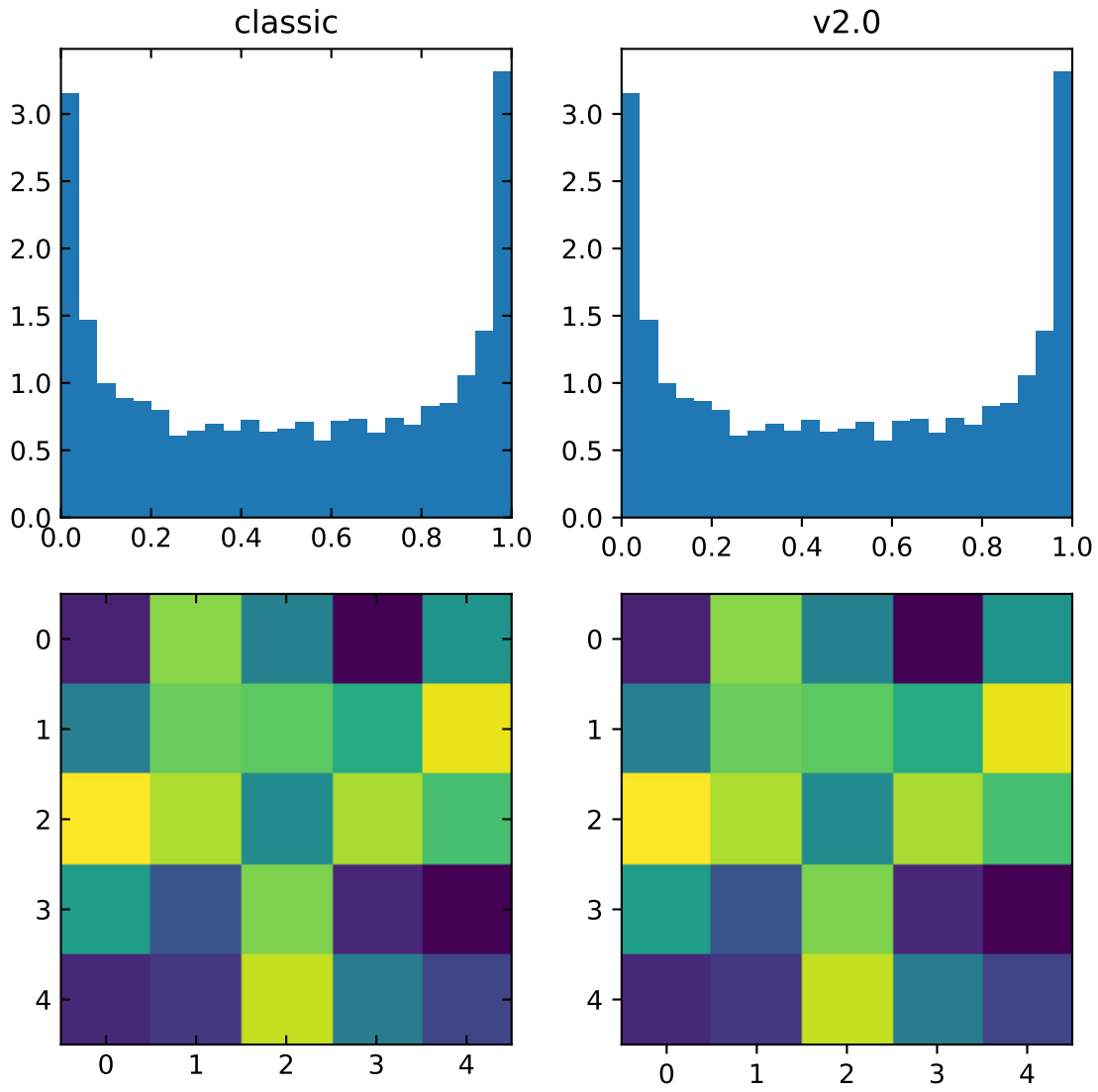
or set:

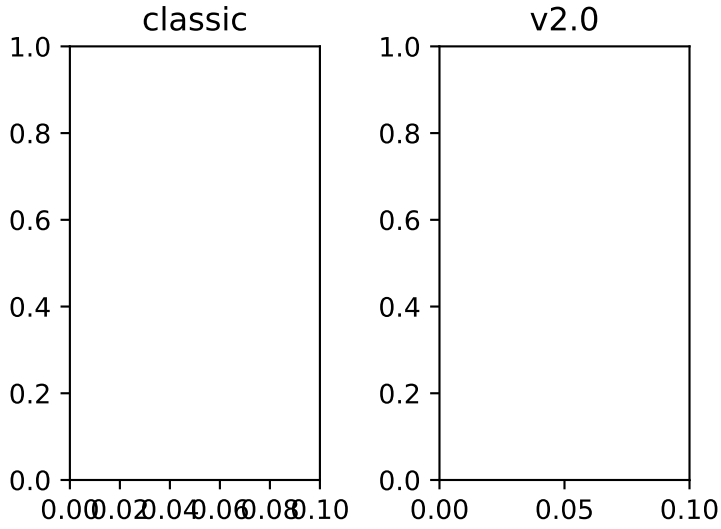
```
xtick.top: True
xtick.direction: in

ytick.right: True
ytick.direction: in
```

in your `matplotlibrc` file.

Number of ticks

The default *`Locator`* used for the x and y axis is *`AutoLocator`* which tries to find, up to some maximum number, 'nicely' spaced ticks. The locator now includes an algorithm to estimate the maximum number of ticks that will leave room for the tick labels. By default it also ensures that there are at least two ticks visible.


classic
v2.0
0.0
0.5
1.0
1.5
2.0
2.5
3.0
0.0
0.2
0.4
0.6
0.8
1.0
0
1
2
3
4



There is no way, other than using `mpl.style.use('classic')`, to restore the previous behavior as the default. On an axis-by-axis basis you may either control the existing locator via:

```
ax.xaxis.get_major_locator().set_params(nbins=9, steps=[1, 2, 5, 10])
```

or create a new *MaxNLocator*:

```
import matplotlib.ticker as mticker
ax.set_major_locator(mticker.MaxNLocator(nbins=9, steps=[1, 2, 5, 10])
```

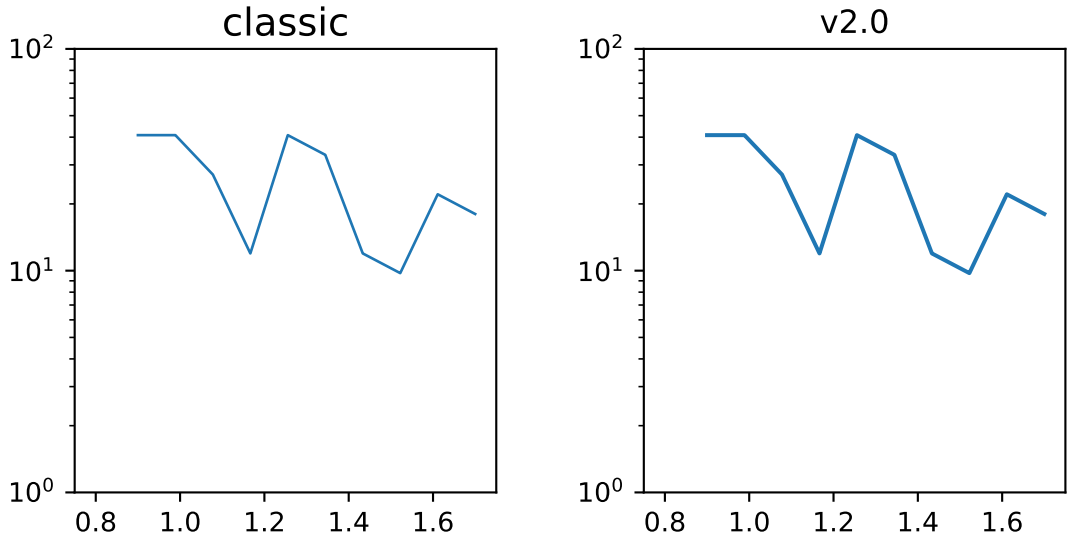
The algorithm used by *MaxNLocator* has been improved, and this may change the choice of tick locations in some cases. This also affects *AutoLocator*, which uses `MaxNLocator` internally.

For a log-scaled axis the default locator is the *LogLocator*. Previously the maximum number of ticks was set to 15, and could not be changed. Now there is a *numticks* kwarg for setting the maximum to any integer value, to the string 'auto', or to its default value of None which is equivalent to 'auto'. With the 'auto' setting the maximum number will be no larger than 9, and will be reduced depending on the length of the axis in units of the tick font size. As in the case of the AutoLocator, the heuristic algorithm reduces the incidence of overlapping tick labels but does not prevent it.

Tick label formatting

`LogFormatter` labeling of minor ticks

Minor ticks on a log axis are now labeled when the axis view limits span a range less than or equal to the interval between two major ticks. See *`LogFormatter`* for details. The minor tick labeling is turned off when using `mpl.style.use('classic')`, but cannot be controlled independently via *`rcParams`*.

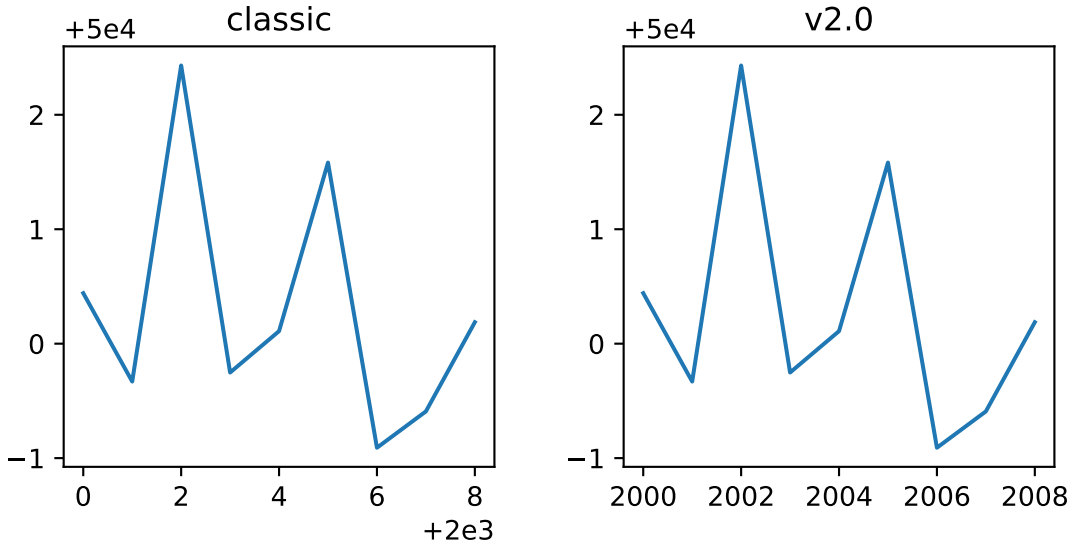


`ScalarFormatter` tick label formatting with offsets

With the default `rcParams["axes.formatter.useoffset"]` (default: `True`), an offset will be used when it will save 4 or more digits. This can be controlled with the new `rcParams["axes.formatter.offset_threshold"]` (default: `4`). To restore the previous behavior of using an offset to save 2 or more digits, use `rcParams['axes.formatter.offset_threshold'] = 2`.

`AutoDateFormatter` format strings

The default date formats are now all based on ISO format, i.e., with the slowest-moving value first. The date formatters are configurable through the `date.autoformatter.*` rcParams.



Threshold (tick interval >= than)	rcParam	classic	v2.0
365 days	'date.autoformatter.year'	'%Y'	'%Y'
30 days	'date.autoformatter.month'	'%b %Y'	'%Y-%m'
1 day	'date.autoformatter.day'	'%b %d %Y'	'%Y-%m-%d'
1 hour	'date.autoformatter.hour'	'%H:%M:%S'	'%H:%M'
1 minute	'date.autoformatter.minute'	'%H:%M:%S.%f'	'%H:%M:%S'
1 second	'date.autoformatter.second'	'%H:%M:%S.%f'	'%H:%M:%S'
1 microsecond	'date.autoformatter.microsecond'	'%H:%M:%S.%f'	'%H:%M:%S.%f'

Python's `%x` and `%X` date formats may be of particular interest to format dates based on the current locale.

The previous default can be restored by:

```
mpl.rcParams['date.autoformatter.year'] = '%Y'
mpl.rcParams['date.autoformatter.month'] = '%b %Y'
mpl.rcParams['date.autoformatter.day'] = '%b %d %Y'
mpl.rcParams['date.autoformatter.hour'] = '%H:%M:%S'
mpl.rcParams['date.autoformatter.minute'] = '%H:%M:%S.%f'
mpl.rcParams['date.autoformatter.second'] = '%H:%M:%S.%f'
mpl.rcParams['date.autoformatter.microsecond'] = '%H:%M:%S.%f'
```

or setting

```
date.autoformatter.year   : %Y
date.autoformatter.month  : %b %Y
```

(continues on next page)

(continued from previous page)

```
date.autoformatter.day    : %b %d %Y
date.autoformatter.hour   : %H:%M:%S
date.autoformatter.minute : %H:%M:%S.%f
date.autoformatter.second : %H:%M:%S.%f
date.autoformatter.microsecond : %H:%M:%S.%f
```

in your `matplotlibrc` file.

mplot3d

- mplot3d now obeys some style-related rcParams, rather than using hard-coded defaults. These include:
  - xtick.major.width
  - ytick.major.width
  - xtick.color
  - ytick.color
  - axes.linewidth
  - axes.edgecolor
  - grid.color
  - grid.linewidth
  - grid.linestyle

8.6.2 Improved color conversion API and RGBA support

The *colors* gained a new color conversion API with full support for the alpha channel. The main public functions are *is_color_like()*, *matplotlib.colors.to_rgba()*, *matplotlib.colors.to_rgba_array()* and *to_hex()*. RGBA quadruplets are encoded in hex format as "#rrggbbaa".

A side benefit is that the Qt options editor now allows setting the alpha channel of the artists as well.

8.6.3 New Configuration (rcParams)

New rcparams added

Parameter	Description
rcParams["date.autoformatter.year"] (default: '%Y')	format string for 'year' scale dates
rcParams["date.autoformatter.month"] (default: '%Y-%m')	format string for 'month' scale dates
rcParams["date.autoformatter.day"] (default: '%Y-%m-%d')	format string for 'day' scale dates
rcParams["date.autoformatter.hour"] (default: '%m-%d %H')	format string for 'hour' scale times
rcParams["date.autoformatter.minute"] (default: '%d %H:%M')	format string for 'minute' scale times
rcParams["date.autoformatter.second"] (default: '%H:%M:%S')	format string for 'second' scale times
rcParams["date.autoformatter.microsecond"] (default: '%M:%S.%f')	format string for 'microsecond' scale times
rcParams["scatter.marker"] (default: 'o')	default marker for scatter plot
rcParams["svg.hashsalt"] (default: None)	see note
rcParams["xtick.top"] (default: False), rcParams["xtick.major.top"] (default: True) rcParams["xtick.minor.top"] (default: True), rcParams["xtick.bottom"] (default: True), rcParams["xtick.major.bottom"] (default: True) rcParams["xtick.minor.bottom"] (default: True), rcParams["ytick.left"] (default: True), rcParams["ytick.minor.left"] (default: True), rcParams["ytick.major.left"] (default: True) rcParams["ytick.right"] (default: False), rcParams["ytick.minor.right"] (default: True), rcParams["ytick.major.right"] (default: True)	Control where major and minor ticks are drawn. The global values are anded with the corresponding major/minor values. corresponding major/minor values.
rcParams["hist.bins"] (default: 10)	The default number of bins to use in *hist*. This can be an int, a list of floats, or 'auto' if numpy >= 1.11 is installed.
rcParams["lines.scale_dashes"] (default: True)	Whether the line dash patterns should scale with linewidth.
rcParams["axes.formatter.offset_threshold"] (default: 4)	Minimum number of digits saved in tick labels that triggers using an offset.

Added `svg.hashsalt` key to rcParams

If `svg.hashsalt` is `None` (which it is by default), the svg backend uses `uuid4` to generate the hash salt. If it is not `None`, it must be a string that is used as the hash salt instead of `uuid4`. This allows for deterministic SVG output.

Removed the `svg.image_noscale` rcParam

As a result of the extensive changes to image handling, the `svg.image_noscale` rcParam has been removed. The same functionality may be achieved by setting `interpolation='none'` on individual images or globally using the `image.interpolation` rcParam.

8.6.4 Qualitative colormaps

ColorBrewer's "qualitative" colormaps ("Accent", "Dark2", "Paired", "Pastel1", "Pastel2", "Set1", "Set2", "Set3") were intended for discrete categorical data, with no implication of value, and therefore have been converted to `ListedColormap` instead of `LinearSegmentedColormap`, so the colors will no longer be interpolated and they can be used for choropleths, labeled image features, etc.

8.6.5 Axis offset label now responds to `labelcolor`

Axis offset labels are now colored the same as axis tick markers when `labelcolor` is altered.

8.6.6 Improved offset text choice

The default offset-text choice was changed to only use significant digits that are common to all ticks (e.g. 1231..1239 -> 1230, instead of 1231), except when they straddle a relatively large multiple of a power of ten, in which case that multiple is chosen (e.g. 1999..2001->2000).

8.6.7 Style parameter blacklist

In order to prevent unexpected consequences from using a style, style files are no longer able to set parameters that affect things unrelated to style. These parameters include:

```
'interactive', 'backend', 'backend.qt4', 'webagg.port',
'webagg.port_retries', 'webagg.open_in_browser', 'backend_fallback',
'toolbar', 'timezone', 'datapath', 'figure.max_open_warning',
'savefig.directory', 'tk.window_focus', 'docstring.hardcopy'
```

8.6.8 Change in default font

The default font used by matplotlib in text has been changed to DejaVu Sans and DejaVu Serif for the sans-serif and serif families, respectively. The DejaVu font family is based on the previous matplotlib default --Bitstream Vera-- but includes a much wider range of characters.

The default mathtext font has been changed from Computer Modern to the DejaVu family to maintain consistency with regular text. Two new options for the `mathtext.fontset` configuration parameter have been added: `dejavusans` (default) and `dejavuserif`. Both of these options use DejaVu glyphs whenever possible and fall back to STIX symbols when a glyph is not found in DejaVu. To return to the previous behavior, set the rcParam `mathtext.fontset` to `cm`.

8.6.9 Faster text rendering

Rendering text in the Agg backend is now less fuzzy and about 20% faster to draw.

8.6.10 Improvements for the Qt figure options editor

Various usability improvements were implemented for the Qt figure options editor, among which:

- Line style entries are now sorted without duplicates.
- The colormap and normalization limits can now be set for images.
- Line edits for floating values now display only as many digits as necessary to avoid precision loss. An important bug was also fixed regarding input validation using Qt5 and a locale where the decimal separator is ",".
- The axes selector now uses shorter, more user-friendly names for axes, and does not crash if there are no axes.
- Line and image entries using the default labels ("_lineX", "_imageX") are now sorted numerically even when there are more than 10 entries.

8.6.11 Improved image support

Prior to version 2.0, matplotlib resampled images by first applying the color map and then resizing the result. Since the resampling was performed on the colored image, this introduced colors in the output image that didn't actually exist in the color map. Now, images are resampled first (and entirely in floating-point, if the input image is floating-point), and then the color map is applied.

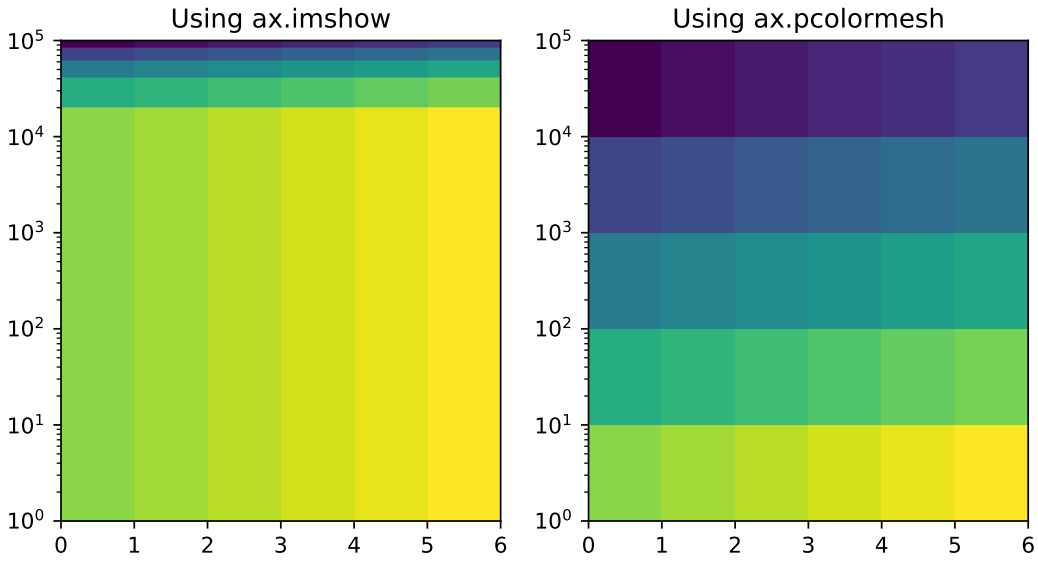
In order to make this important change, the image handling code was almost entirely rewritten. As a side effect, image resampling uses less memory and fewer datatype conversions than before.

The experimental private feature where one could "skew" an image by setting the private member `_image_skew_coordinate` has been removed. Instead, images will obey the transform of the axes on which they are drawn.

Non-linear scales on image plots

imshow now draws data at the requested points in data space after the application of non-linear scales.

The image on the left demonstrates the new, correct behavior. The old behavior can be recreated using *pcolormesh* as demonstrated on the right.



This can be understood by analogy to plotting a histogram with linearly spaced bins with a logarithmic x-axis. Equal sized bins will be displayed as wider for small x and narrower for large x.

8.6.12 Support for HiDPI (Retina) displays in the NbAgg and WebAgg backends

The NbAgg and WebAgg backends will now use the full resolution of your high-pixel-density display.

8.6.13 Change in the default animation codec

The default animation codec has been changed from `mpeg4` to `h264`, which is more efficient. It can be set via the `animation.codec` rcParam.

8.6.14 Deprecated support for mencoder in animation

The use of mencoder for writing video files with mpl is problematic; switching to ffmpeg is strongly advised. All support for mencoder will be removed in version 2.2.

8.6.15 Boxplot Zorder Keyword Argument

The *zorder* parameter now exists for *boxplot*. This allows the zorder of a boxplot to be set in the plotting function call.

```
boxplot(np.arange(10), zorder=10)
```

8.6.16 Filled + and x markers

New fillable *plus* and *x* markers have been added. See the *markers* module and marker reference examples.

8.6.17 *rcount* and *ccount* for `plot_surface`

As of v2.0, mplot3d's *plot_surface* now accepts *rcount* and *ccount* arguments for controlling the sampling of the input data for plotting. These arguments specify the maximum number of evenly spaced samples to take from the input data. These arguments are also the new default sampling method for the function, and is considered a style change.

The old *rstride* and *cstride* arguments, which specified the size of the evenly spaced samples, become the default when 'classic' mode is invoked, and are still available for use. There are no plans for deprecating these arguments.

8.6.18 Streamplot Zorder Keyword Argument Changes

The `zorder` parameter for *streamplot* now has default value of `None` instead of `2`. If `None` is given as `zorder`, *streamplot* has a default `zorder` of `matplotlib.lines.Line2D.zorder`.

8.6.19 Extension to `matplotlib.backend_bases.GraphicsContextBase`

To support standardizing hatch behavior across the backends we ship the *matplotlib.backend_bases.GraphicsContextBase.get_hatch_color* method as added to *matplotlib.backend_bases.GraphicsContextBase*. This is only used during the render process in the backends we ship so will not break any third-party backends.

If you maintain a third-party backend which extends *GraphicsContextBase* this method is now available to you and should be used to color hatch patterns.

8.7 New in matplotlib 1.5

Table of Contents

Note: matplotlib 1.5 supports Python 2.7, 3.4, and 3.5

8.7.1 Interactive OO usage

All *Artist*s now keep track of if their internal state has been changed but not reflected in the display ('stale') by a call to `draw`. It is thus possible to pragmatically determine if a given *Figure* needs to be re-drawn in an interactive session.

To facilitate interactive usage a `draw_all` method has been added to `pyplot` which will redraw all of the figures which are 'stale'.

To make this convenient for interactive use matplotlib now registers a function either with IPython's 'post_execute' event or with the displayhook in the standard python REPL to automatically call `plt.draw_all` just before control is returned to the REPL. This ensures that the draw command is deferred and only called once.

The upshot of this is that for interactive backends (including `%matplotlib notebook`) in interactive mode (with `plt.ion()`)

```
import matplotlib.pyplot as plt
fig, ax = plt.subplots()
ln, = ax.plot([0, 1, 4, 9, 16])
plt.show()
ln.set_color('g')
```

will automatically update the plot to be green. Any subsequent modifications to the `Artist` objects will do likewise.

This is the first step of a larger consolidation and simplification of the pyplot internals.

8.7.2 Working with labeled data like pandas DataFrames

Plot methods which take arrays as inputs can now also work with labeled data and unpack such data.

This means that the following two examples produce the same plot:

Example

```
df = pandas.DataFrame({"var1":[1,2,3,4,5,6], "var2":[1,2,3,4,5,6]})
plt.plot(df["var1"], df["var2"])
```

Example

```
plt.plot("var1", "var2", data=df)
```

This works for most plotting methods, which expect arrays/sequences as inputs. `data` can be anything which supports `__getitem__` (`dict`, `pandas.DataFrame`, `h5py`, ...) to access `array` like values with string keys.

In addition to this, some other changes were made, which makes working with labeled data (ex `pandas.Series`) easier:

- For plotting methods with `label` keyword argument, one of the data inputs is designated as the label source. If the user does not supply a `label` that value object will be introspected for a label, currently by looking for a `name` attribute. If the value object does not have a `name` attribute but was specified by as a key into the `data` kwarg, then the key is used. In the above examples, this results in an implicit `label="var2"` for both cases.
- `plot()` now uses the index of a `Series` instead of `np.arange(len(y))`, if no `x` argument is supplied.

8.7.3 Added `axes.prop_cycle` key to rcParams

This is a more generic form of the now-deprecated `axes.color_cycle` param. Now, we can cycle more than just colors, but also linestyles, hatches, and just about any other artist property. Cycler notation is used for defining property cycles. Adding cyclers together will be like you are `zip`-ing together two or more property cycles together:

```
axes.prop_cycle: cycler('color', 'rgb') + cycler('lw', [1, 2, 3])
```

You can even multiply cyclers, which is like using `itertools.product` on two or more property cycles.

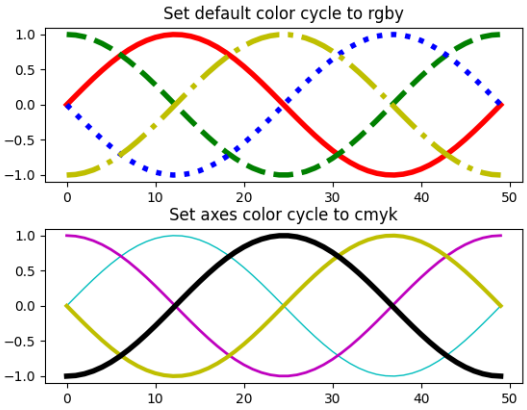
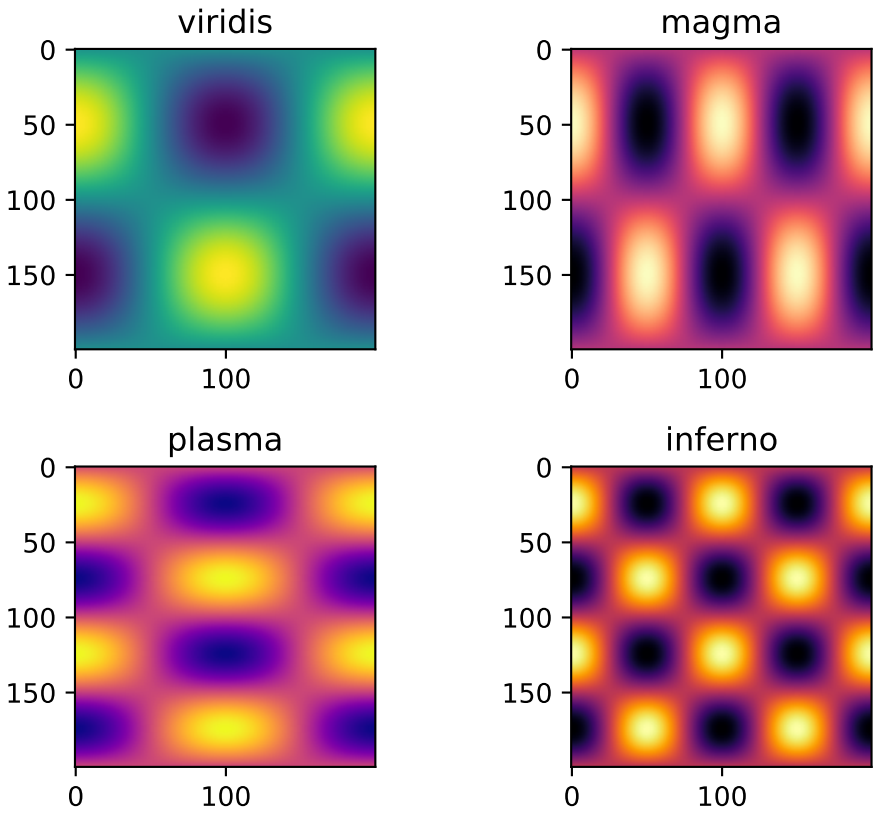


Fig. 5: Color Cycle

8.7.4 New Colormaps

All four of the colormaps proposed as the new default are available as `'viridis'` (the new default in 2.0), `'magma'`, `'plasma'`, and `'inferno'`



8.7.5 Styles

Several new styles have been added, including many styles from the Seaborn project. Additionally, in order to prep for the upcoming 2.0 style-change release, a 'classic' and 'default' style has been added. For this release, the 'default' and 'classic' styles are identical. By using them now in your scripts, you can help ensure a smooth transition during future upgrades of matplotlib, so that you can upgrade to the snazzy new defaults when you are ready!

```
import matplotlib.style
matplotlib.style.use('classic')
```

The 'default' style will give you matplotlib's latest plotting styles:

```
matplotlib.style.use('default')
```

8.7.6 Backends

New backend selection

The environment variable *MPLBACKEND* can now be used to set the matplotlib backend.

wx backend has been updated

The wx backend can now be used with both wxPython classic and Phoenix.

wxPython classic has to be at least version 2.8.12 and works on Python 2.x. As of May 2015 no official release of wxPython Phoenix is available but a current snapshot will work on Python 2.7+ and 3.4+.

If you have multiple versions of wxPython installed, then the user code is responsible setting the wxPython version. How to do this is explained in the comment at the beginning of the example /gallery/user_interfaces/embedding_in_wx2_sgskip.

8.7.7 Configuration (rcParams)

Some parameters have been added, others have been improved.

Parameter	Description
rcParams["xaxis.labelpad"], rcParams["yaxis.labelpad"]	mplot3d now respects these parameters
rcParams["axes.labelpad"] (default: 4.0)	Default space between the axis and the label
rcParams["errorbar.capsize"] (default: 0.0)	Default length of end caps on error bars
rcParams["xtick.minor.visible"] (default: False), rcParams["ytick.minor.visible"] (default: False)	Default visibility of minor x/y ticks
rcParams["legend.framealpha"] (default: 0.8)	Default transparency of the legend frame box
rcParams["legend.facecolor"] (default: 'inherit')	Default facecolor of legend frame box (or 'inherit' from rcParams["axes.facecolor"] (default: 'white'))
rcParams["legend.edgecolor"] (default: '0.8')	Default edgecolor of legend frame box (or 'inherit' from rcParams["axes.edgecolor"] (default: 'black'))
rcParams["figure.titlesize"] (default: 'large')	Default font size for figure suptitles
rcParams["figure.titleweight"] (default: 'normal')	Default font weight for figure suptitles
rcParams["image.composite_image"] (default: True)	Whether a vector graphics backend should composite several images into a single image or not when saving. Useful when needing to edit the files further in Inkscape or other programs.
rcParams["markers.fillstyle"] (default: 'full')	Default fillstyle of markers. Possible values are 'full' (the default), 'left', 'right', 'bottom', 'top' and 'none'
rcParams["toolbar"] (default: 'toolbar2')	Added 'toolmanager' as a valid value, enabling the experimental ToolManager feature.

8.7.8 Widgets

Active state of Selectors

All selectors now implement set_active and get_active methods (also called when accessing the active property) to properly update and query whether they are active.

Moved `ignore`, `set_active`, and `get_active` methods to base class `Widget`

Pushes up duplicate methods in child class to parent class to avoid duplication of code.

Adds enable/disable feature to MultiCursor

A MultiCursor object can be disabled (and enabled) after it has been created without destroying the object. Example:

```
multi_cursor.active = False
```

Improved RectangleSelector and new EllipseSelector Widget

Adds an *interactive* keyword which enables visible handles for manipulating the shape after it has been drawn.

Adds keyboard modifiers for:

- Moving the existing shape (default key = 'space')
- Making the shape square (default 'shift')
- Make the initial point the center of the shape (default 'control')
- Square and center can be combined

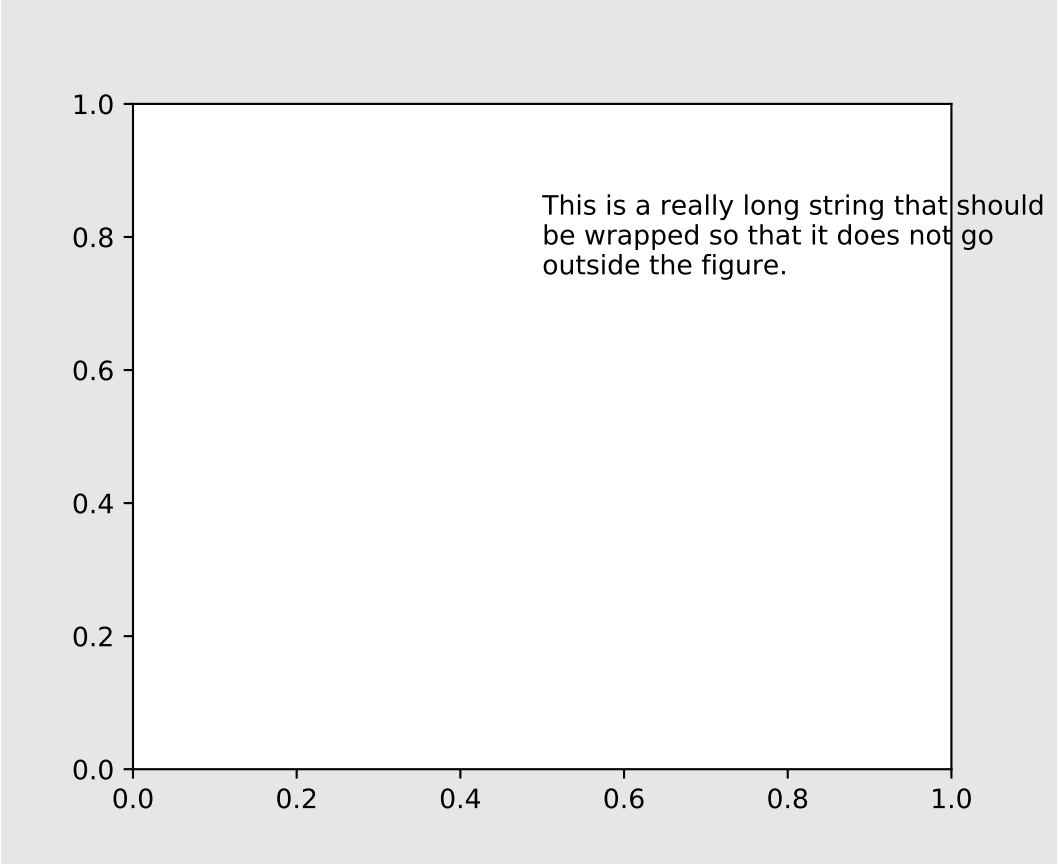
Allow Artists to Display Pixel Data in Cursor

Adds *get_cursor_data* and *format_cursor_data* methods to artists which can be used to add zdata to the cursor display in the status bar. Also adds an implementation for Images.

8.7.9 New plotting features

Auto-wrapping Text

Added the keyword argument "wrap" to Text, which automatically breaks long lines of text when being drawn. Works for any rotated text, different modes of alignment, and for text that are either labels or titles. This breaks at the `Figure`, not `Axes` edge.



Contour plot corner masking

Ian Thomas rewrote the C++ code that calculates contours to add support for corner masking. This is controlled by a new keyword argument `corner_mask` in the functions *`contour()`* and *`contourf()`*. The previous behaviour, which is now obtained using `corner_mask=False`, was for a single masked point to completely mask out all four quads touching that point. The new behaviour, obtained using `corner_mask=True`, only masks the corners of those quads touching the point; any triangular corners comprising three unmasked points are contoured as usual. If the `corner_mask` keyword argument is not specified, the default value is taken from rcParams.

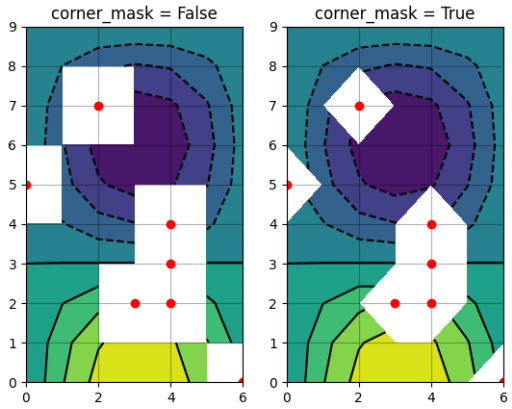


Fig. 6: Contour Corner Mask

Mostly unified linestyles for `Line2D`, `Patch` and `Collection`

The handling of linestyles for Lines, Patches and Collections has been unified. Now they all support defining linestyles with short symbols, like "--", as well as with full names, like "dashed". Also the definition using a dash pattern (`(0., [3., 3.])`) is supported for all methods using *`Line2D`*, *`Patch`* or *`Collection`*.

Legend marker order

Added ability to place the label before the marker in a legend box with `markerfirst` keyword

Support for legend for PolyCollection and stackplot

Added a *legend_handler* for *PolyCollection* as well as a *labels* argument to *stackplot()*.

Support for alternate pivots in mplot3d quiver plot

Added a `pivot` kwarg to *quiver()* that controls the pivot point around which the quiver line rotates. This also determines the placement of the arrow head along the quiver line.

Logit Scale

Added support for the 'logit' axis scale, a nonlinear transformation

$$x- > \log 10(x/(1-x))$$

for data between 0 and 1 excluded.

Add step kwargs to fill_between

Added `step` kwarg to *Axes.fill_between* to allow to fill between lines drawn using the 'step' draw style. The values of `step` match those of the `where` kwarg of *Axes.step*. The asymmetry of of the kwargs names is not ideal, but *Axes.fill_between* already has a `where` kwarg.

This is particularly useful for plotting pre-binned histograms.

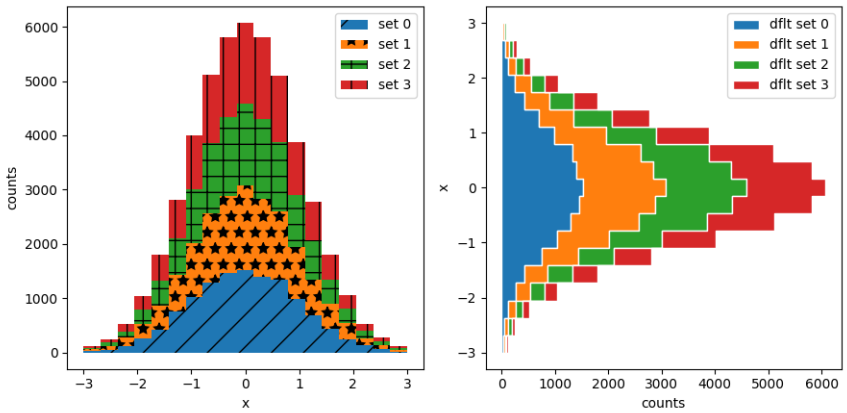
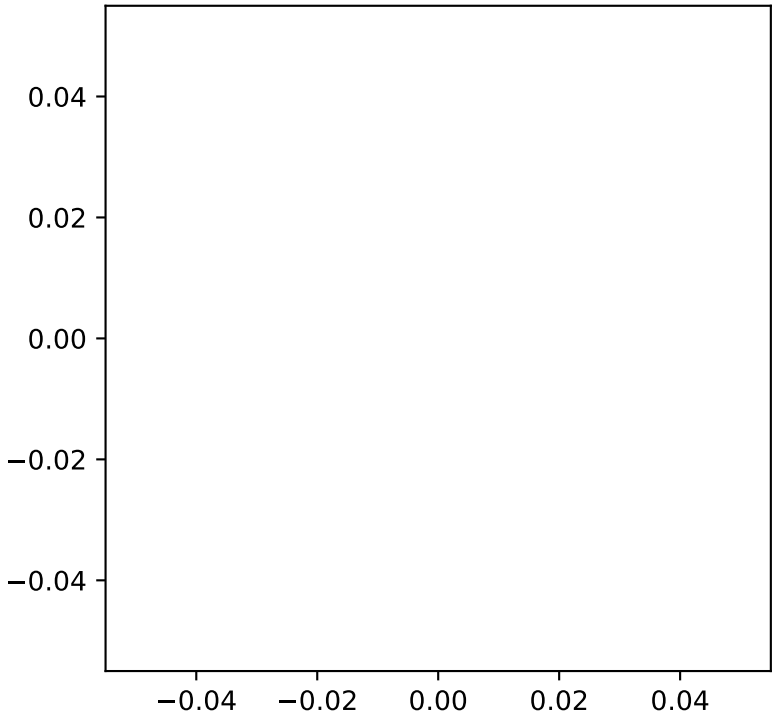


Fig. 7: Filled Step

Square Plot

Implemented square plots feature as a new parameter in the axis function. When argument 'square' is specified, equal scaling is set, and the limits are set such that `xmax-xmin == ymax-ymin`.



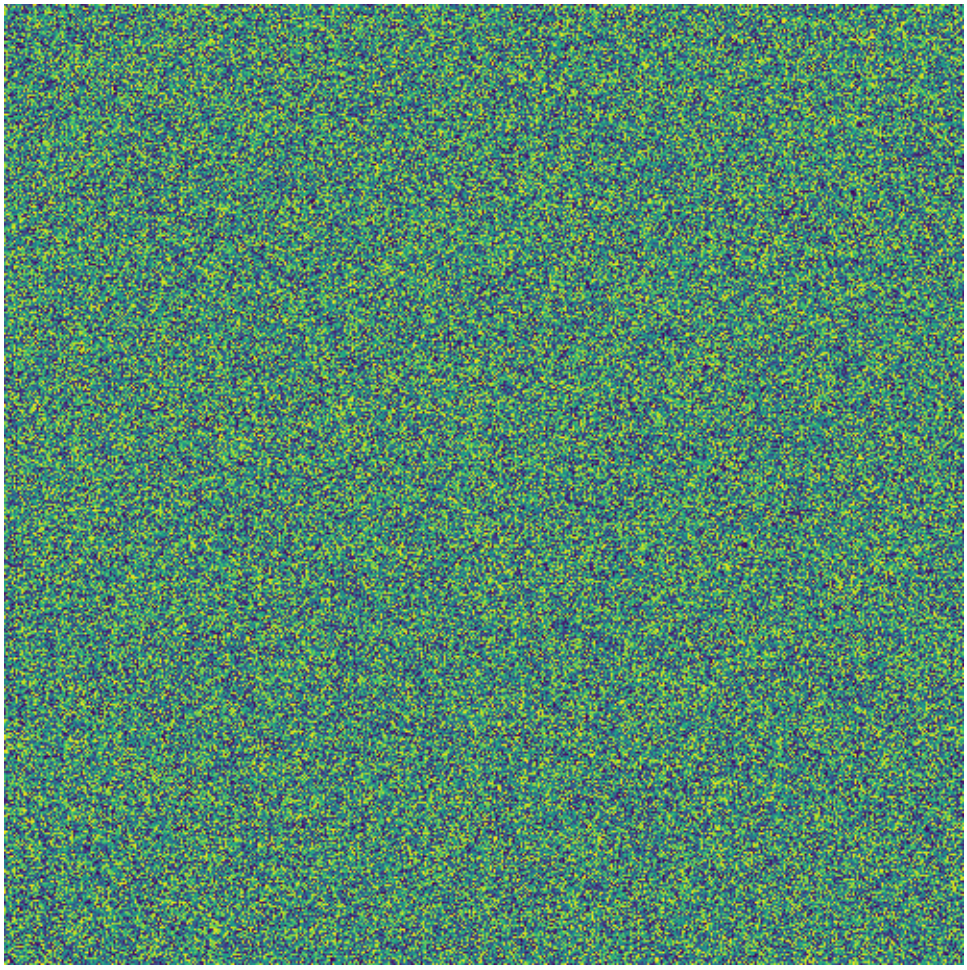
Updated figimage to take optional resize parameter

Added the ability to plot simple 2D-Array using `plt.figimage(X, resize=True)`. This is useful for plotting simple 2D-Array without the Axes or whitespacing around the image.

Updated Figure.savefig() can now use figure's dpi

Added support to save the figure with the same dpi as the figure on the screen using `dpi='figure'`:.

Example:



```
f = plt.figure(dpi=25)  # dpi set to 25
S = plt.scatter([1,2,3],[4,5,6])
f.savefig('output.png', dpi='figure')  # output savefig dpi set to 25 (same as figure)
```

Updated Table to control edge visibility

Added the ability to toggle the visibility of lines in Tables. Functionality added to the *pyplot.table* factory function under the keyword argument "edges". Values can be the strings "open", "closed", "horizontal", "vertical" or combinations of the letters "L", "R", "T", "B" which represent left, right, top, and bottom respectively.

Example:

```
table(..., edges="open")  # No line visible
table(..., edges="closed")  # All lines visible
table(..., edges="horizontal")  # Only top and bottom lines visible
table(..., edges="LT")  # Only left and top lines visible.
```

Zero r/cstride support in plot_wireframe

Adam Hughes added support to mplot3d's plot_wireframe to draw only row or column line plots.

Plot bar and barh with labels

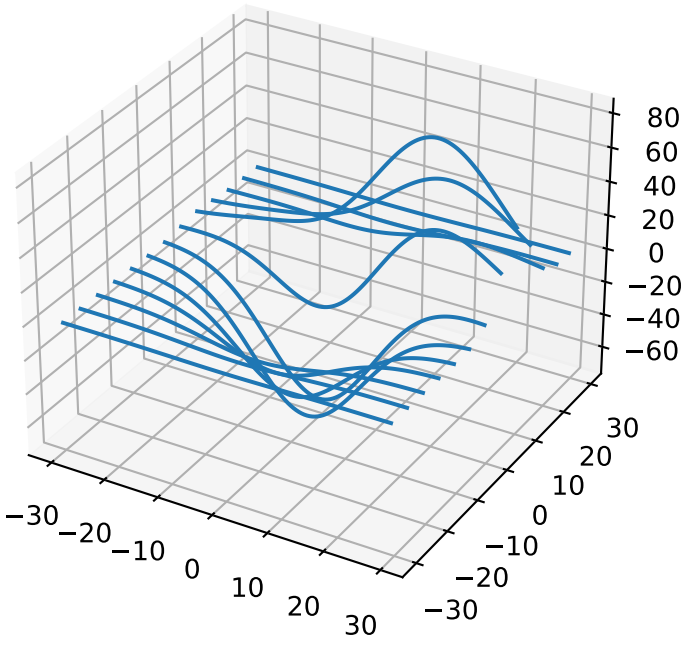
Added kwarg *tick_label* to *bar* and *barh* to support plotting bar graphs with a text label for each bar.

Added center and frame kwargs to pie

These control where the center of the pie graph are and if the Axes frame is shown.

Fixed 3D filled contour plot polygon rendering

Certain cases of 3D filled contour plots that produce polygons with multiple holes produced improper rendering due to a loss of path information between *PolyCollection* and *Poly3DCollection*. A function *set_verts_and_codes()* was added to allow path information to be retained for proper rendering.

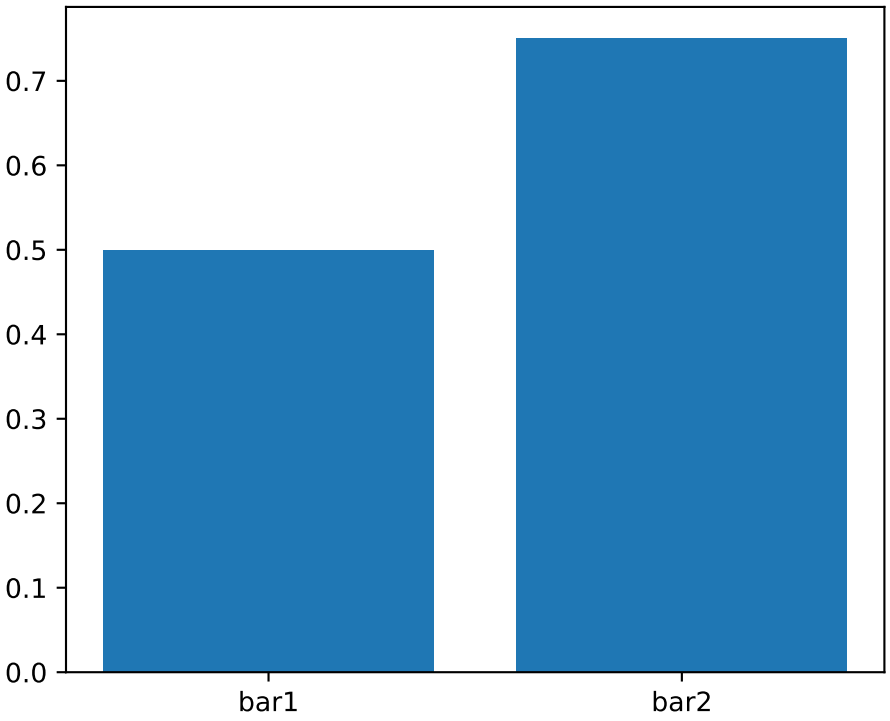


Dense colorbars are rasterized

Vector file formats (pdf, ps, svg) are efficient for many types of plot element, but for some they can yield excessive file size and even rendering artifacts, depending on the renderer used for screen display. This is a problem for colorbars that show a large number of shades, as is most commonly the case. Now, if a colorbar is showing 50 or more colors, it will be rasterized in vector backends.

DateFormatter strftime

DateFormatter's `__call__()` method will format a `datetime.datetime` object with the format string passed to the formatter's constructor. This method accepts datetimes with years before 1900, unlike `datetime.datetime.strftime()`.



Artist-level {get,set}_usetex for text

Add `{get,set}_usetex` methods to *Text* objects which allow artist-level control of LaTeX rendering vs. the internal mathtex rendering.

`Axes.remove()` works as expected

As with artists added to an *Axes*, *Axes* objects can be removed from their figure via *remove()*.

API Consistency fix within Locators set_params() function

set_params() function, which sets parameters within a *Locator* type instance, is now available to all *Locator* types. The implementation also prevents unsafe usage by strictly defining the parameters that a user can set.

To use, call `set_params()` on a *Locator* instance with desired arguments:

```
loc = matplotlib.ticker.LogLocator()
# Set given attributes for loc.
```

(continues on next page)

(continued from previous page)

```
loc.set_params(numticks=8, numdecs=8, subs=[2.0], base=8)
# The below will error, as there is no such parameter for LogLocator
# named foo
# loc.set_params(foo='bar')
```

Date Locators

Date Locators (derived from *DateLocator*) now implement the *tick_values* method. This is expected of all Locators derived from *Locator*.

The Date Locators can now be used easily without creating axes

```
from datetime import datetime
from matplotlib.dates import YearLocator
t0 = datetime(2002, 10, 9, 12, 10)
tf = datetime(2005, 10, 9, 12, 15)
loc = YearLocator()
values = loc.tick_values(t0, tf)
```

OffsetBoxes now support clipping

*Artist*s draw onto objects of type *OffsetBox* through *DrawingArea* and *TextArea*. The *TextArea* calculates the required space for the text and so the text is always within the bounds, for this nothing has changed.

However, *DrawingArea* acts as a parent for zero or more *Artist*s that draw on it and may do so beyond the bounds. Now child *Artist*s can be clipped to the bounds of the *DrawingArea*.

OffsetBoxes now considered by tight_layout

When *tight_layout()* or *Figure.tight_layout* or *GridSpec.tight_layout()* is called, *OffsetBox*es that are anchored outside the axes will not get chopped out. The *OffsetBox*es will also not get overlapped by other axes in case of multiple subplots.

Per-page pdf notes in multi-page pdfs (PdfPages)

Add a new method *attach_note()* to the PdfPages class, allowing the attachment of simple text notes to pages in a multi-page pdf of figures. The new note is visible in the list of pdf annotations in a viewer that has this facility (Adobe Reader, OSX Preview, Skim, etc.). Per default the note itself is kept off-page to prevent it to appear in print-outs.

PdfPages.attach_note needs to be called before *savefig* in order to be added to the correct figure.

Updated fignum_exists to take figure name

Added the ability to check the existence of a figure using its name instead of just the figure number. Example:

```
figure('figure')
fignum_exists('figure') #true
```

8.7.10 ToolManager

Federico Ariza wrote the new *ToolManager* that comes as replacement for *NavigationToolbar2*

ToolManager offers a new way of looking at the user interactions with the figures. Before we had the *NavigationToolbar2* with its own tools like `zoom/pan/home/save/ ...` and also we had the shortcuts like `yscale/grid/quit/.....` *ToolManager* relocate all those actions as `Tools` (located in *backend_tools*), and defines a way to access/trigger/reconfigure them.

The `Toolbars` are replaced for `ToolContainers` that are just GUI interfaces to `trigger` the tools. But don't worry the default backends include a `ToolContainer` called `toolbar`

Note: At the moment, we release this primarily for feedback purposes and should be treated as experimental until further notice as API changes will occur. For the moment the *ToolManager* works only with the GTK3 and Tk backends. Make sure you use one of those. Port for the rest of the backends is coming soon.

To activate the *ToolManager* include the following at the top of your file

```
>>> matplotlib.rcParams['toolbar'] = 'toolmanager'
```

Interact with the ToolContainer

The most important feature is the ability to easily reconfigure the ToolContainer (aka toolbar). For example, if we want to remove the "forward" button we would just do.

```
>>> fig.canvas.manager.toolmanager.remove_tool('forward')
```

Now if you want to programmatically trigger the "home" button

```
>>> fig.canvas.manager.toolmanager.trigger_tool('home')
```

New Tools for ToolManager

It is possible to add new tools to the ToolManager

A very simple tool that prints "You're awesome" would be:

```
from matplotlib.backend_tools import ToolBase
class AwesomeTool(ToolBase):
    def trigger(self, *args, **kwargs):
        print("You're awesome")
```

To add this tool to *ToolManager*

```
>>> fig.canvas.manager.toolmanager.add_tool('Awesome', AwesomeTool)
```

If we want to add a shortcut ("d") for the tool

```
>>> fig.canvas.manager.toolmanager.update_keymap('Awesome', 'd')
```

To add it to the toolbar inside the group 'foo'

```
>>> fig.canvas.manager.toolbar.add_tool('Awesome', 'foo')
```

There is a second class of tools, "Toggleable Tools", this are almost the same as our basic tools, just that belong to a group, and are mutually exclusive inside that group. For tools derived from *ToolToggleBase* there are two basic methods *enable* and *disable* that are called automatically whenever it is toggled.

A full example is located in /gallery/user_interfaces/toolmanager_sgskip

8.7.11 cbook.is_sequence_of_strings recognizes string objects

This is primarily how pandas stores a sequence of strings

```
import pandas as pd
import matplotlib.cbook as cbook

a = np.array(['a', 'b', 'c'])
print(cbook.is_sequence_of_strings(a))  # True

a = np.array(['a', 'b', 'c'], dtype=object)
print(cbook.is_sequence_of_strings(a))  # True

s = pd.Series(['a', 'b', 'c'])
print(cbook.is_sequence_of_strings(s))  # True
```

Previously, the last two prints returned false.

8.7.12 New close-figs argument for plot directive

Matplotlib has a sphinx extension plot_directive that creates plots for inclusion in sphinx documents. Matplotlib 1.5 adds a new option to the plot directive - close-figs - that closes any previous figure windows before creating the plots. This can help avoid some surprising duplicates of plots when using plot_directive.

8.7.13 Support for URL string arguments to imread

The *imread()* function now accepts URL strings that point to remote PNG files. This circumvents the generation of a HTTPResponse object directly.

8.7.14 Display hook for animations in the IPython notebook

Animation instances gained a _repr_html_ method to support inline display of animations in the notebook. The method used to display is controlled by the animation.html rc parameter, which currently supports values of none and html5. none is the default, performing no display. html5 converts the animation to an h264 encoded video, which is embedded directly in the notebook.

Users not wishing to use the _repr_html_ display hook can also manually call the to_html5_video method to get the HTML and display using IPython's HTML display class:

```
from IPython.display import HTML
HTML(anim.to_html5_video())
```

8.7.15 Prefixed pkg-config for building

Handling of pkg-config has been fixed in so far as it is now possible to set it using the environment variable PKG_CONFIG. This is important if your toolchain is prefixed. This is done in a simpilar way as setting CC or CXX before building. An example follows.

```
export PKG_CONFIG=x86_64-pc-linux-gnu-pkg-config
```

8.8 New in matplotlib 1.4

Thomas A. Caswell served as the release manager for the 1.4 release.

Table of Contents

Note: matplotlib 1.4 supports Python 2.6, 2.7, 3.3, and 3.4

8.8.1 New colormap

In heatmaps, a green-to-red spectrum is often used to indicate intensity of activity, but this can be problematic for the red/green colorblind. A new, colorblind-friendly colormap is now available at `matplotlib.cm.Wistia`. This colormap maintains the red/green symbolism while achieving deuteranopic legibility through brightness variations. See here for more information.

8.8.2 The nbagg backend

Phil Elson added a new backend, named "nbagg", which enables interactive figures in a live IPython notebook session. The backend makes use of the infrastructure developed for the webagg backend, which itself gives standalone server backed interactive figures in the browser, however nbagg does not require a dedicated matplotlib server as all communications are handled through the IPython Comm machinery.

As with other backends nbagg can be enabled inside the IPython notebook with:

```
import matplotlib
matplotlib.use('nbagg')
```

Once figures are created and then subsequently shown, they will placed in an interactive widget inside the notebook allowing panning and zooming in the same way as any other matplotlib backend. Because figures require a connection to the IPython notebook server for their interactivity, once the notebook is saved, each figure will

be rendered as a static image - thus allowing non-interactive viewing of figures on services such as nbviewer.

8.8.3 New plotting features

Power-law normalization

Ben Gamari added a power-law normalization method, *PowerNorm*. This class maps a range of values to the interval [0,1] with power-law scaling with the exponent provided by the constructor's *gamma* argument. Power law normalization can be useful for, e.g., emphasizing small populations in a histogram.

Fully customizable boxplots

Paul Hobson overhauled the *boxplot()* method such that it is now completely customizable in terms of the styles and positions of the individual artists. Under the hood, *boxplot()* relies on a new function (*boxplot_stats()*), which accepts any data structure currently compatible with *boxplot()*, and returns a list of dictionaries containing the positions for each element of the boxplots. Then a second method, *bxp* is called to draw the boxplots based on the stats.

The *boxplot()* function can be used as before to generate boxplots from data in one step. But now the user has the flexibility to generate the statistics independently, or to modify the output of *boxplot_stats()* prior to plotting with *bxp*.

Lastly, each artist (e.g., the box, outliers, cap, notches) can now be toggled on or off and their styles can be passed in through individual kwargs. See the examples: /gallery/statistics/boxplot and /gallery/statistics/bxp

Added a bool kwarg, `manage_xticks`, which if False disables the management of the ticks and limits on the x-axis by *bxp()*.

Support for datetime axes in 2d plots

Andrew Dawson added support for datetime axes to *contour()*, *contourf()*, *pcolormesh()* and *pcolor()*.

Support for additional spectrum types

Todd Jennings added support for new types of frequency spectrum plots: *magnitude_spectrum()*, *phase_spectrum()*, and *angle_spectrum()*, as well as corresponding functions in mlab.

He also added these spectrum types to *specgram()*, as well as adding support for linear scaling there (in addition to the existing dB scaling). Support for additional spectrum types was also added to *specgram()*.

He also increased the performance for all of these functions and plot types.

Support for detrending and windowing 2D arrays in mlab

Todd Jennings added support for 2D arrays in the *detrend_mean()*, *detrend_none()*, and *detrend()*, as well as adding *apply_window()* which support windowing 2D arrays.

Support for strides in mlab

Todd Jennings added some functions to mlab to make it easier to use numpy strides to create memory-efficient 2D arrays. This includes *stride_repeat()*, which repeats an array to create a 2D array, and *stride_windows()*, which uses a moving window to create a 2D array from a 1D array.

Formatter for new-style formatting strings

Added *StrMethodFormatter* which does the same job as *FormatStrFormatter*, but accepts new-style formatting strings instead of printf-style formatting strings

Consistent grid sizes in streamplots

streamplot() uses a base grid size of 30x30 for both `density=1` and `density=(1, 1)`. Previously a grid size of 30x30 was used for `density=1`, but a grid size of 25x25 was used for `density=(1, 1)`.

Get a list of all tick labels (major and minor)

Added the kwarg 'which' to *Axes.get_xticklabels*, *Axes.get_yticklabels* and *Axis.get_ticklabels*. 'which' can be 'major', 'minor', or 'both' select which ticks to return, like `set_ticks_position()`. If 'which' is `None` then the old behaviour (controlled by the bool *minor*).

Separate horizontal/vertical axes padding support in ImageGrid

The kwarg 'axes_pad' to *mpl_toolkits.axes_grid1.axes_grid.ImageGrid* can now be a tuple if separate horizontal/vertical padding is needed. This is supposed to be very helpful when you have a labelled legend next to every subplot and you need to make some space for legend's labels.

Support for skewed transformations

The *Affine2D* gained additional methods *skew* and *skew_deg* to create skewed transformations. Additionally, matplotlib internals were cleaned up to support using such transforms in *Axes*. This transform is important for some plot types, specifically the Skew-T used in meteorology.

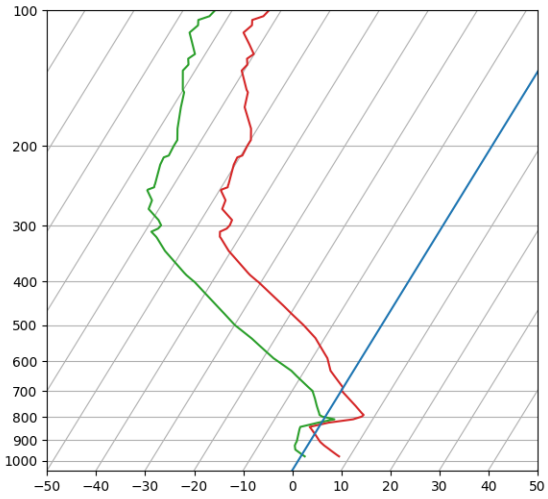


Fig. 8: Skewt

Support for specifying properties of wedge and text in pie charts.

Added the kwargs 'wedgeprops' and 'textprops' to *pie* to accept properties for wedge and text objects in a pie. For example, one can specify wedgeprops = {'linewidth':3} to specify the width of the borders of the wedges in the pie. For more properties that the user can specify, look at the docs for the wedge and text objects.

Fixed the direction of errorbar upper/lower limits

Larry Bradley fixed the *errorbar()* method such that the upper and lower limits (*lolims*, *uplims*, *xlolims*, *xuplims*) now point in the correct direction.

More consistent add-object API for Axes

Added the Axes method *add_image* to put image handling on a par with artists, collections, containers, lines, patches, and tables.

Violin Plots

Per Parker, Gregory Kelsie, Adam Ortiz, Kevin Chan, Geoffrey Lee, Deokjae Donald Seo, and Taesu Terry Lim added a basic implementation for violin plots. Violin plots can be used to represent the distribution of sample data. They are similar to box plots, but use a kernel density estimation function to present a smooth approximation of the data sample used. The added features are:

violin - Renders a violin plot from a collection of statistics. *violin_stats()* - Produces a collection of statistics suitable for rendering a violin plot. *violinplot()* - Creates a violin plot from a set of sample data. This method makes use of *violin_stats()* to process the input data, and *violin_stats()* to do the actual rendering. Users are also free to modify or replace the output of *violin_stats()* in order to customize the violin plots to their liking.

This feature was implemented for a software engineering course at the University of Toronto, Scarborough, run in Winter 2014 by Anya Tafliovich.

More `markevery` options to show only a subset of markers

Rohan Walker extended the `markevery` property in *Line2D*. You can now specify a subset of markers to show with an int, slice object, numpy fancy indexing, or float. Using a float shows markers at approximately equal display-coordinate-distances along the line.

Added size related functions to specialized `Collections`

Added the `get_size` and `set_size` functions to control the size of elements of specialized collections (*AsteriskPolygonCollection* *BrokenBarHCollection* *CircleCollection* *PathCollection* *PolyCollection* *RegularPolyCollection* *StarPolygonCollection*).

Fixed the mouse coordinates giving the wrong theta value in Polar graph

Added code to `transform_non_affine()` to ensure that the calculated theta value was between the range of 0 and 2 * pi since the problem was that the value can become negative after applying the direction and rotation to the theta calculation.

Simple quiver plot for mplot3d toolkit

A team of students in an *Engineering Large Software Systems* course, taught by Prof. Anya Tafliovich at the University of Toronto, implemented a simple version of a quiver plot in 3D space for the mplot3d toolkit as one of their term project. This feature is documented in *quiver()*. The team members are: Ryan Steve D'Souza, Victor B, xbtsw, Yang Wang, David, Caradec Bisesar and Vlad Vassilovski.

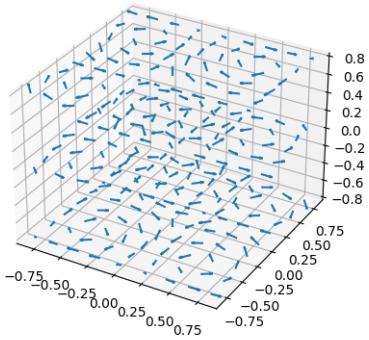


Fig. 9: Quiver3d

polar-plot r-tick locations

Added the ability to control the angular position of the r-tick labels on a polar plot via *set_rlabel_position*.

8.8.4 Date handling

n-d array support for date conversion

Andrew Dawson added support for n-d array handling to *matplotlib.dates.num2date()*, *matplotlib.dates.date2num()* and *matplotlib.dates.datestr2num()*. Support is also added to the unit conversion interfaces *matplotlib.dates.DateConverter* and *matplotlib.units.Registry*.

8.8.5 Configuration (rcParams)

`savefig.transparent` added

Controls whether figures are saved with a transparent background by default. Previously *savefig* always defaulted to a non-transparent background.

`axes.titleweight`

Added rcParam to control the weight of the title

`axes.formatter.useoffset` added

Controls the default value of *useOffset* in *ScalarFormatter*. If `True` and the data range is much smaller than the data average, then an offset will be determined such that the tick labels are meaningful. If `False` then the full number will be formatted in all conditions.

`nbagg.transparent` added

Controls whether nbagg figures have a transparent background. `nbagg.transparent` is `True` by default.

XDG compliance

Matplotlib now looks for configuration files (both rcparams and style) in XDG compliant locations.

8.8.6 `style` package added

You can now easily switch between different styles using the new `style` package:

```
>>> from matplotlib import style
>>> style.use('dark_background')
```

Subsequent plots will use updated colors, sizes, etc. To list all available styles, use:

```
>>> print style.available
```

You can add your own custom `<style name>.mplstyle` files to `~/.matplotlib/stylelib` or call `use` with a URL pointing to a file with `matplotlibrc` settings.

Note that this is an experimental feature, and the interface may change as users test out this new feature.

8.8.7 Backends

Qt5 backend

Martin Fitzpatrick and Tom Badran implemented a Qt5 backend. The differences in namespace locations between Qt4 and Qt5 was dealt with by shimming Qt4 to look like Qt5, thus the Qt5 implementation is the primary implementation. Backwards compatibility for Qt4 is maintained by wrapping the Qt5 implementation.

The Qt5Agg backend currently does not work with IPython's %matplotlib magic.

The 1.4.0 release has a known bug where the toolbar is broken. This can be fixed by:

```
cd path/to/installed/matplotlib
wget https://github.com/matplotlib/matplotlib/pull/3322.diff
# unix2dos 3322.diff (if on windows to fix line endings)
patch -p2 < 3322.diff
```

Qt4 backend

Rudolf Höfler changed the appearance of the subplottool. All sliders are vertically arranged now, buttons for tight layout and reset were added. Furthermore, the subplottool is now implemented as a modal dialog. It was previously a QMainWindow, leaving the SPT open if one closed the plot window.

In the figure options dialog one can now choose to (re-)generate a simple automatic legend. Any explicitly set legend entries will be lost, but changes to the curves' label, linestyle, et cetera will now be updated in the legend.

Interactive performance of the Qt4 backend has been dramatically improved under windows.

The mapping of key-signals from Qt to values matplotlib understands was greatly improved (For both Qt4 and Qt5).

Cairo backends

The Cairo backends are now able to use the cairocffi bindings which are more actively maintained than the pycairo bindings.

Gtk3Agg backend

The Gtk3Agg backend now works on Python 3.x, if the cairocffi bindings are installed.

PDF backend

Added context manager for saving to multi-page PDFs.

8.8.8 Text

Text URLs supported by SVG backend

The SVG backend will now render *Text* objects' url as a link in output SVGs. This allows one to make clickable text in saved figures using the url kwarg of the *Text* class.

Anchored sizebar font

Added the `fontproperties` kwarg to *AnchoredSizeBar* to control the font properties.

8.8.9 Sphinx extensions

The `:context:` directive in the *plot_directive* Sphinx extension can now accept an optional `reset` setting, which will cause the context to be reset. This allows more than one distinct context to be present in documentation. To enable this option, use `:context: reset` instead of `:context:` any time you want to reset the context.

8.8.10 Legend and PathEffects documentation

The *Legend guide* and *Path effects guide* have both been updated to better reflect the full potential of each of these powerful features.

8.8.11 Widgets

Span Selector

Added an option `span_stays` to the *SpanSelector* which makes the selector rectangle stay on the axes after you release the mouse.

8.8.12 GAE integration

Matplotlib will now run on google app engine.

8.9 New in matplotlib 1.3

Table of Contents

Note: matplotlib 1.3 supports Python 2.6, 2.7, 3.2, and 3.3

8.9.1 New in 1.3.1

1.3.1 is a bugfix release, primarily dealing with improved setup and handling of dependencies, and correcting and enhancing the documentation.

The following changes were made in 1.3.1 since 1.3.0.

Enhancements

- Added a context manager for creating multi-page pdfs (see *matplotlib.backends.backend_pdf.PdfPages*).
- The WebAgg backend should now have lower latency over heterogeneous Internet connections.

Bug fixes

- Histogram plots now contain the endline.
- Fixes to the Molleweide projection.
- Handling recent fonts from Microsoft and Macintosh-style fonts with non-ascii metadata is improved.
- Hatching of fill between plots now works correctly in the PDF backend.
- Tight bounding box support now works in the PGF backend.
- Transparent figures now display correctly in the Qt4Agg backend.
- Drawing lines from one subplot to another now works.
- Unit handling on masked arrays has been improved.

Setup and dependencies

- Now works with any version of pyparsing 1.5.6 or later, without displaying hundreds of warnings.
- Now works with 64-bit versions of Ghostscript on MS-Windows.
- When installing from source into an environment without Numpy, Numpy will first be downloaded and built and then used to build matplotlib.
- Externally installed backends are now always imported using a fully-qualified path to the module.
- Works with newer version of wxPython.
- Can now build with a PyCXX installed globally on the system from source.
- Better detection of Gtk3 dependencies.

Testing

- Tests should now work in non-English locales.
- PEP8 conformance tests now report on locations of issues.

8.9.2 New plotting features

`xkcd`-style sketch plotting

To give your plots a sense of authority that they may be missing, Michael Droettboom (inspired by the work of many others in PR #1329) has added an xkcd-style sketch plotting mode. To use it, simply call *`matplotlib.pyplot.xkcd`* before creating your plot. For really fine control, it is also possible to modify each artist's sketch parameters individually with *`matplotlib.artist.Artist.set_sketch_params()`*.

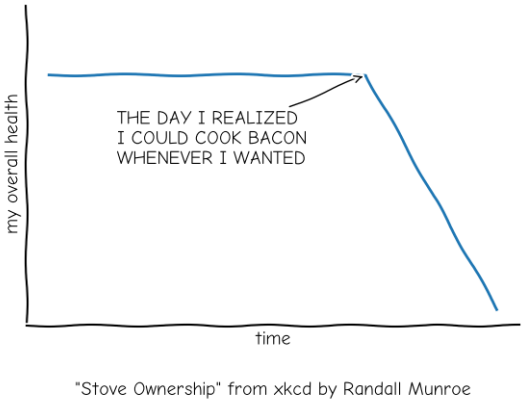


Fig. 10: xkcd

8.9.3 Updated Axes3D.contour methods

Damon McDougall updated the *`tricontour()`* and *`tricontourf()`* methods to allow 3D contour plots on abitrary unstructured user-specified triangulations.

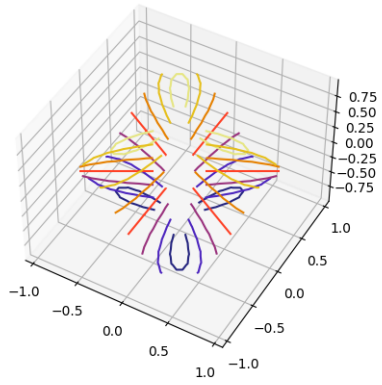


Fig. 11: Tricontour3d

New eventplot plot type

Todd Jennings added a *eventplot()* function to create multiple rows or columns of identical line segments

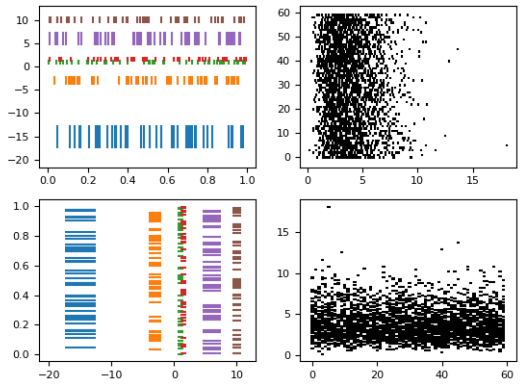


Fig. 12: Eventplot Demo

As part of this feature, there is a new *EventCollection* class that allows for plotting and manipulating rows or columns of identical line segments.

Triangular grid interpolation

Geoffroy Billotey and Ian Thomas added classes to perform interpolation within triangular grids: (*LinearTriInterpolator* and *CubicTriInterpolator*) and a utility class to find the triangles in which points lie (*TrapezoidMapTriFinder*). A helper class to perform mesh refinement and smooth contouring was also added (*UniformTriRefiner*). Finally, a class implementing some basic tools for triangular mesh improvement was added (*TriAnalyzer*).

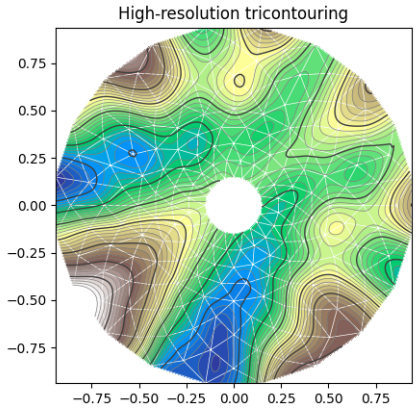


Fig. 13: Tricontour Smooth User

Baselines for stackplot

Till Stensitzki added non-zero baselines to *stackplot()*. They may be symmetric or weighted.

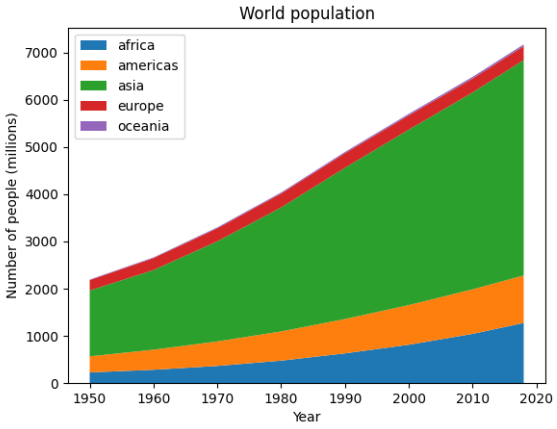


Fig. 14: Stackplot Demo2

Rectangular colorbar extensions

Andrew Dawson added a new keyword argument *extendrect* to *colorbar()* to optionally make colorbar extensions rectangular instead of triangular.

More robust boxplots

Paul Hobson provided a fix to the *boxplot()* method that prevent whiskers from being drawn inside the box for oddly distributed data sets.

Calling subplot() without arguments

A call to *subplot()* without any arguments now acts the same as `subplot(111)` or `subplot(1, 1, 1)` -- it creates one axes for the whole figure. This was already the behavior for both *axes()* and *subplots()*, and now this consistency is shared with *subplot()*.

8.9.4 Drawing

Independent alpha values for face and edge colors

Wes Campaigne modified how *Patch* objects are drawn such that (for backends supporting transparency) you can set different alpha values for faces and edges, by specifying their colors in RGBA format. Note that if you set the alpha attribute for the patch object (e.g. using *set_alpha()* or the `alpha` keyword argument), that value will override the alpha components set in both the face and edge colors.

Path effects on lines

Thanks to Jae-Joon Lee, path effects now also work on plot lines.

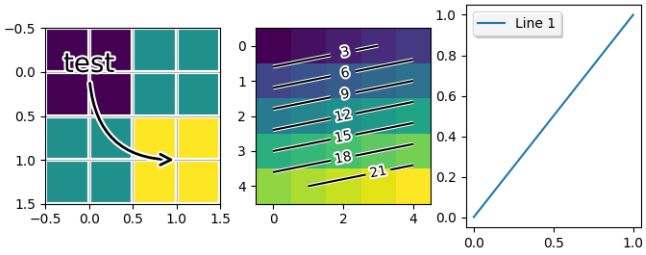


Fig. 15: Patheffect Demo

Easier creation of colormap and normalizer for levels with colors

Phil Elson added the *matplotlib.colors.from_levels_and_colors()* function to easily create a colormap and normalizer for representation of discrete colors for plot types such as *matplotlib.pyplot.pcolormesh()*, with a similar interface to that of *matplotlib.pyplot.contourf*.

Full control of the background color

Wes Campaigne and Phil Elson fixed the Agg backend such that PNGs are now saved with the correct background color when `fig.patch.get_alpha()` is not 1.

Improved `bbox_inches="tight"` functionality

Passing `bbox_inches="tight"` through to *pyplot.savefig* now takes into account *all* artists on a figure - this was previously not the case and led to several corner cases which did not function as expected.

Initialize a rotated rectangle

Damon McDougall extended the *Rectangle* constructor to accept an *angle* kwarg, specifying the rotation of a rectangle in degrees.

8.9.5 Text

Anchored text support

The SVG and pgf backends are now able to save text alignment information to their output formats. This allows to edit text elements in saved figures, using Inkscape for example, while preserving their intended position. For SVG please note that you'll have to disable the default text-to-path conversion (`mpl.rc('svg', fonttype='none')`).

Better vertical text alignment and multi-line text

The vertical alignment of text is now consistent across backends. You may see small differences in text placement, particularly with rotated text.

If you are using a custom backend, note that the *draw_text* renderer method is now passed the location of the baseline, not the location of the bottom of the text bounding box.

Multi-line text will now leave enough room for the height of very tall or very low text, such as superscripts and subscripts.

Left and right side axes titles

Andrew Dawson added the ability to add axes titles flush with the left and right sides of the top of the axes using a new keyword argument *loc* to *title()*.

Improved manual contour plot label positioning

Brian Mattern modified the manual contour plot label positioning code to interpolate along line segments and find the actual closest point on a contour to the requested position. Previously, the closest path vertex was used, which, in the case of straight contours was sometimes quite distant from the requested location. Much more precise label positioning is now possible.

8.9.6 Configuration (rcParams)

Quickly find rcParams

Phil Elson made it easier to search for rcParameters by passing a valid regular expression to *matplotlib.RcParams.find_all()*. *matplotlib.RcParams* now also has a pretty repr and str representation so that search results are printed prettily:

```
>>> import matplotlib
>>> print(matplotlib.rcParams.find_all('\.size'))
RcParams({'font.size': 12,
          'xtick.major.size': 4,
          'xtick.minor.size': 2,
          'ytick.major.size': 4,
          'ytick.minor.size': 2})
```

`axes.xmargin` and `axes.ymargin` added to rcParams

`rcParams["axes.xmargin"]` (default: `0.05`) and `rcParams["axes.ymargin"]` (default: `0.05`) were added to configure the default margins used. Previously they were hard-coded to default to 0, default value of both rcParam values is 0.

Changes to font rcParams

The `font.*` rcParams now affect only text objects created after the rcParam has been set, and will not retroactively affect already existing text objects. This brings their behavior in line with most other rcParams.

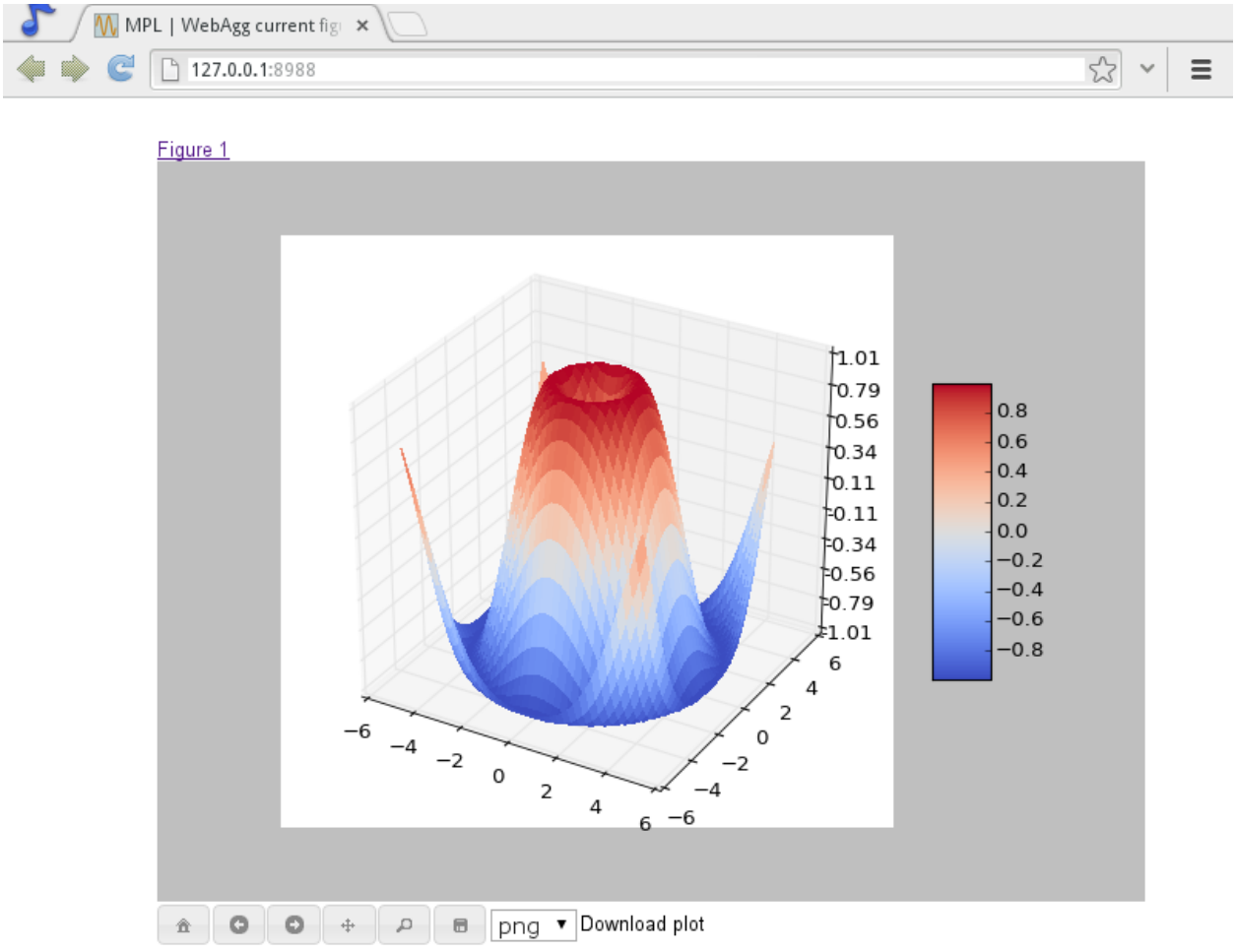
Added `rcParams["savefig.jpeg_quality"]` (default: `95`)

rcParam value `rcParams["savefig.jpeg_quality"]` (default: `95`) was added so that the user can configure the default quality used when a figure is written as a JPEG. The default quality is 95; previously, the default quality was 75. This change minimizes the artifacting inherent in JPEG images, particularly with images that have sharp changes in color as plots often do.

8.9.7 Backends

WebAgg backend

Michael Droettboom, Phil Elson and others have developed a new backend, WebAgg, to display figures in a web browser. It works with animations as well as being fully interactive.



Future versions of matplotlib will integrate this backend with the IPython notebook for a fully web browser based plotting frontend.

Remember save directory

Martin Spacek made the save figure dialog remember the last directory saved to. The default is configurable with the new `rcParams["savefig.directory"]` (default: `'~'`) rcParam in `matplotlibrc`.

8.9.8 Documentation and examples

Numpydoc docstrings

Nelle Varoquaux has started an ongoing project to convert matplotlib's docstrings to numpydoc format. See MEP10 for more information.

Example reorganization

Tony Yu has begun work reorganizing the examples into more meaningful categories. The new gallery page is the fruit of this ongoing work. See MEP12 for more information.

Examples now use subplots()

For the sake of brevity and clarity, most of the examples now use the newer *`subplots()`*, which creates a figure and one (or multiple) axes object(s) in one call. The old way involved a call to *`figure()`*, followed by one (or multiple) *`subplot()`* calls.

8.9.9 Infrastructure

Housecleaning

A number of features that were deprecated in 1.2 or earlier, or have not been in a working state for a long time have been removed. Highlights include removing the Qt version 3 backends, and the FltkAgg and Emf backends. See *Changes in 1.3.x* for a complete list.

New setup script

matplotlib 1.3 includes an entirely rewritten setup script. We now ship fewer dependencies with the tarballs and installers themselves. Notably, pytz, dateutil, pyparsing and six are no longer included with matplotlib. You can either install them manually first, or let pip install them as dependencies along with matplotlib. It is now possible to not include certain subcomponents, such as the unit test data, in the install. See `setup.cfg.template` for more information.

XDG base directory support

On Linux, matplotlib now uses the XDG base directory specification to find the `matplotlibrc` configuration file. `matplotlibrc` should now be kept in `~/.config/matplotlib`, rather than `~/.matplotlib`. If your configuration is found in the old location, it will still be used, but a warning will be displayed.

Catch opening too many figures using pyplot

Figures created through *`pyplot.figure`* are retained until they are explicitly closed. It is therefore common for new users of matplotlib to run out of memory when creating a large series of figures in a loop without closing them.

matplotlib will now display a `RuntimeWarning` when too many figures have been opened at once. By default, this is displayed for 20 or more figures, but the exact number may be controlled using the `figure.max_open_warning` rcParam.

8.10 New in matplotlib 1.2.2

Table of Contents

8.10.1 Improved collections

The individual items of a collection may now have different alpha values and be rendered correctly. This also fixes a bug where collections were always filled in the PDF backend.

8.10.2 Multiple images on same axes are correctly transparent

When putting multiple images onto the same axes, the background color of the axes will now show through correctly.

8.11 New in matplotlib 1.2

Table of Contents

Note: matplotlib 1.2 supports Python 2.6, 2.7, and 3.1

8.11.1 Python 3.x support

Matplotlib 1.2 is the first version to support Python 3.x, specifically Python 3.1 and 3.2. To make this happen in a reasonable way, we also had to drop support for Python versions earlier than 2.6.

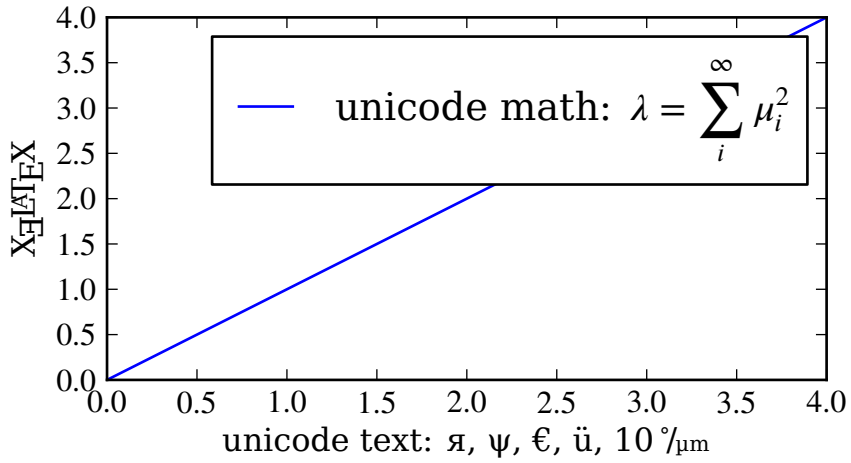
This work was done by Michael Droettboom, the Cape Town Python Users' Group, many others and supported financially in part by the SAGE project.

The following GUI backends work under Python 3.x: Gtk3Cairo, Qt4Agg, TkAgg and MacOSX. The other GUI backends do not yet have adequate bindings for Python 3.x, but continue to work on Python 2.6 and 2.7, particularly the Qt and QtAgg backends (which have been deprecated). The non-GUI backends, such as PDF, PS and SVG, work on both Python 2.x and 3.x.

Features that depend on the Python Imaging Library, such as JPEG handling, do not work, since the version of PIL for Python 3.x is not sufficiently mature.

8.11.2 PGF/TikZ backend

Peter Würtz wrote a backend that allows matplotlib to export figures as drawing commands for LaTeX. These can be processed by PdfLaTeX, XeLaTeX or LuaLaTeX using the PGF/TikZ package. Usage examples and documentation are found in *Typesetting With XeLaTeX/LuaLaTeX*.



8.11.3 Locator interface

Philip Elson exposed the intelligence behind the tick Locator classes with a simple interface. For instance, to get no more than 5 sensible steps which span the values 10 and 19.5:

```
>>> import matplotlib.ticker as mticker
>>> locator = mticker.MaxNLocator(nbins=5)
>>> print(locator.tick_values(10, 19.5))
[ 10.  12.  14.  16.  18.  20.]
```

8.11.4 Tri-Surface Plots

Damon McDougall added a new plotting method for the *mplot3d* toolkit called *plot_trisurf()*.

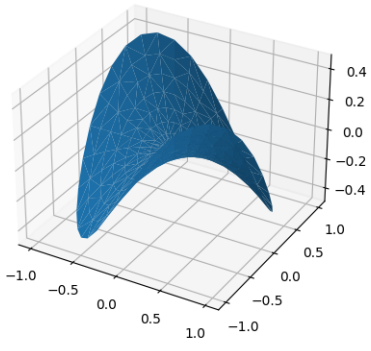
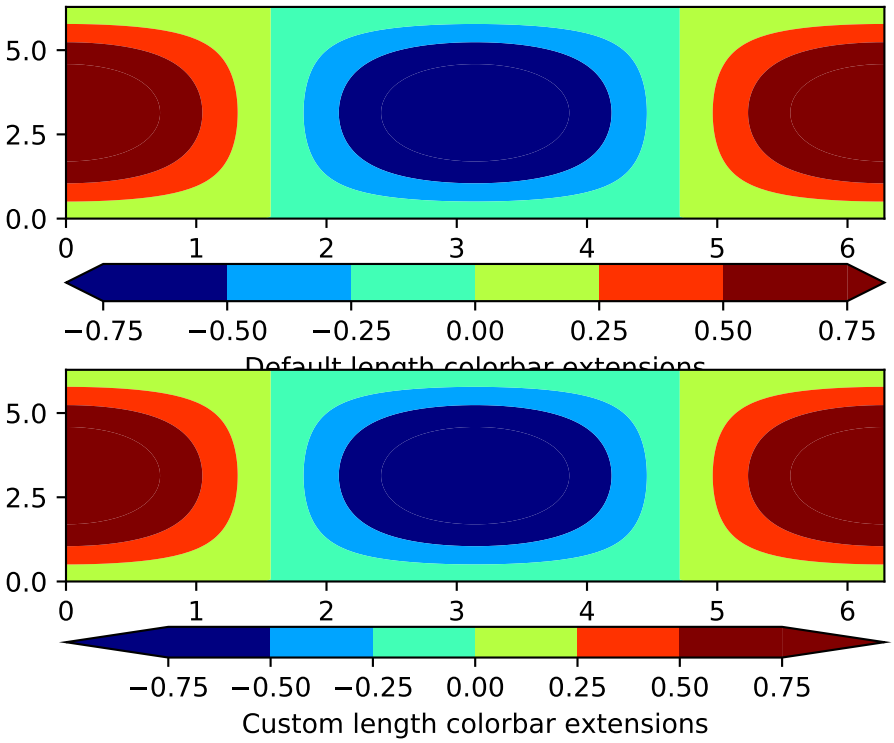


Fig. 16: Trisurf3d

8.11.5 Control the lengths of colorbar extensions

Andrew Dawson added a new keyword argument *extendfrac* to *colorbar()* to control the length of minimum and maximum colorbar extensions.



8.11.6 Figures are picklable

Philip Elson added an experimental feature to make figures picklable for quick and easy short-term storage of plots. Pickle files are not designed for long term storage, are unsupported when restoring a pickle saved in another matplotlib version and are insecure when restoring a pickle from an untrusted source. Having said this, they are useful for short term storage for later modification inside matplotlib.

8.11.7 Set default bounding box in matplotlibrc

Two new defaults are available in the matplotlibrc configuration file: `savefig.bbox`, which can be set to 'standard' or 'tight', and `savefig.pad_inches`, which controls the bounding box padding.

8.11.8 New Boxplot Functionality

Users can now incorporate their own methods for computing the median and its confidence intervals into the *boxplot* method. For every column of data passed to boxplot, the user can specify an accompanying median and confidence interval.

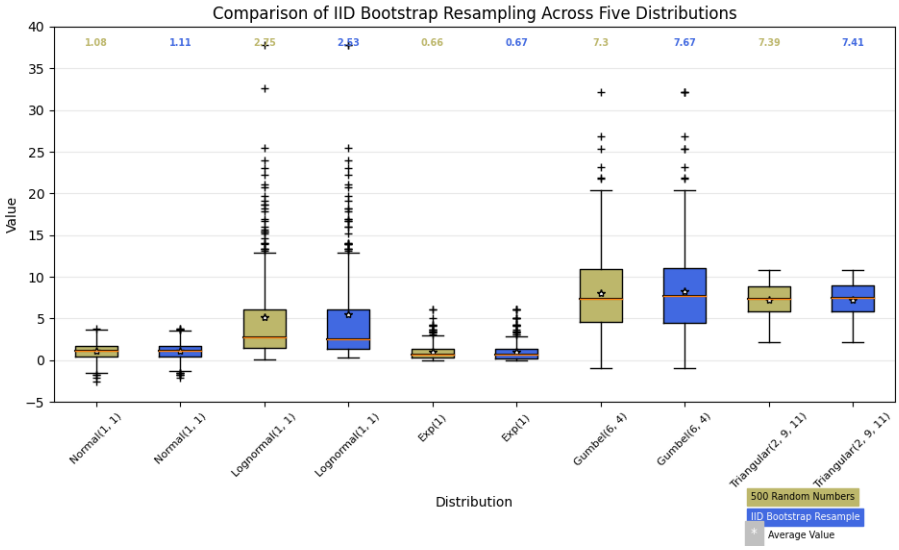


Fig. 17: Boxplot Demo3

8.11.9 New RC parameter functionality

Matthew Emmett added a function and a context manager to help manage RC parameters: *rc_file()* and *rc_context*. To load RC parameters from a file:

```
>>> mpl.rc_file('mpl.rc')
```

To temporarily use RC parameters:

```
>>> with mpl.rc_context(fname='mpl.rc', rc={'text.usetex': True}):
>>>     ...
```

8.11.10 Streamplot

Tom Flannaghan and Tony Yu have added a new *streamplot()* function to plot the streamlines of a vector field. This has been a long-requested feature and complements the existing *quiver()* function for plotting vector fields. In addition to simply plotting the streamlines of the vector field, *streamplot()* allows users to map the colors and/or line widths of the streamlines to a separate parameter, such as the speed or local intensity of the vector field.

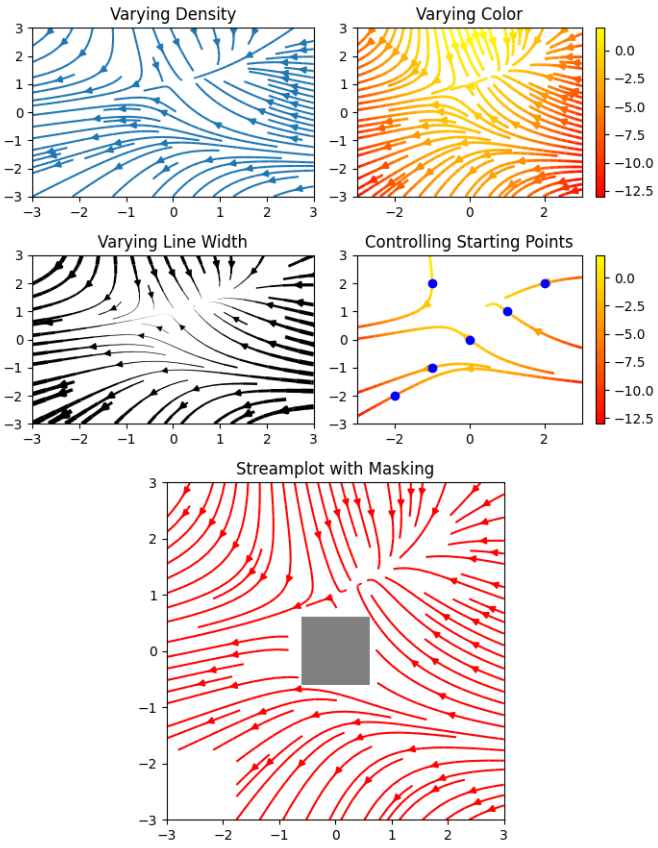


Fig. 18: Plot Streamplot

8.11.11 New hist functionality

Nic Eggert added a new *stacked* kwarg to *hist()* that allows creation of stacked histograms using any of the histogram types. Previously, this functionality was only available by using the `barstacked` histogram type. Now, when `stacked=True` is passed to the function, any of the histogram types can be stacked. The `barstacked` histogram type retains its previous functionality for backwards compatibility.

8.11.12 Updated shipped dependencies

The following dependencies that ship with matplotlib and are optionally installed alongside it have been updated:

- pytz 2012d
- **dateutil 1.5 on Python 2.x,**

  and 2.1 on Python 3.x

8.11.13 Face-centred colors in tripcolor plots

Ian Thomas extended *tripcolor()* to allow one color value to be specified for each triangular face rather than for each point in a triangulation.

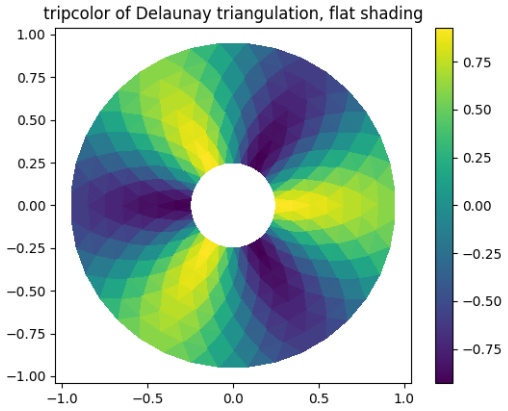


Fig. 19: Tripcolor Demo

8.11.14 Hatching patterns in filled contour plots, with legends

Phil Elson added support for hatching to *contourf()*, together with the ability to use a legend to identify contoured ranges.

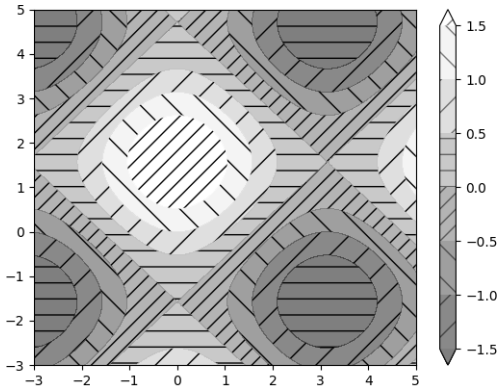


Fig. 20: Contourf Hatching

8.11.15 Known issues in the matplotlib 1.2 release

- When using the Qt4Agg backend with IPython 0.11 or later, the save dialog will not display. This should be fixed in a future version of IPython.

8.12 New in matplotlib 1.1

Table of Contents

- *Other improvements*

Note: matplotlib 1.1 supports Python 2.4 to 2.7

8.12.1 Sankey Diagrams

Kevin Davies has extended Yannick Copin's original Sankey example into a module (*sankey*) and provided new examples (/gallery/specialty_plots/sankey_basics, /gallery/specialty_plots/sankey_links, /gallery/specialty_plots/sankey_rankine).

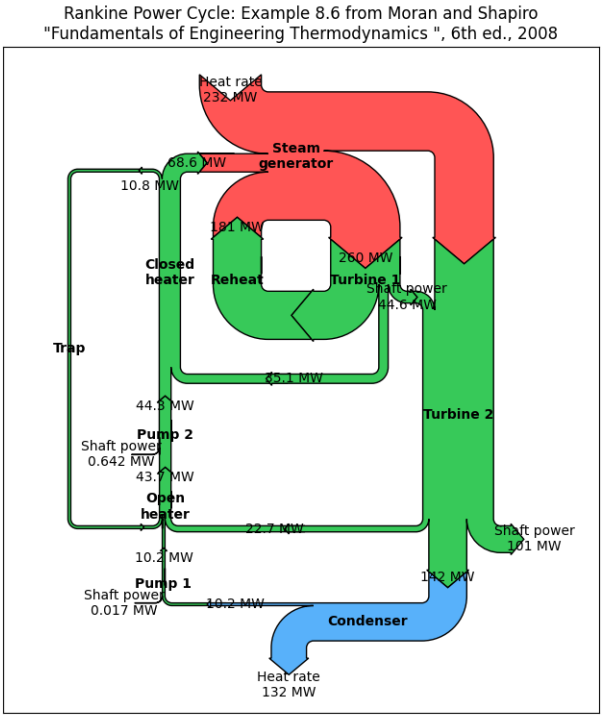


Fig. 21: Sankey Rankine

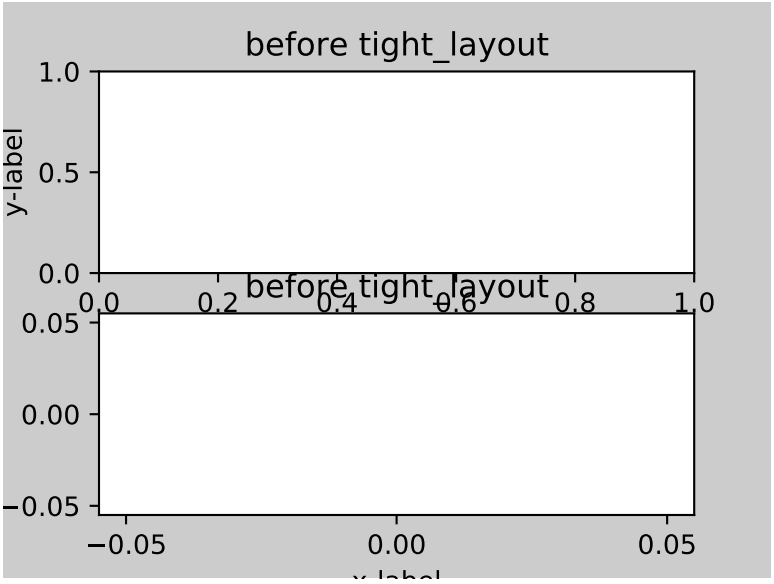
8.12.2 Animation

Ryan May has written a backend-independent framework for creating animated figures. The *animation* module is intended to replace the backend-specific examples formerly in the examples-index listings. Examples using the new framework are in animation-examples-index; see the entrancing `double pendulum <gallery/animation/double_pendulum_sgskip.py>` which uses *matplotlib.animation.Animation.save()* to create the movie below.

This should be considered as a beta release of the framework; please try it and provide feedback.

8.12.3 Tight Layout

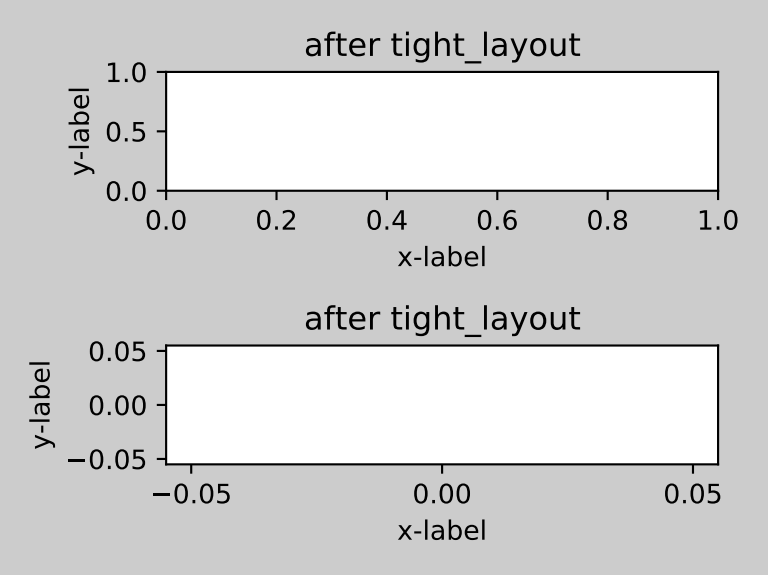
A frequent issue raised by users of matplotlib is the lack of a layout engine to nicely space out elements of the plots. While matplotlib still adheres to the philosophy of giving users complete control over the placement of plot elements, Jae-Joon Lee created the *tight_layout* module and introduced a new command *tight_layout()* to address the most common layout issues.



The usage of this functionality can be as simple as

```
plt.tight_layout()
```

and it will adjust the spacing between subplots so that the axis labels do not overlap with neighboring subplots. A *Tight Layout guide* has been created to show how to use this new tool.



8.12.4 PyQT4, PySide, and IPython

Gerald Storer made the Qt4 backend compatible with PySide as well as PyQT4. At present, however, PySide does not support the PyOS_InputHook mechanism for handling gui events while waiting for text input, so it cannot be used with the new version 0.11 of IPython. Until this feature appears in PySide, IPython users should use the PyQT4 wrapper for QT4, which remains the matplotlib default.

An rcParam entry, "backend.qt4", has been added to allow users to select PyQt4, PyQt4v2, or PySide. The latter two use the Version 2 Qt API. In most cases, users can ignore this rcParam variable; it is available to aid in testing, and to provide control for users who are embedding matplotlib in a PyQt4 or PySide app.

8.12.5 Legend

Jae-Joon Lee has improved plot legends. First, legends for complex plots such as *stem()* plots will now display correctly. Second, the 'best' placement of a legend has been improved in the presence of NANs.

See the *Legend guide* for more detailed explanation and examples.

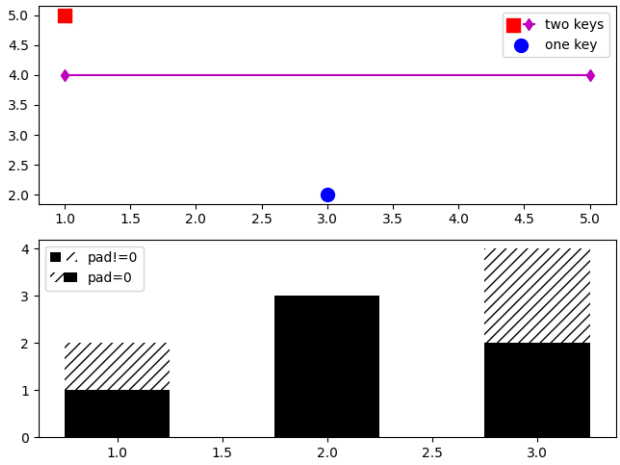


Fig. 22: Legend Demo4

8.12.6 mplot3d

In continuing the efforts to make 3D plotting in matplotlib just as easy as 2D plotting, Ben Root has made several improvements to the *mplot3d* module.

- *Axes3D* has been improved to bring the class towards feature-parity with regular Axes objects
- Documentation for *Getting started* was significantly expanded
- Axis labels and orientation improved
- Most 3D plotting functions now support empty inputs
- Ticker offset display added:

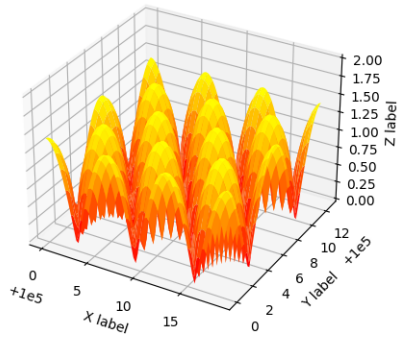


Fig. 23: Offset

- *contourf()* gains *zdir* and *offset* kwargs. You can now do this:

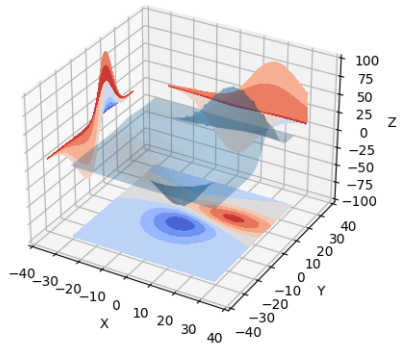


Fig. 24: Contourf3d 2

8.12.7 Numerix support removed

After more than two years of deprecation warnings, Numerix support has now been completely removed from matplotlib.

8.12.8 Markers

The list of available markers for *plot()* and *scatter()* has now been merged. While they were mostly similar, some markers existed for one function, but not the other. This merge did result in a conflict for the 'd' diamond marker. Now, 'd' will be interpreted to always mean "thin" diamond while 'D' will mean "regular" diamond.

Thanks to Michael Droettboom for this effort.

8.12.9 Other improvements

- Unit support for polar axes and *arrow()*
- *PolarAxes* gains getters and setters for "theta_direction", and "theta_offset" to allow for theta to go in either the clock-wise or counter-clockwise direction and to specify where zero degrees should be placed. *set_theta_zero_location()* is an added convenience function.
- Fixed error in argument handling for tri-functions such as *tripcolor()*
- `axes.labelweight` parameter added to rcParams.
- For *imshow()*, *interpolation*='*nearest*' will now always perform an interpolation. A "none" option has been added to indicate no interpolation at all.
- An error in the Hammer projection has been fixed.

- *clabel* for *contour()* now accepts a callable. Thanks to Daniel Hyams for the original patch.
- Jae-Joon Lee added the *HBoxDivider* and *VBoxDivider* classes.
- Christoph Gohlke reduced memory usage in *imshow()*.
- *scatter()* now accepts empty inputs.
- The behavior for 'symlog' scale has been fixed, but this may result in some minor changes to existing plots. This work was refined by ssyr.
- Peter Butterworth added named figure support to *figure()*.
- Michiel de Hoon has modified the MacOSX backend to make its interactive behavior consistent with the other backends.
- Pim Schellart added a new colormap called "cubehelix". Sameer Grover also added a colormap called "coolwarm". See it and all other colormaps *here*.
- Many bug fixes and documentation improvements.

8.13 New in matplotlib 1.0

Table of Contents

8.13.1 HTML5/Canvas backend

Simon Ratcliffe and Ludwig Schwardt have released an HTML5/Canvas backend for matplotlib. The backend is almost feature complete, and they have done a lot of work comparing their html5 rendered images with our core renderer Agg. The backend features client/server interactive navigation of matplotlib figures in an html5 compliant browser.

8.13.2 Sophisticated subplot grid layout

Jae-Joon Lee has written *gridspec*, a new module for doing complex subplot layouts, featuring row and column spans and more. See *Customizing Figure Layouts Using GridSpec and Other Functions* for a tutorial overview.

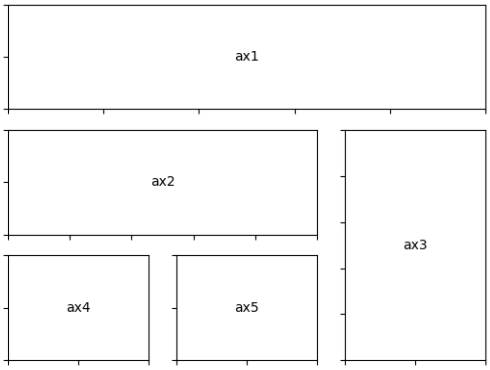


Fig. 25: Demo Gridspec01

8.13.3 Easy pythonic subplots

Fernando Perez got tired of all the boilerplate code needed to create a figure and multiple subplots when using the matplotlib API, and wrote a *subplots()* helper function. Basic usage allows you to create the figure and an array of subplots with numpy indexing (starts with 0). e.g.:

```
fig, axarr = plt.subplots(2, 2)
axarr[0,0].plot([1,2,3])   # upper, left
```

See /gallery/subplots_axes_and_figures/subplot_demo for several code examples.

8.13.4 Contour fixes and and triplot

Ian Thomas has fixed a long-standing bug that has vexed our most talented developers for years. *contourf()* now handles interior masked regions, and the boundaries of line and filled contours coincide.

Additionally, he has contributed a new module *tri* and helper function *triplot()* for creating and plotting unstructured triangular grids.

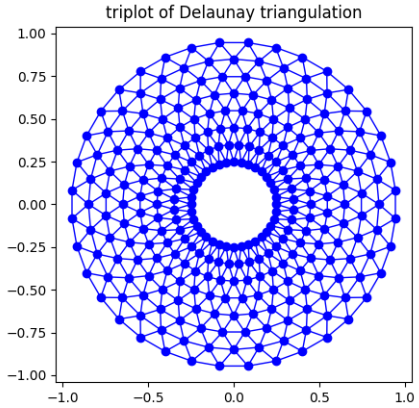


Fig. 26: Triplot Demo

8.13.5 multiple calls to show supported

A long standing request is to support multiple calls to *show()*. This has been difficult because it is hard to get consistent behavior across operating systems, user interface toolkits and versions. Eric Firing has done a lot of work on rationalizing show across backends, with the desired behavior to make show raise all newly created figures and block execution until they are closed. Repeated calls to show should raise newly created figures since the last call. Eric has done a lot of testing on the user interface toolkits and versions and platforms he has access to, but it is not possible to test them all, so please report problems to the mailing list and bug tracker.

8.13.6 mplot3d graphs can be embedded in arbitrary axes

You can now place an mplot3d graph into an arbitrary axes location, supporting mixing of 2D and 3D graphs in the same figure, and/or multiple 3D graphs in a single figure, using the "projection" keyword argument to add_axes or add_subplot. Thanks Ben Root.

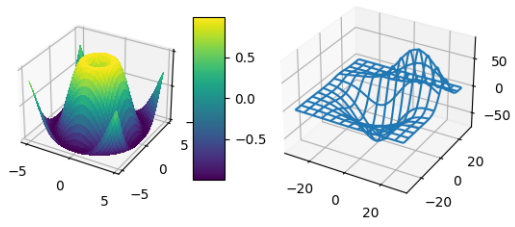


Fig. 27: What's New 1 Subplot3d

8.13.7 tick_params

Eric Firing wrote tick_params, a convenience method for changing the appearance of ticks and tick labels. See pyplot function *tick_params()* and associated Axes method *tick_params()*.

8.13.8 Lots of performance and feature enhancements

- Faster magnification of large images, and the ability to zoom in to a single pixel
- Local installs of documentation work better
- Improved "widgets" -- mouse grabbing is supported
- More accurate snapping of lines to pixel boundaries
- More consistent handling of color, particularly the alpha channel, throughout the API

8.13.9 Much improved software carpentry

The matplotlib trunk is probably in as good a shape as it has ever been, thanks to improved software carpentry. We now have a buildbot which runs a suite of nose regression tests on every svn commit, auto-generating a set of images and comparing them against a set of known-goods, sending emails to developers on failures with a pixel-by-pixel image comparison. Releases and release bugfixes happen in branches, allowing active new feature development to happen in the trunk while keeping the release branches stable. Thanks to Andrew Straw, Michael Droettboom and other matplotlib developers for the heavy lifting.

8.13.10 Bugfix marathon

Eric Firing went on a bug fixing and closing marathon, closing over 100 bugs on the (now-closed) SourceForge bug tracker with help from Jae-Joon Lee, Michael Droettboom, Christoph Gohlke and Michiel de Hoon.

8.14 New in matplotlib 0.99

Table of Contents

8.14.1 New documentation

Jae-Joon Lee has written two new guides *Legend guide* and *Advanced Annotations*. Michael Sarahan has written *Image tutorial*. John Hunter has written two new tutorials on working with paths and transformations: *Path Tutorial* and *Transformations Tutorial*.

8.14.2 mplot3d

Reinier Heeres has ported John Porter's mplot3d over to the new matplotlib transformations framework, and it is now available as a toolkit mpl_toolkits.mplot3d (which now comes standard with all mpl installs). See mplot3d-examples-index and *Getting started*

8.14.3 axes grid toolkit

Jae-Joon Lee has added a new toolkit to ease displaying multiple images in matplotlib, as well as some support for curvilinear grids to support the world coordinate system. The toolkit is included standard with all new mpl installs. See axes_grid1-examples-index, axisartist-examples-index, *What is axes_grid1 toolkit?* and *axisartist*

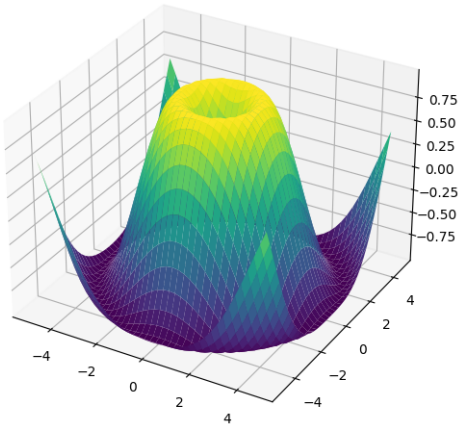


Fig. 28: What's New 99 Mplot3d

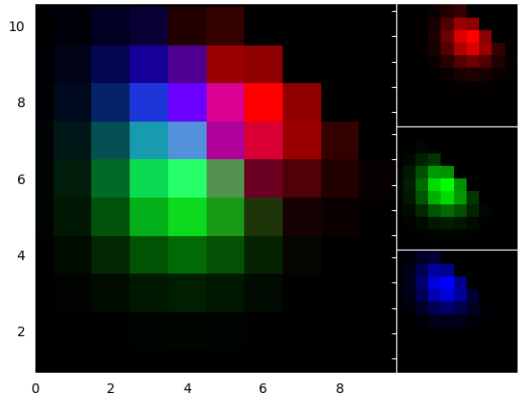


Fig. 29: What's New 99 Axes Grid

8.14.4 Axis spine placement

Andrew Straw has added the ability to place "axis spines" -- the lines that denote the data limits -- in various arbitrary locations. No longer are your axis lines constrained to be a simple rectangle around the figure -- you can turn on or off left, bottom, right and top, as well as "detach" the spine to offset it away from the data. See /gallery/ticks_and_spines/spine_placement_demo and *matplotlib.spines.Spine*.

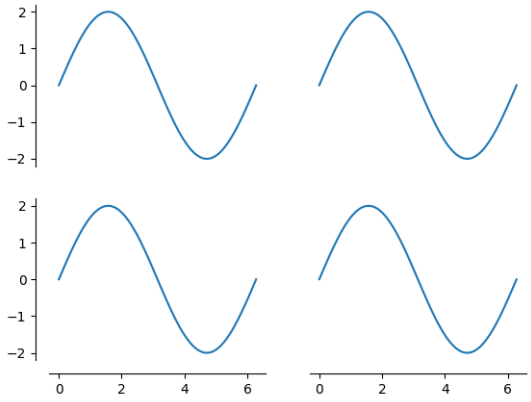


Fig. 30: What's New 99 Spines

8.15 New in matplotlib 0.98.4

Table of Contents

It's been four months since the last matplotlib release, and there are a lot of new features and bug-fixes.

Thanks to Charlie Moad for testing and preparing the source release, including binaries for OS X and Windows for python 2.4 and 2.5 (2.6 and 3.0 will not be available until numpy is available on those releases). Thanks to the many developers who con-

tributed to this release, with contributions from Jae-Joon Lee, Michael Droettboom, Ryan May, Eric Firing, Manuel Metz, Jouni K. Seppänen, Jeff Whitaker, Darren Dale, David Kaplan, Michiel de Hoon and many others who submitted patches

8.15.1 Legend enhancements

Jae-Joon has rewritten the legend class, and added support for multiple columns and rows, as well as fancy box drawing. See *legend()* and *matplotlib.legend.Legend*.

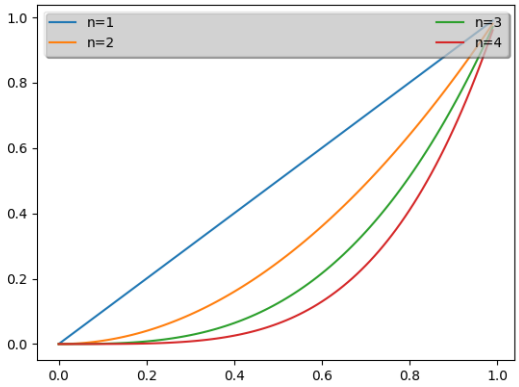


Fig. 31: What's New 98 4 Legend

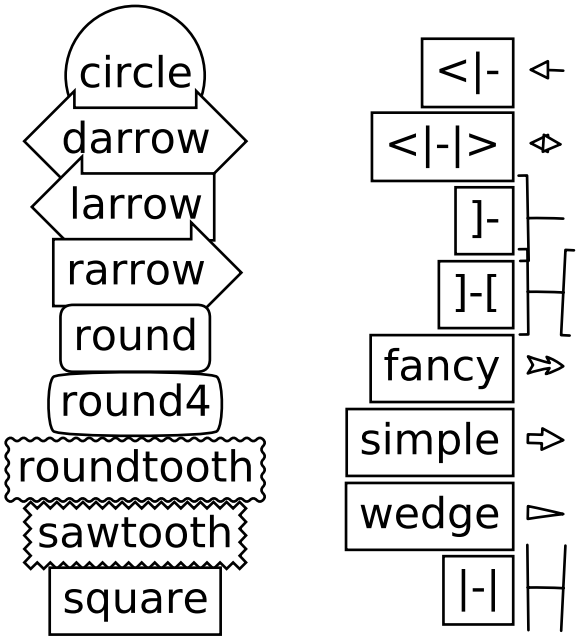
8.15.2 Fancy annotations and arrows

Jae-Joon has added lots of support to annotations for drawing fancy boxes and connectors in annotations. See *annotate()* and *BoxStyle*, *ArrowStyle*, and *ConnectionStyle*.

8.15.3 Native OS X backend

Michiel de Hoon has provided a native Mac OSX backend that is almost completely implemented in C. The backend can therefore use Quartz directly and, depending on the application, can be orders of magnitude faster than the existing backends. In addition, no third-party libraries are needed other than Python and NumPy. The backend is interactive from the usual terminal application on Mac using regular Python. It hasn't been tested with ipython yet, but in principle it should to work there as well. Set 'backend : macosx' in your matplotlibrc file, or run your script with:

```
> python myfile.py -dmacosx
```



8.15.4 psd amplitude scaling

Ryan May did a lot of work to rationalize the amplitude scaling of *psd()* and friends. See /gallery/lines_bars_and_markers/psd_demo. The changes should increase MATLAB compatibility and increase scaling options.

8.15.5 Fill between

Added a *fill_between()* function to make it easier to do shaded region plots in the presence of masked data. You can pass an *x* array and a *ylower* and *yupper* array to fill between, and an optional *where* argument which is a logical mask where you want to do the filling.

8.15.6 Lots more

Here are the 0.98.4 notes from the CHANGELOG:

```
Added mdehoon's native macosx backend from sf patch 2179017 - JDH

Removed the prints in the set_*style commands.  Return the list of
pretty-printed strings instead - JDH

Some of the changes Michael made to improve the output of the
```

(continues on next page)

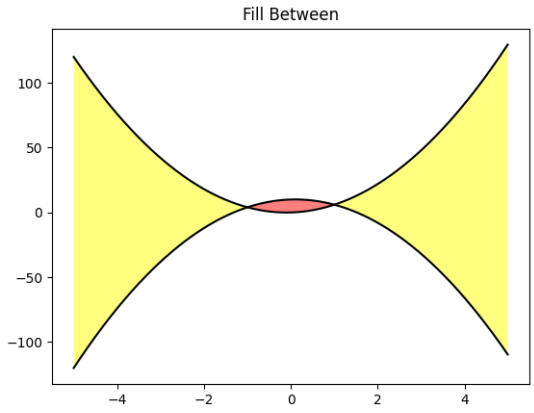


Fig. 32: What's New 98 4 Fill Between

(continued from previous page)

```
property tables in the rest docs broke of made difficult to use
some of the interactive doc helpers, e.g., setp and getp.  Having all
the rest markup in the ipython shell also confused the docstrings.
I added a new rc param docstring.harcopy, to format the docstrings
differently for hardcopy and other use.  The ArtistInspector
could use a little refactoring now since there is duplication of
effort between the rest out put and the non-rest output - JDH

Updated spectral methods (psd, csd, etc.) to scale one-sided
densities by a factor of 2 and, optionally, scale all densities by
the sampling frequency.  This gives better MATLAB
compatibility. -RM

Fixed alignment of ticks in colorbars. -MGD

drop the deprecated "new" keyword of np.histogram() for numpy 1.2
or later.  -JJL

Fixed a bug in svg backend that new_figure_manager() ignores
keywords arguments such as figsize, etc. -JJL

Fixed a bug that the handlelength of the new legend class set too
short when numpoints=1 -JJL

Added support for data with units (e.g., dates) to
Axes.fill_between. -RM

Added fancybox keyword to legend. Also applied some changes for
better look, including baseline adjustment of the multiline texts
so that it is center aligned. -JJL

The transmuter classes in the patches.py are reorganized as
subclasses of the Style classes. A few more box and arrow styles
```

(continues on next page)

(continued from previous page)

```
are added. -JJL

Fixed a bug in the new legend class that didn't allowed a tuple of
coordinate values as loc. -JJL

Improve checks for external dependencies, using subprocess
(instead of deprecated popen*) and distutils (for version
checking) - DSD

Reimplementation of the legend which supports baseline alignment,
multi-column, and expand mode. - JJL

Fixed histogram autoscaling bug when bins or range are given
explicitly (fixes Debian bug 503148) - MM

Added rcParam axes.unicode_minus which allows plain hyphen for
minus when False - JDH

Added scatterpoints support in Legend. patch by Erik Tollerud -
JJL

Fix crash in log ticking. - MGD

Added static helper method BrokenHBarCollection.span_where and
Axes/pyplot method fill_between.  See
examples/pylab/fill_between.py - JDH

Add x_isdata and y_isdata attributes to Artist instances, and use
them to determine whether either or both coordinates are used when
updating dataLim.  This is used to fix autoscaling problems that
had been triggered by axhline, axhspan, axvline, axvspan. - EF

Update the psd(), csd(), cohere(), and specgram() methods of Axes
and the csd() cohere(), and specgram() functions in mlab to be in
sync with the changes to psd().  In fact, under the hood, these
all call the same core to do computations. - RM

Add 'pad_to' and 'sides' parameters to mlab.psd() to allow
controlling of zero padding and returning of negative frequency
components, respectively.  These are added in a way that does not
change the API. - RM

Fix handling of c kwarg by scatter; generalize is_string_like to
accept numpy and numpy.ma string array scalars. - RM and EF

Fix a possible EINTR problem in dviread, which might help when
saving pdf files from the qt backend. - JKS

Fix bug with zoom to rectangle and twin axes - MGD

Added Jae Joon's fancy arrow, box and annotation enhancements --
```

(continues on next page)

(continued from previous page)

```
see examples/pylab_examples/annotation_demo2.py

Autoscaling is now supported with shared axes - EF

Fixed exception in dviread that happened with Minion - JKS

set_xlim, ylim now return a copy of the viewlim array to avoid
modify inplace surprises

Added image thumbnail generating function
matplotlib.image.thumbnail.  See examples/misc/image_thumbnail.py
- JDH

Applied scatleg patch based on ideas and work by Erik Tollerud and
Jae-Joon Lee. - MM

Fixed bug in pdf backend: if you pass a file object for output
instead of a filename, e.g., in a wep app, we now flush the object
at the end. - JKS

Add path simplification support to paths with gaps. - EF

Fix problem with AFM files that don't specify the font's full name
or family name. - JKS

Added 'scilimits' kwarg to Axes.ticklabel_format() method, for
easy access to the set_powerlimits method of the major
ScalarFormatter. - EF

Experimental new kwarg borderpad to replace pad in legend, based
on suggestion by Jae-Joon Lee.  - EF

Allow spy to ignore zero values in sparse arrays, based on patch
by Tony Yu.  Also fixed plot to handle empty data arrays, and
fixed handling of markers in figlegend. - EF

Introduce drawstyles for lines. Transparently split linestyles
like 'steps--' into drawstyle 'steps' and linestyle '--'.  Legends
always use drawstyle 'default'. - MM

Fixed quiver and quiverkey bugs (failure to scale properly when
resizing) and added additional methods for determining the arrow
angles - EF

Fix polar interpolation to handle negative values of theta - MGD

Reorganized cbook and mlab methods related to numerical
calculations that have little to do with the goals of those two
modules into a separate module numerical_methods.py Also, added
ability to select points and stop point selection with keyboard in
ginput and manual contour labeling code.  Finally, fixed contour
```

(continues on next page)

(continued from previous page)

```
labeling bug. - DMK

Fix backtick in Postscript output. - MGD

[ 2089958 ] Path simplification for vector output backends
Leverage the simplification code exposed through path_to_polygons
to simplify certain well-behaved paths in the vector backends
(PDF, PS and SVG).  "path.simplify" must be set to True in
matplotlibrc for this to work.  - MGD

Add "filled" kwarg to Path.intersects_path and
Path.intersects_bbox. - MGD

Changed full arrows slightly to avoid an xpdf rendering problem
reported by Friedrich Hagedorn. - JKS

Fix conversion of quadratic to cubic Bezier curves in PDF and PS
backends. Patch by Jae-Joon Lee. - JKS

Added 5-point star marker to plot command q- EF

Fix hatching in PS backend - MGD

Fix log with base 2 - MGD

Added support for bilinear interpolation in
NonUniformImage; patch by Gregory Lielens. - EF

Added support for multiple histograms with data of
different length - MM

Fix step plots with log scale - MGD

Fix masked arrays with markers in non-Agg backends - MGD

Fix clip_on kwarg so it actually works correctly - MGD

Fix locale problems in SVG backend - MGD

fix quiver so masked values are not plotted - JSW

improve interactive pan/zoom in qt4 backend on windows - DSD

Fix more bugs in NaN/inf handling.  In particular, path
simplification (which does not handle NaNs or infs) will be turned
off automatically when infs or NaNs are present.  Also masked
arrays are now converted to arrays with NaNs for consistent
handling of masks and NaNs - MGD and EF

Added support for arbitrary rasterization resolutions to the SVG
backend. - MW
```

8.16 List of changes to Matplotlib prior to 2015

This is a list of the changes made to Matplotlib from 2003 to 2015. For more recent changes, please refer to the what's new or the API changes.

2015-11-16 Levels passed to contour(f) and tricontour(f) must be in increasing

order.

2015-10-21 Added TextBox widget

2015-10-21 Added get_ticks_direction()

2015-02-27 Added the rcParam 'image.composite_image' to permit users

to decide whether they want the vector graphics backends to combine all images within a set of axes into a single composite image. (If images do not get combined, users can open vector graphics files in Adobe Illustrator or Inkscape and edit each image individually.)

2015-02-19 Rewrite of C++ code that calculates contours to add support for

corner masking. This is controlled by the 'corner_mask' keyword in plotting commands 'contour' and 'contourf'. - IMT

2015-01-23 Text bounding boxes are now computed with advance width rather than

ink area. This may result in slightly different placement of text.

2014-10-27 Allowed selection of the backend using the *MPLBACKEND* environment

variable. Added documentation on backend selection methods.

2014-09-27 Overhauled *colors.LightSource*. Added *LightSource.hillshade* to

allow the independent generation of illumination maps. Added new types of blending for creating more visually appealing shaded relief plots (e.g. `blend_mode="overlay"`, etc, in addition to the legacy "hsv" mode).

2014-06-10 Added Colorbar.remove()

2014-06-07 Fixed bug so radial plots can be saved as ps in py3k.

2014-06-01 Changed the fmt kwarg of errorbar to support the

the mpl convention that "none" means "don't draw it", and to default to the empty string, so that plotting of data points is done with the plot() function defaults. Deprecated use of the None object in place "none".

2014-05-22 Allow the linscale keyword parameter of symlog scale to be

smaller than one.

2014-05-20 Added logic to in FontManager to invalidate font-cache if

if font-family rcparams have changed.

2014-05-16 Fixed the positioning of multi-line text in the PGF backend.

2014-05-14 Added Axes.add_image() as the standard way to add AxesImage

instances to Axes. This improves the consistency with add_artist(), add_collection(), add_container(), add_line(), add_patch(), and add_table().

2014-05-02 Added colorblind-friendly colormap, named 'Wistia'.

2014-04-27 Improved input clean up in Axes.{h|v}lines

Coerce input into a 1D ndarrays (after dealing with units).

2014-04-27 removed un-needed cast to float in stem

2014-04-23 Updated references to "ipython -pylab"

The preferred method for invoking pylab is now using the "%pylab" magic. -Chris G.

2014-04-22 Added (re-)generate a simple automatic legend to "Figure Options"

dialog of the Qt4Agg backend.

2014-04-22 Added an example showing the difference between

interpolation = 'none' and interpolation = 'nearest' in *imshow* when saving vector graphics files.

2014-04-22 Added violin plotting functions. See *Axes.violinplot*,

Axes.violin, *cbook.violin_stats* and *mlab.GaussianKDE* for details.

2014-04-10 Fixed the triangular marker rendering error. The "Up" triangle was

rendered instead of "Right" triangle and vice-versa.

2014-04-08 Fixed a bug in parasite_axes.py by making a list out

of a generator at line 263.

2014-04-02 Added `clipon=False` to patch creation of wedges and shadows

in *pie*.

2014-02-25 In backend_qt4agg changed from using update -> repaint under

windows. See comment in source near `self._priv_update` for longer explanation.

2014-03-27 Added tests for pie ccw parameter. Removed pdf and svg images

from tests for pie linewidth parameter.

2014-03-24 Changed the behaviour of axes to not ignore leading or trailing

patches of height 0 (or width 0) while calculating the x and y axis limits. Patches having both height == 0 and width == 0 are ignored.

2014-03-24 Added bool kwarg (manage_xticks) to boxplot to enable/disable

the managemnet of the xlimits and ticks when making a boxplot. Default in True which maintains current behavior by default.

2014-03-23 Fixed a bug in projections/polar.py by making sure that the theta

value being calculated when given the mouse coordinates stays within the range of 0 and 2 * pi.

2014-03-22 Added the keyword arguments wedgeprops and textprops to pie.

Users can control the wedge and text properties of the pie in more detail, if they choose.

2014-03-17 Bug was fixed in append_axes from the AxesDivider class would not

append axes in the right location with respect to the reference locator axes

2014-03-13 Add parameter 'clockwise' to function pie, True by default.

2014-02-28 Added 'origin' kwarg to *spy*

2014-02-27 Implemented separate horizontal/vertical axes padding to the

ImageGrid in the AxesGrid toolkit

2014-02-27 Allowed markevery property of matplotlib.lines.Line2D to be, an int

numpy fancy index, slice object, or float. The float behaviour turns on markers at approximately equal display-coordinate-distances along the line.

2014-02-25 In backend_qt4agg changed from using update -> repaint under

windows. See comment in source near `self._priv_update` for longer explanation.

2014-01-02 *triplot* now returns the artist it adds and support of line and

marker kwargs has been improved. GBY

2013-12-30 Made streamplot grid size consistent for different types of density

argument. A 30x30 grid is now used for both density=1 and density=(1, 1).

2013-12-03 Added a pure boxplot-drawing method that allow a more complete

customization of boxplots. It takes a list of dicts contains stats. Also created a function (*cbook.boxplot_stats*) that generates the stats needed.

2013-11-28 Added qhull extension module to perform Delaunay triangulation more

robustly than before. It is used by tri.Triangulation (and hence all pyplot.tri* methods) and mlab.griddata. Deprecated matplotlib.delaunay module. - IMT

2013-11-05 Add power-law normalization method. This is useful for,

e.g., showing small populations in a "hist2d" histogram.

2013-10-27 Added get_rlabel_position and set_rlabel_position methods to

PolarAxes to control angular position of radial tick labels.

2013-10-06 Add stride-based functions to mlab for easy creation of 2D arrays

with less memory.

2013-10-06 Improve window and detrend functions in mlab, particulart support for

2D arrays.

2013-10-06 Improve performance of all spectrum-related mlab functions and plots.

2013-10-06 Added support for magnitude, phase, and angle spectrums to

axes.specgram, and support for magnitude, phase, angle, and complex spectrums to mlab-specgram.

2013-10-06 Added magnitude_spectrum, angle_spectrum, and phase_spectrum plots,

as well as magnitude_spectrum, angle_spectrum, phase_spectrum, and complex_spectrum functions to mlab

2013-07-12 Added support for datetime axes to 2d plots. Axis values are passed

through Axes.convert_xunits/Axes.convert_yunits before being used by contour/contourf, pcolormesh and pcolor.

2013-07-12 Allowed matplotlib.dates.date2num, matplotlib.dates.num2date,

and matplotlib.dates.datestr2num to accept n-d inputs. Also factored in support for n-d arrays to matplotlib.dates.DateConverter and matplotlib.units.Registry.

2013-06-26 Refactored the axes module: the axes module is now a folder,

containing the following submodule:

- _subplots.py, containing all the subplots helper methods

- _base.py, containing several private methods and a new _AxesBase class. This _AxesBase class contains all the methods that are not directly linked to plots of the "old" Axes
- _axes.py contains the Axes class. This class now inherits from _AxesBase: it contains all "plotting" methods and labelling methods.

This refactoring should not affect the API. Only private methods are not importable from the axes module anymore.

2013-05-18 Added support for arbitrary rasterization resolutions to the

SVG backend. Previously the resolution was hard coded to 72 dpi. Now the backend class takes a image_dpi argument for its constructor, adjusts the image bounding box accordingly and forwards a magnification factor to the image renderer. The code and results now resemble those of the PDF backend. - MW

2013-05-08 Changed behavior of hist when given stacked=True and normed=True.

Histograms are now stacked first, then the sum is normalized. Previously, each histogram was normalized, then they were stacked.

2013-04-25 Changed all instances of:

from matplotlib import MatplotlibDeprecationWarning as mplDeprecation
to:

from cbook import mplDeprecation

and removed the import into the matplotlib namespace in __init__.py Thomas Caswell

2013-04-15 Added 'axes.xmargin' and 'axes.ymargin' to rpParams to set default

margins on auto-scaleing. - TAC

2013-04-16 Added patheffect support for Line2D objects. -JJL

2013-03-31 Added support for arbitrary unstructured user-specified

triangulations to Axes3D.tricontour[f] - Damon McDougall

2013-03-19 Added support for passing *linestyle* kwarg to *step* so all *plot*

kwargs are passed to the underlying *plot* call. -TAC

2013-02-25 Added classes CubicTriInterpolator, UniformTriRefiner, TriAnalyzer

to matplotlib.tri module. - GBy

2013-01-23 Add 'savefig.directory' to rcParams to remember and fill in the last

directory saved to for figure save dialogs - Martin Spacek

2013-01-13 Add eventplot method to axes and pyplot and EventCollection class

to collections.

2013-01-08 Added two extra titles to axes which are flush with the left and

right edges of the plot respectively. Andrew Dawson

2013-01-07 Add framealpha keyword argument to legend - PO

2013-01-16 Till Stensitzki added a baseline feature to stackplot

2012-12-22 Added classes for interpolation within triangular grids

(LinearTriInterpolator) and to find the triangles in which points lie (TrapezoidMapTriFinder) to matplotlib.tri module. - IMT

2012-12-05 Added MatplotlibDeprecationWarning class for signaling deprecation.

Matplotlib developers can use this class as follows:

from matplotlib import MatplotlibDeprecationWarning as mplDeprecation

In light of the fact that Python builtin DeprecationWarnings are ignored by default as of Python 2.7, this class was put in to allow for the signaling of deprecation, but via UserWarnings which are not ignored by default. - PI

2012-11-27 Added the *mtext* parameter for supplying matplotlib.text.Text

instances to RendererBase.draw_tex and RendererBase.draw_text. This allows backends to utilize additional text attributes, like the alignment of text elements. - pwuertz

2012-11-26 deprecate matplotlib/mpl.py, which was used only in pylab.py and is

now replaced by the more suitable `import matplotlib as mpl`. - PI

2012-11-25 Make rc_context available via pyplot interface - PI

2012-11-16 plt.set_cmap no longer throws errors if there is not already

an active colorable artist, such as an image, and just sets up the colormap to use from that point forward. - PI

2012-11-16 Added the funcction _get_rbga_face, which is identical to

_get_rbg_face except it return a (r,g,b,a) tuble, to line2D. Modified Line2D.draw to use _get_rbga_face to get the markerface color so that any alpha set by markerfacecolor will respected. - Thomas Caswell

2012-11-13 Add a symmetric log normalization class to colors.py.

Also added some tests for the normalization class. Till Stensitzki

2012-11-12 Make axes.stem take at least one argument.

Uses a default range(n) when the first arg not provided. Damon McDougall

2012-11-09 Make plt.subplot() without arguments act as subplot(111) - PI

2012-11-08 Replaced plt.figure and plt.subplot calls by the newer, more

convenient single call to plt.subplots() in the documentation examples - PI

2012-10-05 Add support for saving animations as animated GIFs. - JVDP

2012-08-11 Fix path-closing bug in patches.Polygon, so that regardless

of whether the path is the initial one or was subsequently set by set_xy(), get_xy() will return a closed path if and only if get_closed() is True. Thanks to Jacob Vanderplas. - EF

2012-08-05 When a norm is passed to contourf, either or both of the

vmin, vmax attributes of that norm are now respected. Formerly they were respected only if both were specified. In addition, vmin and/or vmax can now be passed to contourf directly as kwargs. - EF

2012-07-24 Contourf handles the extend kwarg by mapping the extended

ranges outside the normed 0-1 range so that they are handled by colormap colors determined by the set_under and set_over methods. Previously the extended ranges were mapped to 0 or 1 so that the "under" and "over" colormap colors were ignored. This change also increases slightly the color contrast for a given set of contour levels. - EF

2012-06-24 Make use of mathtext in tick labels configurable - DSD

2012-06-05 Images loaded through PIL are now ordered correctly - CG

2012-06-02 Add new Axes method and pyplot function, hist2d. - PO

2012-05-31 Remove support for 'cairo.<format>' style of backend specification.

Deprecate 'cairo.format' and 'savefig.extension' rcParams and replace with 'savefig.format'. - Martin Spacek

2012-05-29 pcolormesh now obeys the passed in "edgecolor" kwarg.

To support this, the "shading" argument to pcolormesh now only takes "flat" or "gouraud". To achieve the old "faceted" behavior, pass "edgecolors='k'". - MGD

2012-05-22 Added radius kwarg to pie charts. - HH

2012-05-22 Collections now have a setting "offset_position" to select whether

the offsets are given in "screen" coordinates (default, following the old behavior) or "data" coordinates. This is currently used internally to improve the performance of hexbin.

As a result, the "draw_path_collection" backend methods have grown a new argument "offset_position". - MGD

2012-05-04 Add a new argument to pie charts - startingangle - that

allows one to specify the angle offset for the first wedge of the chart. - EP

2012-05-03 symlog scale now obeys the logarithmic base. Previously, it was

completely ignored and always treated as base e. - MGD

2012-05-03 Allow linscalex/y keyword to symlog scale that allows the size of

the linear portion relative to the logarithmic portion to be adjusted. - MGD

2012-04-14 Added new plot style: stackplot. This new feature supports stacked

area plots. - Damon McDougall

2012-04-06 When path clipping changes a LINETO to a MOVETO, it also

changes any CLOSEPOLY command to a LINETO to the initial point. This fixes a problem with pdf and svg where the CLOSEPOLY would then draw a line to the latest MOVETO position instead of the intended initial position. - JKS

2012-03-27 Add support to ImageGrid for placing colorbars only at

one edge of each column/row. - RMM

2012-03-07 Refactor movie writing into useful classes that make use

of pipes to write image data to ffmpeg or mencoder. Also improve settings for these and the ability to pass custom options. - RMM

2012-02-29 errorevery keyword added to errorbar to enable errorbar

subsampling. fixes issue #600.

2012-02-28 Added plot_trisurf to the mplot3d toolkit. This supports plotting

three dimensional surfaces on an irregular grid. - Damon McDougall

2012-01-23 The radius labels in polar plots no longer use a fixed

padding, but use a different alignment depending on the quadrant they are in. This fixes numerical problems when (rmax - rmin) gets too small. - MGD

2012-01-08 Add axes.streamplot to plot streamlines of a velocity field.

Adapted from Tom Flannaghan streamplot implementation. -TSY

2011-12-29 ps and pdf markers are now stroked only if the line width

is nonzero for consistency with agg, fixes issue #621. - JKS

2011-12-27 Work around an EINTR bug in some versions of subprocess. - JKS

2011-10-25 added support for operatorname to mathtext,

including the ability to insert spaces, such as $operatorname{arg,max}$ - PI

2011-08-18 Change api of Axes.get_tightbbox and add an optional

keyword parameter *call_axes_locator*. - JJL

2011-07-29 A new rcParam "axes.formatter.use_locale" was added, that,

when True, will use the current locale to format tick labels. This means that, for example, in the fr_FR locale, ',' will be used as a decimal separator. - MGD

2011-07-15 The set of markers available in the plot() and scatter()

commands has been unified. In general, this gives more options to both than were previously available, however, there is one backward-incompatible change to the markers in scatter:

> "d" used to mean "diamond", it now means "narrow diamond". "D" can be used for a "diamond".

-MGD

2011-07-13 Fix numerical problems in symlog scale, particularly when

linthresh <= 1.0. Symlog plots may look different if one was depending on the old broken behavior - MGD

2011-07-10 Fixed argument handling error in tripcolor/triplot/tricontour,

issue #203. - IMT

2011-07-08 Many functions added to mplot3d.axes3d to bring Axes3D

objects more feature-parity with regular Axes objects. Significant revisions to the documentation as well. - BVR

2011-07-07 Added compatibility with IPython strategy for picking

a version of Qt4 support, and an rcParam for making the choice explicitly: backend.qt4. - EF

2011-07-07 Modified AutoMinorLocator to improve automatic choice of

the number of minor intervals per major interval, and to allow one to specify this number via a kwarg. - EF

2011-06-28 3D versions of scatter, plot, plot_wireframe, plot_surface,

bar3d, and some other functions now support empty inputs. - BVR

2011-06-22 Add set_theta_offset, set_theta_direction and

set_theta_zero_location to polar axes to control the location of 0 and directionality of theta. - MGD

2011-06-22 Add axes.labelweight parameter to set font weight to axis

labels - MGD.

2011-06-20 Add pause function to pyplot. - EF

2011-06-16 Added *bottom* keyword parameter for the stem command.

Also, implemented a legend handler for the stem plot. - JJL

2011-06-16 Added legend.frameon rcParams. - Mike Kaufman

2011-05-31 Made backend_qt4 compatible with PySide . - Gerald Storer

2011-04-17 Disable keyboard auto-repeat in qt4 backend by ignoring

key events resulting from auto-repeat. This makes constrained zoom/pan work. - EF

2011-04-14 interpolation="nearest" always interpolate images. A new

mode "none" is introduced for no interpolation - JJL

2011-04-03 Fixed broken pick interface to AsteriskCollection objects

used by scatter. - EF

2011-04-01 The plot directive Sphinx extension now supports all of the

features in the Numpy fork of that extension. These include doctest formatting, an 'include-source' option, and a number of new configuration options. - MGD

2011-03-29 Wrapped ViewVCCachedServer definition in a factory function.

This class now inherits from urllib2.HTTPSHandler in order to fetch data from github, but HTTPSHandler is not defined if python was built without SSL support. - DSD

2011-03-10 Update pytz version to 2011c, thanks to Simon Cross. - JKS

2011-03-06 Add standalone tests.py test runner script. - JKS

2011-03-06 Set edgecolor to 'face' for scatter asterisk-type

symbols; this fixes a bug in which these symbols were not responding to the c kwarg. The symbols have no face area, so only the edgecolor is visible. - EF

2011-02-27 Support libpng version 1.5.x; suggestion by Michael

Albert. Changed installation specification to a minimum of libpng version 1.2. - EF

2011-02-20 clabel accepts a callable as an fmt kwarg; modified

patch by Daniel Hyams. - EF

2011-02-18 scatter([], []) is now valid. Also fixed issues

with empty collections - BVR

2011-02-07 Quick workaround for dviread bug #3175113 - JKS

2011-02-05 Add cbook memory monitoring for Windows, using

tasklist. - EF

2011-02-05 Speed up Normalize and LogNorm by using in-place

operations and by using float32 for float32 inputs and for ints of 2 bytes or shorter; based on patch by Christoph Gohlke. - EF

2011-02-04 Changed imshow to use rgba as uint8 from start to

finish, instead of going through an intermediate step as double precision; thanks to Christoph Gohlke. - EF

2011-01-13 Added zdir and offset arguments to contourf3d to

bring contourf3d in feature parity with contour3d. - BVR

2011-01-04 Tag 1.0.1 for release at r8896

2011-01-03 Added display of ticker offset to 3d plots. - BVR

2011-01-03 Turn off tick labeling on interior subplots for

pyplots.subplots when sharex/sharey is True. - JDH

2010-12-29 Implement axes_divider.HBox and VBox. -JJL

2010-11-22 Fixed error with Hammer projection. - BVR

2010-11-12 Fixed the placement and angle of axis labels in 3D plots. - BVR

2010-11-07 New rc parameters examples.download and examples.directory

allow bypassing the download mechanism in get_sample_data. - JKS

2010-10-04 Fix JPEG saving bug: only accept the kwargs documented

by PIL for JPEG files. - JKS

2010-09-15 Remove unused _wxagg extension and numerix.h. - EF

2010-08-25 Add new framework for doing animations with examples.- RM

2010-08-21 Remove unused and inappropriate methods from Tick classes:

set_view_interval, get_minpos, and get_data_interval are properly found in the Axis class and don't need to be duplicated in XTick and YTick. - EF

2010-08-21 Change Axis.set_view_interval() so that when updating an

existing interval, it respects the orientation of that interval, and can enlarge but not reduce the interval. This fixes a bug in which Axis.set_ticks would change the view limits of an inverted axis. Whether set_ticks should be affecting the viewLim at all remains an open question. - EF

2010-08-16 Handle NaN's correctly in path analysis routines. Fixes a

bug where the best location for a legend was not calculated correctly when the line contains NaNs. - MGD

2010-08-14 Fix bug in patch alpha handling, and in bar color kwarg - EF

2010-08-12 Removed all traces of numerix module after 17 months of

deprecation warnings. - EF

2010-08-05 Added keyword arguments 'thetaunits' and 'runits' for polar

plots. Fixed PolarAxes so that when it set default Formatters, it marked them as such. Fixed semilogx and semilogy to no longer blindly reset the ticker information on the non-log axis. Axes.arrow can now accept unitized data. - JRE

2010-08-03 Add support for MPLSETUPCFG variable for custom setup.cfg

filename. Used by sage buildbot to build an mpl w/ no gui support - JDH

2010-08-01 Create directory specified by MPLCONFIGDIR if it does

not exist. - ADS

2010-07-20 Return Qt4's default cursor when leaving the canvas - DSD

2010-07-06 Tagging for mpl 1.0 at r8502

2010-07-05 Added Ben Root's patch to put 3D plots in arbitrary axes,

allowing you to mix 3d and 2d in different axes/subplots or to have multiple 3D plots in one figure. See examples/mplot3d/subplot3d_demo.py - JDH

2010-07-05 Preferred kwarg names in set_xlim are now 'left' and

'right'; in set_ylim, 'bottom' and 'top'; original kwargs are still accepted without complaint. - EF

2010-07-05 TkAgg and FltkAgg backends are now consistent with other

interactive backends: when used in scripts from the command line (not from ipython -pylab), show blocks, and can be called more than once. - EF

2010-07-02 Modified CXX/WrapPython.h to fix "swab bug" on solaris so

mpl can compile on Solaris with CXX6 in the trunk. Closes tracker bug 3022815 - JDH

2010-06-30 Added autoscale convenience method and corresponding

pyplot function for simplified control of autoscaling; and changed axis, set_xlim, and set_ylim so that by default, they turn off the autoscaling on the relevant axis or axes. Therefore one can call set_xlim before plotting a line, for example, and the limits will be retained. - EF

2010-06-20 Added Axes.tick_params and corresponding pyplot function

to control tick and tick label appearance after an Axes has been created. - EF

2010-06-09 Allow Axes.grid to control minor gridlines; allow

Axes.grid and Axis.grid to control major and minor gridlines in the same method call. - EF

2010-06-06 Change the way we do split/dividend adjustments in

finance.py to handle dividends and fix the zero division bug reported in sf bug 2949906 and 2123566. Note that volume is not adjusted because the Yahoo CSV does not distinguish between share split and dividend adjustments making it near impossible to get volume adjustment right (unless we want to guess based on the size of the adjustment or scrape the html tables, which we don't) - JDH

2010-06-06 Updated dateutil to 1.5 and pytz to 2010h.

2010-06-02 Add error_kw kwarg to Axes.bar(). - EF

2010-06-01 Fix pcolormesh() and QuadMesh to pass on kwargs as

appropriate. - RM

2010-05-18 Merge mpl_toolkits.gridspec into the main tree. - JJL

2010-05-04 Improve backend_qt4 so it displays figures with the

correct size - DSD

2010-04-20 Added generic support for connecting to a timer for events. This

adds TimerBase, TimerGTK, TimerQT, TimerWx, and TimerTk to the backends and a new_timer() method to each backend's canvas to allow ease of creating a new timer. - RM

2010-04-20 Added margins() Axes method and pyplot function. - EF

2010-04-18 update the axes_grid documentation. -JJL

2010-04-18 Control MaxNLocator parameters after instantiation,

and via Axes.locator_params method, with corresponding pyplot function. -EF

2010-04-18 Control ScalarFormatter offsets directly and via the

Axes.ticklabel_format() method, and add that to pyplot. -EF

2010-04-16 Add a close_event to the backends. -RM

2010-04-06 modify axes_grid examples to use axes_grid1 and axisartist. -JJL

2010-04-06 rebase axes_grid using axes_grid1 and axisartist modules. -JJL

2010-04-06 axes_grid toolkit is split into two separate modules,

axes_grid1 and axisartist. -JJL

2010-04-05 Speed up import: import pytz only if and when it is

needed. It is not needed if the rc timezone is UTC. - EF

2010-04-03 Added color kwarg to Axes.hist(), based on work by

Jeff Klukas. - EF

2010-03-24 refactor colorbar code so that no cla() is necessary when

mappable is changed. -JJL

2010-03-22 fix incorrect rubber band during the zoom mode when mouse

leaves the axes. -JJL

2010-03-21 x/y key during the zoom mode only changes the x/y limits. -JJL

2010-03-20 Added pyplot.sca() function suggested by JJL. - EF

2010-03-20 Added conditional support for new Tooltip API in gtk backend. - EF

2010-03-20 Changed plt.fig_subplot() to plt.subplots() after discussion on

list, and changed its API to return axes as a numpy object array (with control of dimensions via squeeze keyword). FP.

2010-03-13 Manually brought in commits from branch:

```
  ------------------------------------------------------------------------
  r8191 | leejjoon | 2010-03-13 17:27:57 -0500 (Sat, 13 Mar 2010) | 1 line

fix the bug that handles for scatter are incorrectly set when dpi!=72.
Thanks to Ray Speth for the bug report.
```

2010-03-03 Manually brought in commits from branch via diff/patch (svnmerge is broken):

```
------------------------------------------------------------------------
r8175 | leejjoon | 2010-03-03 10:03:30 -0800 (Wed, 03 Mar 2010) | 1 line

fix arguments of allow_rasterization.draw_wrapper
------------------------------------------------------------------------
r8174 | jdh2358 | 2010-03-03 09:15:58 -0800 (Wed, 03 Mar 2010) | 1 line

added support for favicon in docs build
------------------------------------------------------------------------
r8173 | jdh2358 | 2010-03-03 08:56:16 -0800 (Wed, 03 Mar 2010) | 1 line

applied Mattias get_bounds patch
------------------------------------------------------------------------
r8172 | jdh2358 | 2010-03-03 08:31:42 -0800 (Wed, 03 Mar 2010) | 1 line

fix svnmerge download instructions
------------------------------------------------------------------------
r8171 | jdh2358 | 2010-03-03 07:47:48 -0800 (Wed, 03 Mar 2010) | 1 line
```

2010-02-25 add annotation_demo3.py that demonstrates new functionality. -JJL

2010-02-25 refactor Annotation to support arbitrary Transform as xycoords

or textcoords. Also, if a tuple of two coordinates is provided, they are interpreted as coordinates for each x and y position. -JJL

2010-02-24 Added pyplot.fig_subplot(), to create a figure and a group of

subplots in a single call. This offers an easier pattern than manually making figures and calling add_subplot() multiple times. FP

2010-02-17 Added Gokhan's and Mattias' customizable keybindings patch

for the toolbar. You can now set the keymap.* properties in the matplotlibrc file. Newbindings were added for toggling log scaling on the x-axis. JDH

2010-02-16 Committed TJ's filled marker patch for

left|right|bottom|top|full filled markers. See examples/pylab_examples/filledmarker_demo.py. JDH

2010-02-11 Added 'bootstrap' option to boxplot. This allows bootstrap

estimates of median confidence intervals. Based on an initial patch by Paul Hobson. - ADS

2010-02-06 Added setup.cfg "basedirlist" option to override setting

in setupext.py "basedir" dictionary; added "gnu0" platform requested by Benjamin Drung. - EF

2010-02-06 Added 'xy' scaling option to EllipseCollection. - EF

2010-02-03 Made plot_directive use a custom PlotWarning category, so that

warnings can be turned into fatal errors easily if desired. - FP

2010-01-29 Added draggable method to Legend to allow mouse drag

placement. Thanks Adam Fraser. JDH

2010-01-25 Fixed a bug reported by Olle Engdegard, when using

histograms with stepfilled and log=True - MM

2010-01-16 Upgraded CXX to 6.1.1 - JDH

2009-01-16 Don't create minor ticks on top of existing major

ticks. Patch by Neil Crighton. -ADS

2009-01-16 Ensure three minor ticks always drawn (SF# 2924245). Patch

by Neil Crighton. -ADS

2010-01-16 Applied patch by Ian Thomas to fix two contouring

problems: now contourf handles interior masked regions, and the boundaries of line and filled contours coincide. - EF

2009-01-11 The color of legend patch follows the rc parameters

axes.facecolor and axes.edgecolor. -JJL

2009-01-11 adjustable of Axes can be "box-forced" which allow

sharing axes. -JJL

2009-01-11 Add add_click and pop_click methods in

BlockingContourLabeler. -JJL

2010-01-03 Added rcParams['axes.color_cycle'] - EF

2010-01-03 Added Pierre's qt4 formlayout editor and toolbar button - JDH

2009-12-31 Add support for using math text as marker symbols (Thanks to tcb)

- MGD

2009-12-31 Commit a workaround for a regression in PyQt4-4.6.{0,1} - DSD

2009-12-22 Fix cmap data for gist_earth_r, etc. -JJL

2009-12-20 spines: put spines in data coordinates, add set_bounds()

call. -ADS

2009-12-18 Don't limit notch size in boxplot to q1-q3 range, as this

is effectively making the data look better than it is. - ADS

2009-12-18 mlab.prctile handles even-length data, such that the median

is the mean of the two middle values. - ADS

2009-12-15 Add raw-image (unsampled) support for the ps backend. - JJL

2009-12-14 Add patch_artist kwarg to boxplot, but keep old default.

Convert boxplot_demo2.py to use the new patch_artist. - ADS

2009-12-06 axes_grid: reimplemented AxisArtist with FloatingAxes support.

Added new examples. - JJL

2009-12-01 Applied Laurent Dufrechou's patch to improve blitting with

the qt4 backend - DSD

2009-11-13 The pdf backend now allows changing the contents of

a pdf file's information dictionary via PdfPages.infodict. - JKS

2009-11-12 font_manager.py should no longer cause EINTR on Python 2.6

(but will on the 2.5 version of subprocess). Also the fc-list command in that file was fixed so now it should actually find the list of fontconfig fonts. - JKS

2009-11-10 Single images, and all images in renderers with

option_image_nocomposite (i.e. agg, macosx and the svg backend when rcParams['svg.image_noscale'] is True), are now drawn respecting the zorder relative to other artists. (Note that there may now be inconsistencies across backends when more than one image is drawn at varying zorders, but this change introduces correct behavior for the backends in which it's easy to do so.)

2009-10-21 Make AutoDateLocator more configurable by adding options

to control the maximum and minimum number of ticks. Also add control of the intervals to be used for ticking. This does not change behavior but opens previously hard-coded behavior to runtime modification`. - RMM

2009-10-19 Add "path_effects" support for Text and Patch. See

examples/pylab_examples/patheffect_demo.py -JJL

2009-10-19 Add "use_clabeltext" option to clabel. If True, clabels

will be created with ClabelText class, which recalculates rotation angle of the label during the drawing time. -JJL

2009-10-16 Make AutoDateFormatter actually use any specified

timezone setting.This was only working correctly when no timezone was specified. - RMM

2009-09-27 Beginnings of a capability to test the pdf backend. - JKS

2009-09-27 Add a savefig.extension rcparam to control the default

filename extension used by savefig. - JKS

2009-09-21 Tagged for release 0.99.1

2009-09-20 Fix usetex spacing errors in pdf backend. - JKS

2009-09-20 Add Sphinx extension to highlight IPython console sessions,

originally authored (I think) by Michael Droetboom. - FP

2009-09-20 Fix off-by-one error in dviread.Tfm, and additionally protect

against exceptions in case a dvi font is missing some metrics. - JKS

2009-09-15 Implement draw_text and draw_tex method of backend_base using

the textpath module. Implement draw_tex method of the svg backend. - JJL

2009-09-15 Don't fail on AFM files containing floating-point bounding boxes - JKS

2009-09-13 AxesGrid

[add modified version of colorbar. Add colorbar] location howto. - JJL

2009-09-07 AxesGrid

[implemented axisline style.] Added a demo examples/axes_grid/demo_axisline_style.py- JJL

2009-09-04 Make the textpath class as a separate module

(textpath.py). Add support for mathtext and tex.- JJL

2009-09-01 Added support for Gouraud interpolated triangles.

pcolormesh now accepts shading='gouraud' as an option. - MGD

2009-08-29 Added matplotlib.testing package, which contains a Nose

plugin and a decorator that lets tests be marked as KnownFailures - ADS

2009-08-20 Added scaled dict to AutoDateFormatter for customized

scales - JDH

2009-08-15 Pyplot interface: the current image is now tracked at the

figure and axes level, addressing tracker item 1656374. - EF

2009-08-15 Docstrings are now manipulated with decorators defined

in a new module, docstring.py, thanks to Jason Coombs. - EF

2009-08-14 Add support for image filtering for agg back end. See the example

demo_agg_filter.py. -JJL

2009-08-09 AnnotationBbox added. Similar to Annotation, but works with

OffsetBox instead of Text. See the example demo_annotation_box.py. -JJL

2009-08-07 BboxImage implemented. Two examples, demo_bboximage.py and

demo_ribbon_box.py added. - JJL

2009-08-07 In an effort to simplify the backend API, all clipping rectangles

and paths are now passed in using GraphicsContext objects, even on collections and images. Therefore:

draw_path_collection(self, master_transform, cliprect, clippath,

clippath_trans, paths, all_transforms, offsets, offsetTrans, facecolors, edgecolors, linewidths, linestyles, antialiaseds, urls)

becomes:

draw_path_collection(self, gc, master_transform, paths, all_transforms,

offsets, offsetTrans, facecolors, edgecolors, linewidths, linestyles, antialiaseds, urls)

draw_quad_mesh(self, master_transform, cliprect, clippath,

clippath_trans, meshWidth, meshHeight, coordinates, offsets, offsetTrans, facecolors, antialiased, showedges)

becomes:

draw_quad_mesh(self, gc, master_transform, meshWidth, meshHeight,

coordinates, offsets, offsetTrans, facecolors, antialiased, showedges)

draw_image(self, x, y, im, bbox, clippath=None, clippath_trans=None)

becomes:

draw_image(self, gc, x, y, im)

- MGD

2009-08-06 Tagging the 0.99.0 release at svn r7397 - JDH

- fixed an alpha colormapping bug posted on sf 2832575
- fix typo in axes_divider.py. use nanmin, nanmax in angle_helper.py (patch by Christoph Gohlke)
- remove dup gui event in enter/leave events in gtk
- lots of fixes for os x binaries (Thanks Russell Owen)
- attach gtk events to mpl events -- fixes sf bug 2816580
- applied sf patch 2815064 (middle button events for wx) and patch 2818092 (resize events for wx)
- fixed boilerplate.py so it doesn't break the ReST docs.
- removed a couple of cases of mlab.load
- fixed rec2csv win32 file handle bug from sf patch 2831018
- added two examples from Josh Hemann: examples/pylab_examples/barchart_demo2.py and examples/pylab_examples/boxplot_demo2.py
- handled sf bugs 2831556 and 2830525; better bar error messages and backend driver configs
- added miktex win32 patch from sf patch 2820194
- apply sf patches 2830233 and 2823885 for osx setup and 64 bit; thanks Michiel

2009-08-04 Made cbook.get_sample_data make use of the ETag and Last-Modified

headers of mod_dav_svn. - JKS

2009-08-03 Add PathCollection; modify contourf to use complex

paths instead of simple paths with cuts. - EF

2009-08-03 Fixed boilerplate.py so it doesn't break the ReST docs. - JKS

2009-08-03 pylab no longer provides a load and save function. These

are available in matplotlib.mlab, or you can use numpy.loadtxt and numpy.savetxt for text files, or np.save and np.load for binary numpy arrays. - JDH

2009-07-31 Added cbook.get_sample_data for urllib enabled fetching and

caching of data needed for examples. See examples/misc/sample_data_demo.py - JDH

2009-07-31 Tagging 0.99.0.rc1 at 7314 - MGD

2009-07-30 Add set_cmap and register_cmap, and improve get_cmap,

to provide convenient handling of user-generated colormaps. Reorganized _cm and cm modules. - EF

2009-07-28 Quiver speed improved, thanks to tip by Ray Speth. -EF

2009-07-27 Simplify argument handling code for plot method. -EF

2009-07-25 Allow "plot(1, 2, 'r*')" to work. - EF

2009-07-22 Added an 'interp' keyword to griddata so the faster linear

interpolation method can be chosen. Default is 'nn', so default behavior (using natural neighbor method) is unchanged (JSW)

2009-07-22 Improved boilerplate.py so that it generates the correct

signatures for pyplot functions. - JKS

2009-07-19 Fixed the docstring of Axes.step to reflect the correct

meaning of the kwargs "pre" and "post" - See SF bug https://sourceforge.net/tracker/index.php?func=detail&aid=2823304&group_id=80706&ati
- JDH

2009-07-18 Fix support for hatches without color fills to pdf and svg

backends. Add an example of that to hatch_demo.py. - JKS

2009-07-17 Removed fossils from swig version of agg backend. - EF

2009-07-14 initial submission of the annotation guide. -JJL

2009-07-14 axes_grid

[minor improvements in anchored_artists and] inset_locator. -JJL

2009-07-14 Fix a few bugs in ConnectionStyle algorithms. Add

ConnectionPatch class. -JJL

2009-07-11 Added a fillstyle Line2D property for half filled markers

-- see examples/pylab_examples/fillstyle_demo.py JDH

2009-07-08 Attempt to improve performance of qt4 backend, do not call

qApp.processEvents while processing an event. Thanks Ole Streicher for tracking this down - DSD

2009-06-24 Add withheader option to mlab.rec2csv and changed

use_mrecords default to False in mlab.csv2rec since this is partially broken - JDH

2009-06-24 backend_agg.draw_marker quantizes the main path (as in the

draw_path). - JJL

2009-06-24 axes_grid: floating axis support added. - JJL

2009-06-14 Add new command line options to backend_driver.py to support

running only some directories of tests - JKS

2009-06-13 partial cleanup of mlab and its importation in pylab - EF

2009-06-13 Introduce a rotation_mode property for the Text artist. See

examples/pylab_examples/demo_text_rotation_mode.py -JJL

2009-06-07 add support for bz2 files per sf support request 2794556 -

JDH

2009-06-06 added a properties method to the artist and inspector to

return a dict mapping property name -> value; see sf feature request 2792183 - JDH

2009-06-06 added Neil's auto minor tick patch; sf patch #2789713 - JDH

2009-06-06 do not apply alpha to rgba color conversion if input is

already rgba - JDH

2009-06-03 axes_grid

[Initial check-in of curvelinear grid support. See] examples/axes_grid/demo_curvelinear_grid.py - JJL

2009-06-01 Add set_color method to Patch - EF

2009-06-01 Spine is now derived from Patch - ADS

2009-06-01 use cbook.is_string_like() instead of isinstance() for spines - ADS

2009-06-01 cla() support for spines - ADS

2009-06-01 Removed support for gtk < 2.4. - EF

2009-05-29 Improved the animation_blit_qt4 example, which was a mix

of the object-oriented and pylab interfaces. It is now strictly object-oriented - DSD

2009-05-28 Fix axes_grid toolkit to work with spine patch by ADS. - JJL

2009-05-28 Applied fbianco's patch to handle scroll wheel events in

the qt4 backend - DSD

2009-05-26 Add support for "axis spines" to have arbitrary location. -ADS

2009-05-20 Add an empty matplotlibrc to the tests/ directory so that running

tests will use the default set of rcparams rather than the user's config. - RMM

2009-05-19 Axis.grid(): allow use of which='major,minor' to have grid

on major and minor ticks. -ADS

2009-05-18 Make psd(), csd(), and cohere() wrap properly for complex/two-sided

versions, like specgram() (SF #2791686) - RMM

2009-05-18 Fix the linespacing bug of multiline text (#1239682). See

examples/pylab_examples/multiline.py -JJL

2009-05-18 Add *annotation_clip* attr. for text.Annotation class.

If True, annotation is only drawn when the annotated point is inside the axes area. -JJL

2009-05-17 Fix bug(#2749174) that some properties of minor ticks are

not conserved -JJL

2009-05-17 applied Michiel's sf patch 2790638 to turn off gtk event

loop in setupext for pygtk>=2.15.10 - JDH

2009-05-17 applied Michiel's sf patch 2792742 to speed up Cairo and

macosx collections; speedups can be 20x. Also fixes some bugs in which gc got into inconsistent state

---

2008-05-17 Release 0.98.5.3 at r7107 from the branch - JDH

2009-05-13 An optional offset and bbox support in restore_bbox.

Add animation_blit_gtk2.py. -JJL

2009-05-13 psfrag in backend_ps now uses baseline-alignment

when preview.sty is used ((default is bottom-alignment). Also, a small API improvement in OffsetBox-JJL

2009-05-13 When the x-coordinate of a line is monotonically

increasing, it is now automatically clipped at the stage of generating the transformed path in the draw method; this greatly speeds up zooming and panning when one is looking at a short segment of a long time series, for example. - EF

2009-05-11 aspect=1 in log-log plot gives square decades. -JJL

2009-05-08 clabel takes new kwarg, rightside_up; if False, labels

will not be flipped to keep them rightside-up. This allows the use of clabel to make streamfunction arrows, as requested by Evan Mason. - EF

2009-05-07 'labelpad' can now be passed when setting x/y labels. This allows controlling the spacing between the label and its axis. - RMM

2009-05-06 print_ps now uses mixed-mode renderer. Axes.draw rasterize artists whose zorder smaller than rasterization_zorder. -JJL

2009-05-06 Per-artist Rasterization, originally by Eric Bruning. -JJ

2009-05-05 Add an example that shows how to make a plot that updates using data from another process. Thanks to Robert Cimrman - RMM

2009-05-05 Add Axes.get_legend_handles_labels method. - JJL

2009-05-04 Fix bug that Text.Annotation is still drawn while set to not visible. - JJL

2009-05-04 Added TJ's fill_betweenx patch - JDH

2009-05-02 Added options to plotfile based on question from Joseph Smidt and patch by Matthias Michler. - EF

2009-05-01 Changed add_artist and similar Axes methods to return their argument. - EF

2009-04-30 Incorrect eps bbox for landscape mode fixed - JJL

2009-04-28 Fixed incorrect bbox of eps output when usetex=True. - JJL

2009-04-24 Changed use of os.open* to instead use subprocess.Popen. os.popen* are deprecated in 2.6 and are removed in 3.0. - RMM

2009-04-20 Worked on axes_grid documentation. Added axes_grid.inset_locator. - JJL

2009-04-17 Initial check-in of the axes_grid toolkit. - JJL

2009-04-17 Added a support for bbox_to_anchor in offsetbox.AnchoredOffsetbox. Improved a documentation. - JJL

2009-04-16 Fixed a offsetbox bug that multiline texts are not correctly aligned. - JJL

2009-04-16 Fixed a bug in mixed mode renderer that images produced by an rasterizing backend are placed with incorrect size. - JJL

2009-04-14 Added Jonathan Taylor's Reinier Heeres' port of John Porters' mplot3d to svn trunk. Package in mpl_toolkits.mplot3d and demo is examples/mplot3d/demo.py. Thanks Reiner

2009-04-06 The pdf backend now escapes newlines and linefeeds in strings.

Fixes sf bug #2708559; thanks to Tiago Pereira for the report.

2009-04-06 texmanager.make_dvi now raises an error if LaTeX failed to

create an output file. Thanks to Joao Luis Silva for reporting this. - JKS

2009-04-05 _png.read_png() reads 12 bit PNGs (patch from

Tobias Wood) - ADS

2009-04-04 Allow log axis scale to clip non-positive values to

small positive value; this is useful for errorbars. - EF

2009-03-28 Make images handle nan in their array argument.

A helper, cbook.safe_masked_invalid() was added. - EF

2009-03-25 Make contour and contourf handle nan in their Z argument. - EF

2009-03-20 Add AuxTransformBox in offsetbox.py to support some transformation.

anchored_text.py example is enhanced and renamed (anchored_artists.py). - JJL

2009-03-20 Add "bar" connection style for annotation - JJL

2009-03-17 Fix bugs in edge color handling by contourf, found

by Jae-Joon Lee. - EF

2009-03-14 Added 'LightSource' class to colors module for

creating shaded relief maps. shading_example.py added to illustrate usage. - JSW

2009-03-11 Ensure wx version >= 2.8; thanks to Sandro Tosi and

Chris Barker. - EF

2009-03-10 Fix join style bug in pdf. - JKS

2009-03-07 Add pyplot access to figure number list - EF

2009-02-28 hashing of FontProperties accounts current rcParams - JJL

2009-02-28 Prevent double-rendering of shared axis in twinx, twiny - EF

2009-02-26 Add optional bbox_to_anchor argument for legend class - JJL

2009-02-26 Support image clipping in pdf backend. - JKS

2009-02-25 Improve tick location subset choice in FixedLocator. - EF

2009-02-24 Deprecate numerix, and strip out all but the numpy

part of the code. - EF

2009-02-21 Improve scatter argument handling; add an early error
message, allow inputs to have more than one dimension. - EF

2009-02-16 Move plot_directive.py to the installed source tree. Add
support for inline code content - MGD

2009-02-16 Move mathmpl.py to the installed source tree so it is
available to other projects. - MGD

2009-02-14 Added the legend title support - JJL

2009-02-10 Fixed a bug in backend_pdf so it doesn't break when the setting
pdf.use14corefonts=True is used. Added test case in unit/test_pdf_use14corefonts.py. - NGR

2009-02-08 Added a new imsave function to image.py and exposed it in
the pyplot interface - GR

2009-02-04 Some reorgnization of the legend code. anchored_text.py
added as an example. - JJL

2009-02-04 Add extent keyword arg to hexbin - ADS

2009-02-04 Fix bug in mathtext related to dots and ldots - MGD

2009-02-03 Change default joinstyle to round - MGD

2009-02-02 Reduce number of marker XObjects in pdf output - JKS

2009-02-02 Change default resolution on polar plot to 1 - MGD

2009-02-02 Avoid malloc errors in ttconv for fonts that don't have
e.g., PostName (a version of Tahoma triggered this) - JKS

2009-01-30 Remove support for pyExcelerator in exceltools -- use xlwt
instead - JDH

2009-01-29 Document 'resolution' kwarg for polar plots. Support it
when using pyplot.polar, not just Figure.add_axes. - MGD

2009-01-29 Rework the nan-handling/clipping/quantizing/simplification
framework so each is an independent part of a pipeline. Expose the C++-implementation of all of this so it can be used from all Python backends. Add rcParam "path.simplify_threshold" to control the threshold of similarity below which vertices will be removed.

2009-01-26 Improved tight bbox option of the savefig. - JJL

2009-01-26 Make curves and NaNs play nice together - MGD

2009-01-21 Changed the defaults of acorr and xcorr to use

usevlines=True, maxlags=10 and normed=True since these are the best defaults

2009-01-19 Fix bug in quiver argument handling. - EF

2009-01-19 Fix bug in backend_gtk: don't delete nonexistent toolbar. - EF

2009-01-16 Implement bbox_inches option for savefig. If bbox_inches is

"tight", try to determine the tight bounding box. - JJL

2009-01-16 Fix bug in is_string_like so it doesn't raise an

unnecessary exception. - EF

2009-01-16 Fix an infinite recursion in the unit registry when searching

for a converter for a sequence of strings. Add a corresponding test. - RM

2009-01-16 Bugfix of C typedef of MPL_Int64 that was failing on

Windows XP 64 bit, as reported by George Goussard on numpy mailing list. - ADS

2009-01-16 Added helper function LinearSegmentedColormap.from_list to

facilitate building simple custom colomaps. See examples/pylab_examples/custom_cmap_fromlist.py - JDH

2009-01-16 Applied Michiel's patch for macosx backend to fix rounding

bug. Closed sf bug 2508440 - JSW

2009-01-10 Applied Michiel's hatch patch for macosx backend and

draw_idle patch for qt. Closes sf patched 2497785 and 2468809 - JDH

2009-01-10 Fix bug in pan/zoom with log coordinates. - EF

2009-01-06 Fix bug in setting of dashed negative contours. - EF

2009-01-06 Be fault tolerant when len(linestyles)>NLev in contour. - MM

2009-01-06 Added marginals kwarg to hexbin to plot marginal densities

JDH

2009-01-06 Change user-visible multipage pdf object to PdfPages to

avoid accidents with the file-like PdfFile. - JKS

2009-01-05 Fix a bug in pdf usetex: allow using non-embedded fonts. - JKS

2009-01-05 optional use of preview.sty in usetex mode. - JJL

2009-01-02 Allow multipage pdf files. - JKS

2008-12-31 Improve pdf usetex by adding support for font effects

(slanting and extending). - JKS

2008-12-29 Fix a bug in pdf usetex support, which occurred if the same

Type-1 font was used with different encodings, e.g., with Minion Pro and Mn-Symbol. - JKS

2008-12-20 fix the dpi-dependent offset of Shadow. - JJL

2008-12-20 fix the hatch bug in the pdf backend. minor update

in docs and example - JJL

2008-12-19 Add axes_locator attribute in Axes. Two examples are added.

- JJL

2008-12-19 Update Axes.legend documentation. /api/api_changes.rst is also

updated to describe changes in keyword parameters. Issue a warning if old keyword parameters are used. - JJL

2008-12-18 add new arrow style, a line + filled triangles. -JJL

2008-12-18 Re-Released 0.98.5.2 from v0_98_5_maint at r6679

Released 0.98.5.2 from v0_98_5_maint at r6667

2008-12-18 Removed configobj, experimental traits and doc/mpl_data link - JDH

2008-12-18 Fix bug where a line with NULL data limits prevents

subsequent data limits from calculating correctly - MGD

2008-12-17 Major documentation generator changes - MGD

2008-12-17 Applied macosx backend patch with support for path

collections, quadmesh, etc... - JDH

2008-12-17 fix dpi-dependent behavior of text bbox and arrow in annotate

-JJL

2008-12-17 Add group id support in artist. Two examples which

demonstrate svg filter are added. -JJL

2008-12-16 Another attempt to fix dpi-dependent behavior of Legend. -JJL

2008-12-16 Fixed dpi-dependent behavior of Legend and fancybox in Text.

2008-12-16 Added markevery property to Line2D to support subsampling

of markers - JDH

2008-12-15 Removed mpl_data symlink in docs. On platforms that do not

support symlinks, these become copies, and the font files are large, so the distro becomes unnecessarily bloated. Keeping the mpl_examples dir because relative links are harder for the plot directive and the *.py files are not so large. - JDH

2008-12-15 Fix $ in non-math text with usetex off. Document

differences between usetex on/off - MGD

2008-12-15 Fix anti-aliasing when auto-snapping - MGD

2008-12-15 Fix grid lines not moving correctly during pan and zoom - MGD

2008-12-12 Preparations to eliminate maskedarray rcParams key: its

use will now generate a warning. Similarly, importing the obsolote numerix.npyma will generate a warning. - EF

2008-12-12 Added support for the numpy.histogram() weights parameter

to the axes hist() method. Docs taken from numpy - MM

2008-12-12 Fixed warning in hist() with numpy 1.2 - MM

2008-12-12 Removed external packages: configobj and enthought.traits

which are only required by the experimental traited config and are somewhat out of date. If needed, install them independently, see:

http://code.enthought.com/pages/traits.html

and:

http://www.voidspace.org.uk/python/configobj.html

2008-12-12 Added support to assign labels to histograms of multiple

data. - MM

2008-12-11 Released 0.98.5 at svn r6573

2008-12-11 Use subprocess.Popen instead of os.popen in dviread

(Windows problem reported by Jorgen Stenarson) - JKS

2008-12-10 Added Michael's font_manager fix and Jae-Joon's

figure/subplot fix. Bumped version number to 0.98.5 - JDH

2008-12-09 Released 0.98.4 at svn r6536

2008-12-08 Added mdehoon's native macosx backend from sf patch 2179017 - JDH

2008-12-08 Removed the prints in the set_*style commands. Return the

list of pprinted strings instead - JDH

2008-12-08 Some of the changes Michael made to improve the output of

the property tables in the rest docs broke of made difficult to use some of the interactive doc helpers, e.g., setp and getp. Having all the rest markup in the ipython shell also confused the docstrings. I added a new rc param docstring.hardcopy, to format the docstrings differently for hard copy and other use.

The ArtistInspector could use a little refactoring now since there is duplication of effort between the rest out put and the non-rest output - JDH

2008-12-08 Updated spectral methods (psd, csd, etc.) to scale one-sided

densities by a factor of 2 and, optionally, scale all densities by the sampling frequency. This gives better MatLab compatibility. -RM

2008-12-08 Fixed alignment of ticks in colorbars. -MGD

2008-12-07 drop the deprecated "new" keyword of np.histogram() for

numpy 1.2 or later. -JJL

2008-12-06 Fixed a bug in svg backend that new_figure_manager()

ignores keywords arguments such as figsize, etc. -JJL

2008-12-05 Fixed a bug that the handlelength of the new legend class

set too short when numpoints=1 -JJL

2008-12-04 Added support for data with units (e.g., dates) to

Axes.fill_between. -RM

2008-12-04 Added fancybox keyword to legend. Also applied some changes

for better look, including baseline adjustment of the multiline texts so that it is center aligned. -JJL

2008-12-02 The transmuter classes in the patches.py are reorganized as

subclasses of the Style classes. A few more box and arrow styles are added. -JJL

2008-12-02 Fixed a bug in the new legend class that didn't allowed

a tuple of coordinate values as loc. -JJL

2008-12-02 Improve checks for external dependencies, using subprocess

(instead of deprecated popen*) and distutils (for version checking) - DSD

2008-11-30 Reimplementation of the legend which supports baseline alignment,

multi-column, and expand mode. - JJL

2008-12-01 Fixed histogram autoscaling bug when bins or range are given

explicitly (fixes Debian bug 503148) - MM

2008-11-25 Added rcParam axes.unicode_minus which allows plain hyphen

for minus when False - JDH

2008-11-25 Added scatterpoints support in Legend. patch by Erik

Tollerud - JJL

2008-11-24 Fix crash in log ticking. - MGD

2008-11-20 Added static helper method BrokenHBarCollection.span_where

and Axes/pyplot method fill_between. See examples/pylab/fill_between.py - JDH

2008-11-12 Add x_isdata and y_isdata attributes to Artist instances,

and use them to determine whether either or both coordinates are used when updating dataLim. This is used to fix autoscaling problems that had been triggered by axhline, axhspan, axvline, axvspan. - EF

2008-11-11 Update the psd(), csd(), cohere(), and specgram() methods

of Axes and the csd() cohere(), and specgram() functions in mlab to be in sync with the changes to psd(). In fact, under the hood, these all call the same core to do computations. - RM

2008-11-11 Add 'pad_to' and 'sides' parameters to mlab.psd() to

allow controlling of zero padding and returning of negative frequency components, respecitively. These are added in a way that does not change the API. - RM

2008-11-10 Fix handling of c kwarg by scatter; generalize

is_string_like to accept numpy and numpy.ma string array scalars. - RM and EF

2008-11-09 Fix a possible EINTR problem in dviread, which might help

when saving pdf files from the qt backend. - JKS

2008-11-05 Fix bug with zoom to rectangle and twin axes - MGD

2008-10-24 Added Jae Joon's fancy arrow, box and annotation

enhancements -- see examples/pylab_examples/annotation_demo2.py

2008-10-23 Autoscaling is now supported with shared axes - EF

2008-10-23 Fixed exception in dviread that happened with Minion - JKS

2008-10-21 set_xlim, ylim now return a copy of the viewlim array to

avoid modify inplace surprises

2008-10-20 Added image thumbnail generating function

matplotlib.image.thumbnail. See examples/misc/image_thumbnail.py - JDH

2008-10-20 Applied scatleg patch based on ideas and work by Erik

Tollerud and Jae-Joon Lee. - MM

2008-10-11 Fixed bug in pdf backend: if you pass a file object for

output instead of a filename, e.g., in a wep app, we now flush the object at the end. - JKS

2008-10-08 Add path simplification support to paths with gaps. - EF

2008-10-05 Fix problem with AFM files that don't specify the font's

full name or family name. - JKS

2008-10-04 Added 'scilimits' kwarg to Axes.ticklabel_format() method,

for easy access to the set_powerlimits method of the major ScalarFormatter. - EF

2008-10-04 Experimental new kwarg borderpad to replace pad in legend,

based on suggestion by Jae-Joon Lee. - EF

2008-09-27 Allow spy to ignore zero values in sparse arrays, based

on patch by Tony Yu. Also fixed plot to handle empty data arrays, and fixed handling of markers in figlegend. - EF

2008-09-24 Introduce drawstyles for lines. Transparently split linestyles

like 'steps--' into drawstyle 'steps' and linestyle '--'. Legends always use drawstyle 'default'. - MM

2008-09-18 Fixed quiver and quiverkey bugs (failure to scale properly

when resizing) and added additional methods for determining the arrow angles - EF

2008-09-18 Fix polar interpolation to handle negative values of theta - MGD

2008-09-14 Reorganized cbook and mlab methods related to numerical

calculations that have little to do with the goals of those two modules into a separate module numerical_methods.py Also, added ability to select points and stop point selection with keyboard in ginput and manual contour labeling code. Finally, fixed contour labeling bug. - DMK

2008-09-11 Fix backtick in Postscript output. - MGD

2008-09-10 [2089958] Path simplification for vector output backends

Leverage the simplification code exposed through path_to_polygons to simplify certain well-behaved paths in the vector backends (PDF, PS and SVG). "path.simplify" must be set to True in matplotlibrc for this to work. - MGD

2008-09-10 Add "filled" kwarg to Path.intersects_path and

Path.intersects_bbox. - MGD

2008-09-07 Changed full arrows slightly to avoid an xpdf rendering

problem reported by Friedrich Hagedorn. - JKS

2008-09-07 Fix conversion of quadratic to cubic Bezier curves in PDF

and PS backends. Patch by Jae-Joon Lee. - JKS

2008-09-06 Added 5-point star marker to plot command - EF

2008-09-05 Fix hatching in PS backend - MGD

2008-09-03 Fix log with base 2 - MGD

2008-09-01 Added support for bilinear interpolation in

NonUniformImage; patch by Gregory Lielens. - EF

2008-08-28 Added support for multiple histograms with data of

different length - MM

2008-08-28 Fix step plots with log scale - MGD

2008-08-28 Fix masked arrays with markers in non-Agg backends - MGD

2008-08-28 Fix clip_on kwarg so it actually works correctly - MGD

2008-08-25 Fix locale problems in SVG backend - MGD

2008-08-22 fix quiver so masked values are not plotted - JSW

2008-08-18 improve interactive pan/zoom in qt4 backend on windows - DSD

2008-08-11 Fix more bugs in NaN/inf handling. In particular, path simplification

(which does not handle NaNs or infs) will be turned off automatically when infs or NaNs are present. Also masked arrays are now converted to arrays with NaNs for consistent handling of masks and NaNs - MGD and EF

2008-08-03 Released 0.98.3 at svn r5947

2008-08-01 Backported memory leak fixes in _ttconv.cpp - MGD

2008-07-31 Added masked array support to griddata. - JSW

2008-07-26 Added optional C and reduce_C_function arguments to

axes.hexbin(). This allows hexbin to accumulate the values of C based on the x,y coordinates and display in hexagonal bins. - ADS

2008-07-24 Deprecated (raise NotImplementedError) all the mlab2

functions from matplotlib.mlab out of concern that some of them were not clean room implementations. JDH

2008-07-24 Rewrite of a significant portion of the clabel code (class

ContourLabeler) to improve inlining. - DMK

2008-07-22 Added Barbs polygon collection (similar to Quiver) for plotting

wind barbs. Added corresponding helpers to Axes and pyplot as well. (examples/pylab_examples/barb_demo.py shows it off.) - RMM

2008-07-21 Added scikits.delaunay as matplotlib.delaunay. Added griddata

function in matplotlib.mlab, with example (griddata_demo.py) in pylab_examples. griddata function will use mpl_toolkits._natgrid if installed. - JSW

2008-07-21 Re-introduced offset_copy that works in the context of the

new transforms. - MGD

2008-07-21 Committed patch by Ryan May to add get_offsets and

set_offsets to Collections base class - EF

2008-07-21 Changed the "asarray" strategy in image.py so that

colormapping of masked input should work for all image types (thanks Klaus Zimmerman) - EF

2008-07-20 Rewrote cbook.delete_masked_points and corresponding

unit test to support rgb color array inputs, datetime inputs, etc. - EF

2008-07-20 Renamed unit/axes_unit.py to cbook_unit.py and modified

in accord with Ryan's move of delete_masked_points from axes to cbook. - EF

2008-07-18 Check for nan and inf in axes.delete_masked_points().

This should help hexbin and scatter deal with nans. - ADS

2008-07-17 Added ability to manually select contour label locations.

Also added a waitforbuttonpress function. - DMK

2008-07-17 Fix bug with NaNs at end of path (thanks, Andrew Straw for

the report) - MGD

2008-07-16 Improve error handling in texmanager, thanks to Ian Henry

for reporting - DSD

2008-07-12 Added support for external backends with the

"module://my_backend" syntax - JDH

2008-07-11 Fix memory leak related to shared axes. Grouper should

store weak references. - MGD

2008-07-10 Bugfix: crash displaying fontconfig pattern - MGD

2008-07-10 Bugfix: [2013963] update_datalim_bounds in Axes not works - MGD

2008-07-10 Bugfix: [2014183] multiple imshow() causes gray edges - MGD

2008-07-09 Fix rectangular axes patch on polar plots bug - MGD

2008-07-09 Improve mathtext radical rendering - MGD

2008-07-08 Improve mathtext superscript placement - MGD

2008-07-07 Fix custom scales in pcolormesh (thanks Matthew Turk) - MGD

2008-07-03 Implemented findobj method for artist and pyplot - see

examples/pylab_examples/findobj_demo.py - JDH

2008-06-30 Another attempt to fix TextWithDash - DSD

2008-06-30 Removed Qt4 NavigationToolbar2.destroy -- it appears to

have been unnecessary and caused a bug reported by P. Raybaut - DSD

2008-06-27 Fixed tick positioning bug - MM

2008-06-27 Fix dashed text bug where text was at the wrong end of the

dash - MGD

2008-06-26 Fix mathtext bug for expressions like $x_{leftarrow}$ - MGD

2008-06-26 Fix direction of horizontal/vertical hatches - MGD

2008-06-25 Figure.figurePatch renamed Figure.patch, Axes.axesPatch

renamed Axes.patch, Axes.axesFrame renamed Axes.frame, Axes.get_frame, which returns Axes.patch, is deprecated. Examples and users guide updated - JDH

2008-06-25 Fix rendering quality of pcolor - MGD

2008-06-24 Released 0.98.2 at svn r5667 - (source only for debian) JDH

2008-06-24 Added "transparent" kwarg to savefig. - MGD

2008-06-24 Applied Stefan's patch to draw a single centered marker over

a line with numpoints==1 - JDH

2008-06-23 Use splines to render circles in scatter plots - MGD

2008-06-22 Released 0.98.1 at revision 5637

2008-06-22 Removed axes3d support and replaced it with a

NotImplementedError for one release cycle

2008-06-21 fix marker placement bug in backend_ps - DSD

2008-06-20 [1978629] scale documentation missing/incorrect for log - MGD

2008-06-20 Added closed kwarg to PolyCollection. Fixes bug [1994535

] still missing lines on graph with svn (r 5548). - MGD

2008-06-20 Added set/get_closed method to Polygon; fixes error

in hist - MM

2008-06-19 Use relative font sizes (e.g., 'medium' and 'large') in

rcsetup.py and matplotlibrc.template so that text will be scaled by default when changing rcParams['font.size'] - EF

2008-06-17 Add a generic PatchCollection class that can contain any

kind of patch. - MGD

2008-06-13 Change pie chart label alignment to avoid having labels

overwrite the pie - MGD

2008-06-12 Added some helper functions to the mathtext parser to

return bitmap arrays or write pngs to make it easier to use mathtext outside the context of an mpl figure. modified the mathpng sphinxext to use the mathtext png save functionality - see examples/api/mathtext_asarray.py - JDH

2008-06-11 Use matplotlib.mathtext to render math expressions in

online docs - MGD

2008-06-11 Move PNG loading/saving to its own extension module, and

remove duplicate code in _backend_agg.cpp and _image.cpp that does the same thing - MGD

2008-06-11 Numerous mathtext bugfixes, primarily related to

dpi-independence - MGD

2008-06-10 Bar now applies the label only to the first patch only, and

sets '_nolegend_' for the other patch labels. This lets autolegend work as expected for hist and bar - see https://sourceforge.net/tracker/index.php?func=detail&aid=1986597&group_id=80706&ati
JDH

2008-06-10 Fix text baseline alignment bug. [1985420] Repair of

baseline alignment in Text._get_layout. Thanks Stan West - MGD

2008-06-09 Committed Gregor's image resample patch to downsampling

images with new rcparam image.resample - JDH

2008-06-09 Don't install Enthought.Traits along with matplotlib. For

matplotlib developers convenience, it can still be installed by setting an option in setup.cfg while we figure decide if there is a future for the traited config - DSD

2008-06-09 Added range keyword arg to hist() - MM

2008-06-07 Moved list of backends to rcsetup.py; made use of lower

case for backend names consistent; use validate_backend when importing backends subpackage - EF

2008-06-06 hist() revision, applied ideas proposed by Erik Tollerud and

Olle Engdegard: make histtype='step' unfilled by default and introduce histtype='stepfilled'; use default color cycle; introduce reverse cumulative histogram; new align keyword - MM

2008-06-06 Fix closed polygon patch and also provide the option to

not close the polygon - MGD

2008-06-05 Fix some dpi-changing-related problems with PolyCollection,

as called by Axes.scatter() - MGD

2008-06-05 Fix image drawing so there is no extra space to the right

or bottom - MGD

2006-06-04 Added a figure title command suptitle as a Figure method

and pyplot command -- see examples/figure_title.py - JDH

2008-06-02 Added support for log to hist with histtype='step' and fixed

a bug for log-scale stacked histograms - MM

2008-05-29 Released 0.98.0 at revision 5314

2008-05-29 matplotlib.image.imread now no longer always returns RGBA

-- if the image is luminance or RGB, it will return a MxN or MxNx3 array if possible. Also uint8 is no longer always forced to float.

2008-05-29 Implement path clipping in PS backend - JDH

2008-05-29 Fixed two bugs in texmanager.py:

improved comparison of dvipng versions fixed a bug introduced when get_grey method was added - DSD

2008-05-28 Fix crashing of PDFs in xpdf and ghostscript when two-byte

characters are used with Type 3 fonts - MGD

2008-05-28 Allow keyword args to configure widget properties as

requested in http://sourceforge.net/tracker/index.php?func=detail&aid=1866207&group_id=
- JDH

2008-05-28 Replaced '-' with u'u2212' for minus sign as requested in

http://sourceforge.net/tracker/index.php?func=detail&aid=1962574&group_id=80706&atid

2008-05-28 zero width/height Rectangles no longer influence the

autoscaler. Useful for log histograms with empty bins - JDH

2008-05-28 Fix rendering of composite glyphs in Type 3 conversion

(particularly as evidenced in the Eunjin.ttf Korean font) Thanks Jae-Joon Lee for finding this!

2008-05-27 Rewrote the cm.ScalarMappable callback infrastructure to

use cbook.CallbackRegistry rather than custom callback handling. Amy users of add_observer/notify of the cm.ScalarMappable should uae the cm.ScalarMappable.callbacksSM CallbackRegistry instead. JDH

2008-05-27 Fix TkAgg build on Ubuntu 8.04 (and hopefully a more

general solution for other platforms, too.)

2008-05-24 Added PIL support for loading images to imread (if PIL is

available) - JDH

2008-05-23 Provided a function and a method for controlling the

plot color cycle. - EF

2008-05-23 Major revision of hist(). Can handle 2D arrays and create

stacked histogram plots; keyword 'width' deprecated and rwidth (relative width) introduced; align='edge' changed to center of bin - MM

2008-05-22 Added support for ReST-based doumentation using Sphinx.

Documents are located in doc/, and are broken up into a users guide and an API reference. To build, run the make.py files. Sphinx-0.4 is needed to build generate xml, which will be useful for rendering equations with mathml, use sphinx from svn until 0.4 is released - DSD

2008-05-21 Fix segfault in TkAgg backend - MGD

2008-05-21 Fix a "local variable unreferenced" bug in plotfile - MM

2008-05-19 Fix crash when Windows can not access the registry to

determine font path [Bug 1966974, thanks Patrik Simons] - MGD

2008-05-16 removed some unneeded code w/ the python 2.4 requirement.

cbook no longer provides compatibility for reversed, enumerate, set or izip. removed lib/subprocess, mpl1, sandbox/units, and the swig code. This stuff should remain on the maintenance branch for archival purposes. JDH

2008-05-16 Reorganized examples dir - JDH

2008-05-16 Added 'elinewidth' keyword arg to errorbar, based on patch

by Christopher Brown - MM

2008-05-16 Added 'cumulative' keyword arg to hist to plot cumulative

histograms. For normed hists, this is normalized to one - MM

2008-05-15 Fix Tk backend segfault on some machines - MGD

2008-05-14 Don't use stat on Windows (fixes font embedding problem) - MGD

2008-05-09 Fix /singlequote (') in Postscript backend - MGD

2008-05-08 Fix kerning in SVG when embedding character outlines - MGD

2008-05-07 Switched to future numpy histogram semantic in hist - MM

2008-05-06 Fix strange colors when blitting in QtAgg and Qt4Agg - MGD

2008-05-05 pass notify_axes_change to the figure's add_axobserver

in the qt backends, like we do for the other backends. Thanks Glenn Jones for the report - DSD

2008-05-02 Added step histograms, based on patch by Erik Tollerud. - MM

2008-05-02 On PyQt <= 3.14 there is no way to determine the underlying

Qt version. [1851364] - MGD

2008-05-02 Don't call sys.exit() when pyemf is not found [1924199] -

MGD

2008-05-02 Update _subprocess.c from upstream Python 2.5.2 to get a

few memory and reference-counting-related bugfixes. See bug 1949978. - MGD

2008-04-30 Added some record array editing widgets for gtk -- see

examples/rec_edit*.py - JDH

2008-04-29 Fix bug in mlab.sqrtm - MM

2008-04-28 Fix bug in SVG text with Mozilla-based viewers (the symbol

tag is not supported) - MGD

2008-04-27 Applied patch by Michiel de Hoon to add hexbin

axes method and pyplot function - EF

2008-04-25 Enforce python >= 2.4; remove subprocess build - EF

2008-04-25 Enforce the numpy requirement at build time - JDH

2008-04-24 Make numpy 1.1 and python 2.3 required when importing

matplotlib - EF

2008-04-24 Fix compilation issues on VS2003 (Thanks Martin Spacek for

all the help) - MGD

2008-04-24 Fix sub/superscripts when the size of the font has been

changed - MGD

2008-04-22 Use "svg.embed_char_paths" consistently everywhere - MGD

2008-04-20 Add support to MaxNLocator for symmetric axis autoscaling. - EF

2008-04-20 Fix double-zoom bug. - MM

2008-04-15 Speed up color mapping. - EF

2008-04-12 Speed up zooming and panning of dense images. - EF

2008-04-11 Fix global font rcParam setting after initialization

time. - MGD

2008-04-11 Revert commits 5002 and 5031, which were intended to

avoid an unnecessary call to draw(). 5002 broke saving figures before show(). 5031 fixed the problem created in 5002, but broke interactive plotting. Unnecessary call to draw still needs resolution - DSD

2008-04-07 Improve color validation in rc handling, suggested

by Lev Givon - EF

2008-04-02 Allow to use both linestyle definition arguments, '-' and

'solid' etc. in plots/collections - MM

2008-03-27 Fix saving to Unicode filenames with Agg backend

(other backends appear to already work...) (Thanks, Christopher Barker) - MGD

2008-03-26 Fix SVG backend bug that prevents copying and pasting in

Inkscape (thanks Kaushik Ghose) - MGD

2008-03-24 Removed an unnecessary call to draw() in the backend_qt*

mouseReleaseEvent. Thanks to Ted Drain - DSD

2008-03-23 Fix a pdf backend bug which sometimes caused the outermost

gsave to not be balanced with a grestore. - JKS

2008-03-20 Fixed a minor bug in ContourSet._process_linestyles when

len(linestyles)==Nlev - MM

2008-03-19 Changed ma import statements to "from numpy import ma";

this should work with past and future versions of numpy, whereas "import numpy.ma as ma" will work only with numpy >= 1.05, and "import numerix.npyma as ma" is obsolete now that maskedarray is replacing the earlier implementation, as of numpy 1.05.

2008-03-14 Removed an apparently unnecessary call to

FigureCanvasAgg.draw in backend_qt*agg. Thanks to Ted Drain - DSD

2008-03-10 Workaround a bug in backend_qt4agg's blitting due to a

buffer width/bbox width mismatch in _backend_agg's copy_from_bbox - DSD

2008-02-29 Fix class Wx toolbar pan and zoom functions (Thanks Jeff

Peery) - MGD

2008-02-16 Added some new rec array functionality to mlab

(rec_summarize, rec2txt and rec_groupby). See examples/rec_groupby_demo.py. Thanks to Tim M for rec2txt.

2008-02-12 Applied Erik Tollerud's span selector patch - JDH

2008-02-11 Update plotting() doc string to refer to getp/setp. - JKS

2008-02-10 Fixed a problem with square roots in the pdf backend with usetex. - JKS

2008-02-08 Fixed minor __str__ bugs so getp(gca()) works. - JKS

2008-02-05 Added getters for title, xlabel, ylabel, as requested by Brandon Kieth - EF

2008-02-05 Applied Gael's ginput patch and created examples/ginput_demo.py - JDH

2008-02-03 Expose interpnames, a list of valid interpolation methods, as an AxesImage class attribute. - EF

2008-02-03 Added BoundaryNorm, with examples in colorbar_only.py and image_masked.py. - EF

2008-02-03 Force dpi=72 in pdf backend to fix picture size bug. - JKS

2008-02-01 Fix doubly-included font problem in Postscript backend - MGD

2008-02-01 Fix reference leak in ft2font Glyph objects. - MGD

2008-01-31 Don't use unicode strings with usetex by default - DSD

2008-01-31 Fix text spacing problems in PDF backend with *some* fonts, such as STIXGeneral.

2008-01-31 Fix sqrt with radical number (broken by making [and] work below) - MGD

2008-01-27 Applied Martin Teichmann's patch to improve the Qt4 backend. Uses Qt's builtin toolbars and statusbars. See bug 1828848 - DSD

2008-01-10 Moved toolkits to mpl_toolkits, made mpl_toolkits a namespace package - JSWHIT

2008-01-10 Use setup.cfg to set the default parameters (tkagg, numpy) when building windows installers - DSD

2008-01-10 Fix bug displaying [and] in mathtext - MGD

2008-01-10 Fix bug when displaying a tick value offset with scientific notation. (Manifests itself as a warning that the times symbol can not be found). - MGD

2008-01-10 Use setup.cfg to set the default parameters (tkagg,

numpy) when building windows installers - DSD

2008-01-06 Released 0.91.2 at revision 4802

2007-12-26 Reduce too-late use of matplotlib.use() to a warning

instead of an exception, for backwards compatibility - EF

2007-12-25 Fix bug in errorbar, identified by Noriko Minakawa - EF

2007-12-25 Changed masked array importing to work with the upcoming

numpy 1.05 (now the maskedarray branch) as well as with earlier versions. - EF

2007-12-16 rec2csv saves doubles without losing precision. Also, it

does not close filehandles passed in open. - JDH,ADS

2007-12-13 Moved rec2gtk to matplotlib.toolkits.gtktools and rec2excel

to matplotlib.toolkits.exceltools - JDH

2007-12-12 Support alpha-blended text in the Agg and Svg backends -

MGD

2007-12-10 Fix SVG text rendering bug. - MGD

2007-12-10 Increase accuracy of circle and ellipse drawing by using an

8-piece bezier approximation, rather than a 4-piece one. Fix PDF, SVG and Cairo backends so they can draw paths (meaning ellipses as well). - MGD

2007-12-07 Issue a warning when drawing an image on a non-linear axis. - MGD

2007-12-06 let widgets.Cursor initialize to the lower x and y bounds

rather than 0,0, which can cause havoc for dates and other transforms - DSD

2007-12-06 updated references to mpl data directories for py2exe - DSD

2007-12-06 fixed a bug in rcsetup, see bug 1845057 - DSD

2007-12-05 Fix how fonts are cached to avoid loading the same one multiple times.

(This was a regression since 0.90 caused by the refactoring of font_manager.py) - MGD

2007-12-05 Support arbitrary rotation of usetex text in Agg backend. - MGD

2007-12-04 Support '|' as a character in mathtext - MGD

2007-11-27 Released 0.91.1 at revision 4517

2007-11-27 Released 0.91.0 at revision 4478

2007-11-13 All backends now support writing to a file-like object, not

just a regular file. savefig() can be passed a file-like object in place of a file path. - MGD

2007-11-13 Improved the default backend selection at build time:

SVG -> Agg -> TkAgg -> WXAgg -> GTK -> GTKAgg. The last usable backend in this progression will be chosen in the default config file. If a backend is defined in setup.cfg, that will be the default backend - DSD

2007-11-13 Improved creation of default config files at build time for

traited config package - DSD

2007-11-12 Exposed all the build options in setup.cfg. These options are

read into a dict called "options" by setupext.py. Also, added "-mpl" tags to the version strings for packages provided by matplotlib. Versions provided by mpl will be identified and updated on subsequent installs - DSD

2007-11-12 Added support for STIX fonts. A new rcParam,

mathtext.fontset, can be used to choose between:

'cm':

The TeX/LaTeX Computer Modern fonts

'stix':

The STIX fonts (see stixfonts.org)

'stixsans':

The STIX fonts, using sans-serif glyphs by default

'custom':

A generic Unicode font, in which case the mathtext font must be specified using mathtext.bf, mathtext.it, mathtext.sf etc.

Added a new example, stix_fonts_demo.py to show how to access different fonts and unusual symbols.

- MGD

2007-11-12 Options to disable building backend extension modules moved

from setup.py to setup.cfg - DSD

2007-11-09 Applied Martin Teichmann's patch 1828813: a QPainter is used in

paintEvent, which has to be destroyed using the method end(). If matplotlib raises an exception before the call to end - and it does if you feed it with bad data - this method end() is never called and Qt4 will start spitting error messages

2007-11-09 Moved pyparsing back into matplotlib namespace. Don't use

system pyparsing, API is too variable from one release to the next - DSD

2007-11-08 Made pylab use straight numpy instead of oldnumeric

by default - EF

2007-11-08 Added additional record array utilities to mlab (rec2excel,

rec2gtk, rec_join, rec_append_field, rec_drop_field) - JDH

2007-11-08 Updated pytz to version 2007g - DSD

2007-11-08 Updated pyparsing to version 1.4.8 - DSD

2007-11-08 Moved csv2rec to recutils and added other record array

utilities - JDH

2007-11-08 If available, use existing pyparsing installation - DSD

2007-11-07 Removed old enthought.traits from lib/matplotlib, added

Gael Varoquaux's enthought.traits-2.6b1, which is stripped of setuptools. The package is installed to site-packages if not already available - DSD

2007-11-05 Added easy access to minor tick properties; slight mod

of patch by Pierre G-M - EF

2007-11-02 Committed Phil Thompson's patch 1599876, fixes to Qt4Agg

backend and qt4 blitting demo - DSD

2007-11-02 Committed Phil Thompson's patch 1599876, fixes to Qt4Agg

backend and qt4 blitting demo - DSD

2007-10-31 Made log color scale easier to use with contourf;

automatic level generation now works. - EF

2007-10-29 TRANSFORMS REFACTORING

The primary goal of this refactoring was to make it easier to extend matplotlib to support new kinds of projections. This is primarily an internal improvement, and the possible user-visible changes it allows are yet to come.

The transformation framework was completely rewritten in Python (with Numpy). This will make it easier to add news kinds of transformations without writing C/C++ code.

Transforms are composed into a 'transform tree', made of transforms whose value depends on other transforms (their children). When the contents of children change, their parents are automatically updated to reflect those changes. To do this an "invalidation" method is used: when children change, all of their ancestors are marked as "invalid". When the value of a transform is accessed at a later time, its value is recomputed only if it is invalid, otherwise a cached value may be used. This prevents unnecessary

recomputations of transforms, and contributes to better interactive performance.

The framework can be used for both affine and non-affine transformations. However, for speed, we want use the backend renderers to perform affine transformations whenever possible. Therefore, it is possible to perform just the affine or non-affine part of a transformation on a set of data. The affine is always assumed to occur after the non-affine. For any transform:

full transform == non-affine + affine

Much of the drawing has been refactored in terms of compound paths. Therefore, many methods have been removed from the backend interface and replaced with a a handful to draw compound paths. This will make updating the backends easier, since there is less to update. It also should make the backends more consistent in terms of functionality.

User visible changes:

- POLAR PLOTS: Polar plots are now interactively zoomable, and the r-axis labels can be interactively rotated. Straight line segments are now interpolated to follow the curve of the r-axis.
- Non-rectangular clipping works in more backends and with more types of objects.
- Sharing an axis across figures is now done in exactly the same way as sharing an axis between two axes in the same figure:

```
fig1 = figure()
fig2 = figure()

 ax1 = fig1.add_subplot(111)
 ax2 = fig2.add_subplot(111, sharex=ax1, sharey=ax1)
```

- linestyles now include steps-pre, steps-post and steps-mid. The old step still works and is equivalent to step-pre.
- Multiple line styles may be provided to a collection.

See API_CHANGES for more low-level information about this refactoring.

2007-10-24 Added ax kwarg to Figure.colorbar and pyplot.colorbar - EF

2007-10-19 Removed a gsave/grestore pair surrounding _draw_ps, which

was causing a loss graphics state info (see "EPS output problem - scatter & edgecolors" on mpl-dev, 2007-10-29) - DSD

2007-10-15 Fixed a bug in patches.Ellipse that was broken for

aspect='auto'. Scale free ellipses now work properly for equal and auto on Agg and PS, and they fall back on a polygonal approximation for nonlinear transformations until we convince oursleves that the spline approximation holds for non-

linear transformations. Added unit/ellipse_compare.py to compare spline with vertex approx for both aspects. JDH

2007-10-05 remove generator expressions from texmanager and mpltraits.

generator expressions are not supported by python-2.3 - DSD

2007-10-01 Made matplotlib.use() raise an exception if called after

backends has been imported. - EF

2007-09-30 Modified update* methods of Bbox and Interval so they

work with reversed axes. Prior to this, trying to set the ticks on a reversed axis failed with an uninformative error message. - EF

2007-09-30 Applied patches to axes3d to fix index error problem - EF

2007-09-24 Applied Eike Welk's patch reported on mpl-dev on 2007-09-22

Fixes a bug with multiple plot windows in the qt backend, ported the changes to backend_qt4 as well - DSD

2007-09-21 Changed cbook.reversed to yield the same result as the

python reversed builtin - DSD

2007-09-13 The usetex support in the pdf backend is more usable now,

so I am enabling it. - JKS

2007-09-12 Fixed a Axes.bar unit bug - JDH

2007-09-10 Made skiprows=1 the default on csv2rec - JDH

2007-09-09 Split out the plotting part of pylab and put it in

pyplot.py; removed numerix from the remaining pylab.py, which imports everything from pyplot.py. The intention is that apart from cleanups, the result of importing from pylab is nearly unchanged, but there is the new alternative of importing from pyplot to get the state-engine graphics without all the numeric functions. Numpified examples; deleted two that were obsolete; modified some to use pyplot. - EF

2007-09-08 Eliminated gd and paint backends - EF

2007-09-06 .bmp file format is now longer an alias for .raw

2007-09-07 Added clip path support to pdf backend. - JKS

2007-09-06 Fixed a bug in the embedding of Type 1 fonts in PDF.

Now it doesn't crash Preview.app. - JKS

2007-09-06 Refactored image saving code so that all GUI backends can

save most image types. See FILETYPES for a matrix of backends and their supported file types. Backend canvases should no longer write their own print_figure() method -- instead they should write a print_xxx method for each

filetype they can output and add an entry to their class-scoped filetypes dictionary. - MGD

2007-09-05 Fixed Qt version reporting in setupext.py - DSD

2007-09-04 Embedding Type 1 fonts in PDF, and thus usetex support

via dviread, sort of works. To test, enable it by renaming _draw_tex to draw_tex. - JKS

2007-09-03 Added ability of errorbar show limits via caret or

arrowhead ends on the bars; patch by Manual Metz. - EF

2007-09-03 Created type1font.py, added features to AFM and FT2Font

(see API_CHANGES), started work on embedding Type 1 fonts in pdf files. - JKS

2007-09-02 Continued work on dviread.py. - JKS

2007-08-16 Added a set_extent method to AxesImage, allow data extent

to be modified after initial call to imshow - DSD

2007-08-14 Fixed a bug in pyqt4 subplots-adjust. Thanks to

Xavier Gnata for the report and suggested fix - DSD

2007-08-13 Use pickle to cache entire fontManager; change to using

font_manager module-level function findfont wrapper for the fontManager.findfont method - EF

2007-08-11 Numpification and cleanup of mlab.py and some examples - EF

2007-08-06 Removed mathtext2

2007-07-31 Refactoring of distutils scripts.

- Will not fail on the entire build if an optional Python package (e.g., Tkinter) is installed but its development headers are not (e.g., tk-devel). Instead, it will continue to build all other extensions.
- Provide an overview at the top of the output to display what dependencies and their versions were found, and (by extension) what will be built.
- Use pkg-config, when available, to find freetype2, since this was broken on Mac OS-X when using MacPorts in a non- standard location.

2007-07-30 Reorganized configuration code to work with traited config

objects. The new config system is located in the matplotlib.config package, but it is disabled by default. To enable it, set NEWCONFIG=True in matplotlib.__init__.py. The new configuration system will still use the old matplotlibrc files by default. To switch to the experimental, traited configuration, set USE_TRAITED_CONFIG=True in config.__init__.py.

2007-07-29 Changed default pcolor shading to flat; added aliases

to make collection kwargs agree with setter names, so updating works; related minor cleanups. Removed quiver_classic, scatter_classic, pcolor_classic. - EF

2007-07-26 Major rewrite of mathtext.py, using the TeX box layout model.

There is one (known) backward incompatible change. The font commands (cal, rm, it, tt) now behave as TeX does: they are in effect until the next font change command or the end of the grouping. Therefore uses of $cal{R}$ should be changed to ${cal R}$. Alternatively, you may use the new LaTeX-style font commands (mathcal, mathrm, mathit, mathtt) which do affect the following group, e.g., $mathcal{R}$.

Other new features include:

- Math may be interspersed with non-math text. Any text with an even number of $'s (non-escaped) will be sent to the mathtext parser for layout.
- Sub/superscripts are less likely to accidentally overlap.
- Support for sub/superscripts in either order, e.g., x^i_j and x_j^i are equivalent.
- Double sub/superscripts (e.g., x_i_j) are considered ambiguous and raise an exception. Use braces to disambiguate.
- $frac{x}{y}$ can be used for displaying fractions.
- $sqrt[3]{x}$ can be used to display the radical symbol with a root number and body.
- $left(frac{x}{y}right)$ may be used to create parentheses and other delimiters that automatically resize to the height of their contents.
- Spacing around operators etc. is now generally more like TeX.
- Added support (and fonts) for boldface (bf) and sans-serif (sf) symbols.
- Log-like function name shortcuts are supported. For example, $sin(x)$ may be used instead of ${rm sin}(x)$
- Limited use of kerning for the easy case (same font)

Behind the scenes, the pyparsing.py module used for doing the math parsing was updated to the latest stable version (1.4.6). A lot of duplicate code was refactored out of the Font classes.

- MGD

2007-07-19 completed numpification of most trivial cases - NN

2007-07-19 converted non-numpy relicts throughout the code - NN

2007-07-19 replaced the Python code in numerix/ by a minimal wrapper around

numpy that explicitly mentions all symbols that need to be addressed for further numpification - NN

2007-07-18 make usetex respect changes to rcParams. texmanager used to
only configure itself when it was created, now it reconfigures when rcParams are changed. Thank you Alexander Schmolck for contributing a patch - DSD

2007-07-17 added validation to setting and changing rcParams - DSD

2007-07-17 bugfix segfault in transforms module. Thanks Ben North for
the patch. - ADS

2007-07-16 clean up some code in ticker.ScalarFormatter, use unicode to
render multiplication sign in offset ticklabel - DSD

2007-07-16 fixed a formatting bug in ticker.ScalarFormatter's scientific
notation (10^0 was being rendered as 10 in some cases) - DSD

2007-07-13 Add MPL_isfinite64() and MPL_isinf64() for testing
doubles in (the now misnamed) MPL_isnan.h. - ADS

2007-07-13 The matplotlib._isnan module removed (use numpy.isnan) - ADS

2007-07-13 Some minor cleanups in _transforms.cpp - ADS

2007-07-13 Removed the rest of the numerix extension code detritus,
numpified axes.py, and cleaned up the imports in axes.py - JDH

2007-07-13 Added legend.loc as configurable option that could in
future default to 'best'. - NN

2007-07-12 Bugfixes in mlab.py to coerce inputs into numpy arrays. -ADS

2007-07-11 Added linespacing kwarg to text.Text - EF

2007-07-11 Added code to store font paths in SVG files. - MGD

2007-07-10 Store subset of TTF font as a Type 3 font in PDF files. - MGD

2007-07-09 Store subset of TTF font as a Type 3 font in PS files. - MGD

2007-07-09 Applied Paul's pick restructure pick and add pickers,
sourceforge patch 1749829 - JDH

2007-07-09 Applied Allan's draw_lines agg optimization. JDH

2007-07-08 Applied Carl Worth's patch to fix cairo draw_arc - SC

2007-07-07 fixed bug 1712099: xpdf distiller on windows - DSD

2007-06-30 Applied patches to tkagg, gtk, and wx backends to reduce
memory leakage. Patches supplied by Mike Droettboom; see tracker numbers 1745400, 1745406, 1745408. Also made unit/memleak_gui.py more flexible with command-line options. - EF

2007-06-30 Split defaultParams into separate file rcdefaults (together with

validation code). Some heavy refactoring was necessary to do so, but the overall behavior should be the same as before. - NN

2007-06-27 Added MPLCONFIGDIR for the default location for mpl data

and configuration. useful for some apache installs where HOME is not writable. Tried to clean up the logic in _get_config_dir to support non-writable HOME where are writable HOME/.matplotlib already exists - JDH

2007-06-27 Fixed locale bug reported at

http://sourceforge.net/tracker/index.php?func=detail&aid=1744154&group_id=80706&atid
by adding a cbook.unicode_safe function - JDH

2007-06-27 Applied Micheal's tk savefig bugfix described at

http://sourceforge.net/tracker/index.php?func=detail&aid=1716732&group_id=80706&atid
Thanks Michael!

2007-06-27 Patch for get_py2exe_datafiles() to work with new directory

layout. (Thanks Tocer and also Werner Bruhin.) -ADS

2007-06-27 Added a scroll event to the mpl event handling system and

implemented it for backends GTK* -- other backend users/developers/maintainers, please add support for your backend. - JDH

2007-06-25 Changed default to clip=False in colors.Normalize;

modified ColorbarBase for easier colormap display - EF

2007-06-13 Added maskedarray option to rc, numerix - EF

2007-06-11 Python 2.5 compatibility fix for mlab.py - EF

2007-06-10 In matplotlibrc file, use 'dashed' | 'solid' instead

of a pair of floats for contour.negative_linestyle - EF

2007-06-08 Allow plot and fill fmt string to be any mpl string

colorspec - EF

2007-06-08 Added gnuplot file plotfile function to pylab -- see

examples/plotfile_demo.py - JDH

2007-06-07 Disable build of numarray and Numeric extensions for

internal MPL use and the numerix layer. - ADS

2007-06-07 Added csv2rec to matplotlib.mlab to support automatically

converting csv files to record arrays using type introspection, and turned on native datetime support using the new units support in matplotlib.dates. See examples/loadrec.py ! JDH

2007-06-07 Simplified internal code of _auto_legend_data - NN

2007-06-04 Added labeldistance arg to Axes.pie to control the raidal

distance of the wedge labels - JDH

2007-06-03 Turned mathtext in SVG into single <text> with multiple <tspan>

objects (easier to edit in inkscape). - NN

2007-06-02 Released 0.90.1 at revision 3352

2007-06-02 Display only meaningful labels when calling legend()

without args. - NN

2007-06-02 Have errorbar follow the color cycle even if line is not plotted.

Suppress plotting of errorbar caps for capsize=0. - NN

2007-06-02 Set markers to same alpha value as line. - NN

2007-06-02 Fix mathtext position in svg backend. - NN

2007-06-01 Deprecate Numeric and numarray for use as numerix. Props to

Travis -- job well done. - ADS

2007-05-18 Added LaTeX unicode support. Enable with the

'text.latex.unicode' rcParam. This requires the ucs and inputenc LaTeX packages. - ADS

2007-04-23 Fixed some problems with polar -- added general polygon

clipping to clip the lines and grids to the polar axes. Added support for set_rmax to easily change the maximum radial grid. Added support for polar legend - JDH

2007-04-16 Added Figure.autofmt_xdate to handle adjusting the bottom

and rotating the tick labels for date plots when the ticks often overlap - JDH

2007-04-09 Beginnings of usetex support for pdf backend. -JKS

2007-04-07 Fixed legend/LineCollection bug. Added label support

to collections. - EF

2007-04-06 Removed deprecated support for a float value as a gray-scale;

now it must be a string, like '0.5'. Added alpha kwarg to ColorConverter.to_rgba_list. - EF

2007-04-06 Fixed rotation of ellipses in pdf backend

(sf bug #1690559) -JKS

2007-04-04 More matshow tweaks; documentation updates; new method

set_bounds() for formatters and locators. - EF

2007-04-02 Fixed problem with imshow and matshow of integer arrays;

fixed problems with changes to color autoscaling. - EF

2007-04-01 Made image color autoscaling work correctly with

a tracking colorbar; norm.autoscale now scales unconditionally, while norm.autoscale_None changes only None-valued vmin, vmax. - EF

2007-03-31 Added a qt-based subplot-adjustment dialog - DSD

2007-03-30 Fixed a bug in backend_qt4, reported on mpl-dev - DSD

2007-03-26 Removed colorbar_classic from figure.py; fixed bug in

Figure.clf() in which _axobservers was not getting cleared. Modernization and cleanups. - EF

2007-03-26 Refactored some of the units support -- units now live in

the respective x and y Axis instances. See also API_CHANGES for some alterations to the conversion interface. JDH

2007-03-25 Fix masked array handling in quiver.py for numpy. (Numeric

and numarray support for masked arrays is broken in other ways when using quiver. I didn't pursue that.) - ADS

2007-03-23 Made font_manager.py close opened files. - JKS

2007-03-22 Made imshow default extent match matshow - EF

2007-03-22 Some more niceties for xcorr -- a maxlags option, normed

now works for xcorr as well as axorr, usevlines is supported, and a zero correlation hline is added. See examples/xcorr_demo.py. Thanks Sameer for the patch. - JDH

2007-03-21 Axes.vlines and Axes.hlines now create and returns a

LineCollection, not a list of lines. This is much faster. The kwarg signature has changed, so consult the docs. Modified Axes.errorbar which uses vlines and hlines. See API_CHANGES; the return signature for these three functions is now different

2007-03-20 Refactored units support and added new examples - JDH

2007-03-19 Added Mike's units patch - JDH

2007-03-18 Matshow as an Axes method; test version matshow1() in

pylab; added 'integer' Boolean kwarg to MaxNLocator initializer to force ticks at integer locations. - EF

2007-03-17 Preliminary support for clipping to paths agg - JDH

2007-03-17 Text.set_text() accepts anything convertible with '%s' - EF

2007-03-14 Add masked-array support to hist. - EF

2007-03-03 Change barh to take a kwargs dict and pass it to bar.

Fixes sf bug #1669506.

2007-03-02 Add rc parameter pdf.inheritcolor, which disables all

color-setting operations in the pdf backend. The idea is that you include the resulting file in another program and set the colors (both stroke and fill color) there, so you can use the same pdf file for e.g., a paper and a presentation and have them in the surrounding color. You will probably not want to draw figure and axis frames in that case, since they would be filled in the same color. - JKS

2007-02-26 Prevent building _wxagg.so with broken Mac OS X wxPython. - ADS

2007-02-23 Require setuptools for Python 2.3 - ADS

2007-02-22 WXAgg accelerator updates - KM

WXAgg's C++ accelerator has been fixed to use the correct wxBitmap constructor.

The backend has been updated to use new wxPython functionality to provide fast blit() animation without the C++ accelerator. This requires wxPython 2.8 or later. Previous versions of wxPython can use the C++ acclerator or the old pure Python routines.

setup.py no longer builds the C++ accelerator when wxPython >= 2.8 is present.

The blit() method is now faster regardless of which agg/wxPython conversion routines are used.

2007-02-21 Applied the PDF backend patch by Nicolas Grilly.

This impacts several files and directories in matplotlib:

- Created the directory lib/matplotlib/mpl-data/fonts/pdfcorefonts, holding AFM files for the 14 PDF core fonts. These fonts are embedded in every PDF viewing application.
- setup.py: Added the directory pdfcorefonts to package_data.
- lib/matplotlib/__init__.py: Added the default parameter 'pdf.use14corefonts'. When True, the PDF backend uses only the 14 PDF core fonts.
- lib/matplotlib/afm.py: Added some keywords found in recent AFM files. Added a little workaround to handle Euro symbol.
- lib/matplotlib/fontmanager.py: Added support for the 14 PDF core fonts. These fonts have a dedicated cache (file pdfcorefont.cache), not the same as for other AFM files (file .afmfont.cache). Also cleaned comments to conform to CODING_GUIDE.
- lib/matplotlib/backends/backend_pdf.py: Added support for 14 PDF core fonts. Fixed some issues with incorrect character widths and encodings (works only for the most common encoding, WinAnsiEncoding, defined by

the official PDF Reference). Removed parameter 'dpi' because it causes alignment issues.

-JKS (patch by Nicolas Grilly)

2007-02-17 Changed ft2font.get_charmap, and updated all the files where

get_charmap is mentioned - ES

2007-02-13 Added barcode demo- JDH

2007-02-13 Added binary colormap to cm - JDH

2007-02-13 Added twiny to pylab - JDH

2007-02-12 Moved data files into lib/matplotlib so that setuptools'

develop mode works. Re-organized the mpl-data layout so that this source structure is maintained in the installation. (i.e., the 'fonts' and 'images' sub-directories are maintained in site-packages.) Suggest removing site-packages/matplotlib/mpl-data and ~/.matplotlib/ttffont.cache before installing - ADS

2007-02-07 Committed Rob Hetland's patch for qt4: remove

references to text()/latin1(), plus some improvements to the toolbar layout - DSD

2007-02-06 Released 0.90.0 at revision 3003

2007-01-22 Extended the new picker API to text, patches and patch

collections. Added support for user customizable pick hit testing and attribute tagging of the PickEvent - Details and examples in examples/pick_event_demo.py - JDH

2007-01-16 Begun work on a new pick API using the mpl event handling

frameowrk. Artists will define their own pick method with a configurable epsilon tolerance and return pick attrs. All artists that meet the tolerance threshold will fire a PickEvent with artist dependent attrs; e.g., a Line2D can set the indices attribute that shows the indices into the line that are within epsilon of the pick point. See examples/pick_event_demo.py. The implementation of pick for the remaining Artists remains to be done, but the core infrastructure at the level of event handling is in place with a proof-of-concept implementation for Line2D - JDH

2007-01-16 src/_image.cpp: update to use Py_ssize_t (for 64-bit systems).

Use return value of fread() to prevent warning messages - SC.

2007-01-15 src/_image.cpp: combine buffer_argb32() and buffer_bgra32() into

a new method color_conv(format) - SC

2007-01-14 backend_cairo.py: update draw_arc() so that

examples/arctest.py looks correct - SC

2007-01-12 backend_cairo.py: enable clipping. Update draw_image() so that

examples/contour_demo.py looks correct - SC

2007-01-12 backend_cairo.py: fix draw_image() so that examples/image_demo.py

now looks correct - SC

2007-01-11 Added Axes.xcorr and Axes.acorr to plot the cross

correlation of x vs. y or the autocorrelation of x. pylab wrappers also provided. See examples/xcorr_demo.py - JDH

2007-01-10 Added "Subplot.label_outer" method. It will set the

visibility of the ticklabels so that yticklabels are only visible in the first column and xticklabels are only visible in the last row - JDH

2007-01-02 Added additional kwarg documentation - JDH

2006-12-28 Improved error message for nonpositive input to log

transform; added log kwarg to bar, barh, and hist, and modified bar method to behave sensibly by default when the ordinate has a log scale. (This only works if the log scale is set before or by the call to bar, hence the utility of the log kwarg.) - EF

2006-12-27 backend_cairo.py: update draw_image() and _draw_mathtext() to work

with numpy - SC

2006-12-20 Fixed xpdf dependency check, which was failing on windows.

Removed ps2eps dependency check. - DSD

2006-12-19 Added Tim Leslie's spectral patch - JDH

2006-12-17 Added rc param 'axes.formatter.limits' to control

the default threshold for switching to scientific notation. Added convenience method Axes.ticklabel_format() for turning scientific notation on or off on either or both axes. - EF

2006-12-16 Added ability to turn control scientific notation

in ScalarFormatter - EF

2006-12-16 Enhanced boxplot to handle more flexible inputs - EF

2006-12-13 Replaced calls to where() in colors.py with much faster

clip() and putmask() calls; removed inappropriate uses of getmaskorNone (which should be needed only very rarely); all in response to profiling by David Cournapeau. Also fixed bugs in my 2-D array support from 12-09. - EF

2006-12-09 Replaced spy and spy2 with the new spy that combines

marker and image capabilities - EF

2006-12-09 Added support for plotting 2-D arrays with plot:

columns are plotted as in Matlab - EF

2006-12-09 Added linewidth kwarg to bar and barh; fixed arg

checking bugs - EF

2006-12-07 Made pcolormesh argument handling match pcolor;

fixed kwarg handling problem noted by Pierre GM - EF

2006-12-06 Made pcolor support vector X and/or Y instead of

requiring 2-D arrays - EF

2006-12-05 Made the default Artist._transform None (rather than

invoking identity_transform for each artist only to have it overridden later). Use artist.get_transform() rather than artist._transform, even in derived classes, so that the default transform will be created lazily as needed - JDH

2006-12-03 Added LogNorm to colors.py as illustrated by

examples/pcolor_log.py, based on suggestion by Jim McDonald. Colorbar modified to handle LogNorm. Norms have additional "inverse" method. - EF

2006-12-02 Changed class names in colors.py to match convention:

normalize -> Normalize, no_norm -> NoNorm. Old names are still available. Changed __init__.py rc defaults to match those in matplotlibrc - EF

2006-11-22 Fixed bug in set_*lim that I had introduced on 11-15 - EF

2006-11-22 Added examples/clippedline.py, which shows how to clip line

data based on view limits -- it also changes the marker style when zoomed in - JDH

2006-11-21 Some spy bug-fixes and added precision arg per Robert C's

suggestion - JDH

2006-11-19 Added semi-automatic docstring generation detailing all the

kwargs that functions take using the artist introspection tools; e.g., 'help text now details the scatter kwargs that control the Text properties - JDH

2006-11-17 Removed obsolete scatter_classic, leaving a stub to

raise NotImplementedError; same for pcolor_classic - EF

2006-11-15 Removed obsolete pcolor_classic - EF

2006-11-15 Fixed 1588908 reported by Russel Owen; factored

nonsingular method out of ticker.py, put it into transforms.py as a function, and used it in set_xlim and set_ylim. - EF

2006-11-14 Applied patch 1591716 by Ulf Larssen to fix a bug in

apply_aspect. Modified and applied patch 1594894 by mdehoon to fix bugs and improve formatting in lines.py. Applied patch 1573008 by Greg Willden to make psd etc. plot full frequency range for complex inputs. - EF

2006-11-14 Improved the ability of the colorbar to track

changes in corresponding image, pcolor, or contourf. - EF

2006-11-11 Fixed bug that broke Numeric compatibility;

added support for alpha to colorbar. The alpha information is taken from the mappable object, not specified as a kwarg. - EF

2006-11-05 Added broken_barh function for makring a sequence of

horizontal bars broken by gaps -- see examples/broken_barh.py

2006-11-05 Removed lineprops and markerprops from the Annotation code

and replaced them with an arrow configurable with kwarg arrowprops. See examples/annotation_demo.py - JDH

2006-11-02 Fixed a pylab subplot bug that was causing axes to be

deleted with hspace or wspace equals zero in subplots_adjust - JDH

2006-10-31 Applied axes3d patch 1587359

http://sourceforge.net/tracker/index.php?func=detail&aid=1587359&group_id=80706&atid
JDH

2006-10-26 Released 0.87.7 at revision 2835

2006-10-25 Made "tiny" kwarg in Locator.nonsingular much smaller - EF

2006-10-17 Closed sf bug 1562496 update line props dash/solid/cap/join

styles - JDH

2006-10-17 Complete overhaul of the annotations API and example code -

See matplotlib.text.Annotation and examples/annotation_demo.py JDH

2006-10-12 Committed Manuel Metz's StarPolygon code and

examples/scatter_star_poly.py - JDH

2006-10-11 commented out all default values in matplotlibrc.template

Default values should generally be taken from defaultParam in __init__.py - the file matplotlib should only contain those values that the user wants to explicitly

change from the default. (see thread "marker color handling" on matplotlib-devel)

2006-10-10 Changed default comment character for load to '#' - JDH

2006-10-10 deactivated rcfile-configurability of markerfacecolor

and markeredgecolor. Both are now hardcoded to the special value 'auto' to follow the line color. Configurability at run-time (using function arguments) remains functional. - NN

2006-10-07 introduced dummy argument magnification=1.0 to

FigImage.make_image to satisfy unit test figimage_demo.py The argument is not yet handled correctly, which should only show up when using non-standard DPI settings in PS backend, introduced by patch #1562394. - NN

2006-10-06 add backend-agnostic example: simple3d.py - NN

2006-09-29 fix line-breaking for SVG-inline images (purely cosmetic) - NN

2006-09-29 reworked set_linestyle and set_marker

markeredgecolor and markerfacecolor now default to a special value "auto" that keeps the color in sync with the line color further, the intelligence of axes.plot is cleaned up, improved and simplified. Complete compatibility cannot be guaranteed, but the new behavior should be much more predictable (see patch #1104615 for details) - NN

2006-09-29 changed implementation of clip-path in SVG to work around a

limitation in inkscape - NN

2006-09-29 added two options to matplotlibrc:

svg.image_inline svg.image_noscale see patch #1533010 for details - NN

2006-09-29 axes.py: cleaned up kwargs checking - NN

2006-09-29 setup.py: cleaned up setup logic - NN

2006-09-29 setup.py: check for required pygtk versions, fixes bug #1460783 - SC

2006-09-27 Released 0.87.6 at revision 2783

2006-09-24 Added line pointers to the Annotation code, and a pylab

interface. See matplotlib.text.Annotation, examples/annotation_demo.py and examples/annotation_demo_pylab.py - JDH

2006-09-18 mathtext2.py: The SVG backend now supports the same things that

the AGG backend does. Fixed some bugs with rendering, and out of bounds errors in the AGG backend - ES. Changed the return values of math_parse_s_ft2font_svg to support lines (fractions etc.)

2006-09-17 Added an Annotation class to facilitate annotating objects

and an examples file examples/annotation_demo.py. I want to add dash support as in TextWithDash, but haven't decided yet whether inheriting from TextWithDash is the right base class or if another approach is needed - JDH

2006-09-05 Released 0.87.5 at revision 2761

2006-09-04 Added nxutils for some numeric add-on extension code --

specifically a better/more efficient inside polygon tester (see unit/inside_poly_*.py) - JDH

2006-09-04 Made bitstream fonts the rc default - JDH

2006-08-31 Fixed alpha-handling bug in ColorConverter, affecting

collections in general and contour/contourf in particular. - EF

2006-08-30 ft2font.cpp: Added draw_rect_filled method (now used by mathtext2

to draw the fraction bar) to FT2Font - ES

2006-08-29 setupext.py: wrap calls to tk.getvar() with str(). On some

systems, getvar returns a Tcl_Obj instead of a string - DSD

2006-08-28 mathtext2.py: Sub/superscripts can now be complex (i.e.

fractions etc.). The demo is also updated - ES

2006-08-28 font_manager.py: Added /usr/local/share/fonts to list of

X11 font directories - DSD

2006-08-28 mahtext2.py: Initial support for complex fractions. Also,

rendering is now completely separated from parsing. The sub/superscripts now work better. Updated the mathtext2_demo.py - ES

2006-08-27 qt backends: don't create a QApplication when backend is

imported, do it when the FigureCanvasQt is created. Simplifies applications where mpl is embedded in qt. Updated embedding_in_qt* examples - DSD

2006-08-27 mahtext2.py: Now the fonts are searched in the OS font dir and

in the mpl-data dir. Also env is not a dict anymore. - ES

2006-08-26 minor changes to __init__.py, mathtex2_demo.py. Added matplotlibrc

key "mathtext.mathtext2" (removed the key "mathtext2") - ES

2006-08-21 mathtext2.py: Initial support for fractions

Updated the mathtext2_demo.py _mathtext_data.py: removed "" from the unicode dicts mathtext.py: Minor modification (because of _mathtext_data.py)- ES

2006-08-20 Added mathtext2.py: Replacement for mathtext.py. Supports _ ^,

rm, cal etc., sin, cos etc., unicode, recursive nestings, inline math mode. The only backend currently supported is Agg __init__.py: added new rc params for mathtext2 added mathtext2_demo.py example - ES

2006-08-19 Added embedding_in_qt4.py example - DSD

2006-08-11 Added scale free Ellipse patch for Agg - CM

2006-08-10 Added converters to and from julian dates to matplotlib.dates

(num2julian and julian2num) - JDH

2006-08-08 Fixed widget locking so multiple widgets could share the

event handling - JDH

2006-08-07 Added scale free Ellipse patch to SVG and PS - CM

2006-08-05 Re-organized imports in numerix for numpy 1.0b2 -- TEO

2006-08-04 Added draw_markers to PDF backend. - JKS

2006-08-01 Fixed a bug in postscript's rendering of dashed lines - DSD

2006-08-01 figure.py: savefig() update docstring to add support for 'format'

argument. backend_cairo.py: print_figure() add support 'format' argument. - SC

2006-07-31 Don't let postscript's xpdf distiller compress images - DSD

2006-07-31 Added shallowcopy() methods to all Transformations;

removed copy_bbox_transform and copy_bbox_transform_shallow from transforms.py; added offset_copy() function to transforms.py to facilitate positioning artists with offsets. See examples/transoffset.py. - EF

2006-07-31 Don't let postscript's xpdf distiller compress images - DSD

2006-07-29 Fixed numerix polygon bug reported by Nick Fotopoulos.

Added inverse_numerix_xy() transform method. Made autoscale_view() preserve axis direction (e.g., increasing down).- EF

2006-07-28 Added shallow bbox copy routine for transforms -- mainly

useful for copying transforms to apply offset to. - JDH

2006-07-28 Added resize method to FigureManager class

for Qt and Gtk backend - CM

2006-07-28 Added subplots_adjust button to Qt backend - CM

2006-07-26 Use numerix more in collections.

Quiver now handles masked arrays. - EF

2006-07-22 Fixed bug #1209354 - DSD

2006-07-22 make scatter() work with the kwarg "color". Closes bug

1285750 - DSD

2006-07-20 backend_cairo.py: require pycairo 1.2.0.

print_figure() update to output SVG using cairo.

2006-07-19 Added blitting for Qt4Agg - CM

2006-07-19 Added lasso widget and example examples/lasso_demo.py - JDH

2006-07-18 Added blitting for QtAgg backend - CM

2006-07-17 Fixed bug #1523585: skip nans in semilog plots - DSD

2006-07-12 Add support to render the scientific notation label

over the right-side y-axis - DSD

2006-07-11 Released 0.87.4 at revision 2558

2006-07-07 Fixed a usetex bug with older versions of latex - DSD

2006-07-07 Add compatibility for NumPy 1.0 - TEO

2006-06-29 Added a Qt4Agg backend. Thank you James Amundson - DSD

2006-06-26 Fixed a usetex bug. On Windows, usetex will process

postscript output in the current directory rather than in a temp directory. This is due to the use of spaces and tildes in windows paths, which cause problems with latex. The subprocess module is no longer used. - DSD

2006-06-22 Various changes to bar(), barh(), and hist().

Added 'edgecolor' keyword arg to bar() and barh(). The x and y args in barh() have been renamed to width and bottom respectively, and their order has been swapped to maintain a (position, value) order ala matlab. left, height, width and bottom args can now all be scalars or sequences. barh() now defaults to edge alignment instead of center alignment. Added a keyword arg 'align' to bar(), barh() and hist() that controls between edge or center bar alignment. Fixed ignoring the rcParams['patch.facecolor'] for bar color in bar() and barh(). Fixed ignoring the rcParams['lines.color'] for error bar color in bar() and barh(). Fixed a bug where patches would be cleared when error bars were plotted if rcParams['axes.hold'] was False. - MAS

2006-06-22 Added support for numerix 2-D arrays as alternatives to

a sequence of (x,y) tuples for specifying paths in collections, quiver, contour, pcolor, transforms. Fixed contour bug involving setting limits for color mapping. Added numpy-style all() to numerix. - EF

2006-06-20 Added custom FigureClass hook to pylab interface - see

examples/custom_figure_class.py

2006-06-16 Added colormaps from gist (gist_earth, gist_stern,

gist_rainbow, gist_gray, gist_yarg, gist_heat, gist_ncar) - JW

2006-06-16 Added a pointer to parent in figure canvas so you can

access the container with fig.canvas.manager. Useful if you want to set the window title, e.g., in gtk fig.canvas.manager.window.set_title, though a GUI neutral method would be preferable JDH

2006-06-16 Fixed colorbar.py to handle indexed colors (i.e.,

norm = no_norm()) by centering each colored region on its index. - EF

2006-06-15 Added scalex and scaley to Axes.autoscale_view to support

selective autoscaling just the x or y axis, and supported these command in plot so you can say plot(something, scaley=False) and just the x axis will be autoscaled. Modified axvline and axhline to support this, so for example axvline will no longer autoscale the y axis. JDH

2006-06-13 Fix so numpy updates are backward compatible - TEO

2006-06-12 Updated numerix to handle numpy restructuring of

oldnumeric - TEO

2006-06-12 Updated numerix.fft to handle numpy restructuring

Added ImportError to numerix.linear_algebra for numpy -TEO

2006-06-11 Added quiverkey command to pylab and Axes, using

QuiverKey class in quiver.py. Changed pylab and Axes to use quiver2 if possible, but drop back to the newly-renamed quiver_classic if necessary. Modified examples/quiver_demo.py to illustrate the new quiver and quiverkey. Changed LineCollection implementation slightly to improve compatibility with PolyCollection. - EF

2006-06-11 Fixed a usetex bug for windows, running latex on files

with spaces in their names or paths was failing - DSD

2006-06-09 Made additions to numerix, changes to quiver to make it

work with all numeric flavors. - EF

2006-06-09 Added quiver2 function to pylab and method to axes,

with implementation via a Quiver class in quiver.py. quiver2 will replace quiver before the next release; it is placed alongside it initially to facilitate testing and transition. See also examples/quiver2_demo.py. - EF

2006-06-08 Minor bug fix to make ticker.py draw proper minus signs

with usetex - DSD

2006-06-06 Released 0.87.3 at revision 2432

2006-05-30 More partial support for polygons with outline or fill,

but not both. Made LineCollection inherit from ScalarMappable. - EF

2006-05-29 Yet another revision of aspect-ratio handling. - EF

2006-05-27 Committed a patch to prevent stroking zero-width lines in

the svg backend - DSD

2006-05-24 Fixed colorbar positioning bug identified by Helge

Avlesen, and improved the algorithm; added a 'pad' kwarg to control the spacing between colorbar and parent axes. - EF

2006-05-23 Changed color handling so that collection initializers

can take any mpl color arg or sequence of args; deprecated float as grayscale, replaced by string representation of float. - EF

2006-05-19 Fixed bug: plot failed if all points were masked - EF

2006-05-19 Added custom symbol option to scatter - JDH

2006-05-18 New example, multi_image.py; colorbar fixed to show

offset text when the ScalarFormatter is used; FixedFormatter augmented to accept and display offset text. - EF

2006-05-14 New colorbar; old one is renamed to colorbar_classic.

New colorbar code is in colorbar.py, with wrappers in figure.py and pylab.py. Fixed aspect-handling bug reported by Michael Mossey. Made backend_bases.draw_quad_mesh() run.- EF

2006-05-08 Changed handling of end ranges in contourf: replaced

"clip-ends" kwarg with "extend". See docstring for details. -EF

2006-05-08 Added axisbelow to rc - JDH

2006-05-08 If using PyGTK require version 2.2+ - SC

2006-04-19 Added compression support to PDF backend, controlled by

new pdf.compression rc setting. - JKS

2006-04-19 Added Jouni's PDF backend

2006-04-18 Fixed a bug that caused agg to not render long lines

2006-04-16 Masked array support for pcolormesh; made pcolormesh support the

same combinations of X,Y,C dimensions as pcolor does; improved (I hope) description of grid used in pcolor, pcolormesh. - EF

2006-04-14 Reorganized axes.py - EF

2006-04-13 Fixed a bug Ryan found using usetex with sans-serif fonts and

exponential tick labels - DSD

2006-04-11 Refactored backend_ps and backend_agg to prevent module-level

texmanager imports. Now these imports only occur if text.usetex rc setting is true - DSD

2006-04-10 Committed changes required for building mpl on win32

platforms with visual studio. This allows wxpython blitting for fast animations. - CM

2006-04-10 Fixed an off-by-one bug in Axes.change_geometry.

2006-04-10 Fixed bug in pie charts where wedge wouldn't have label in

legend. Submitted by Simon Hildebrandt. - ADS

2006-05-06 Usetex makes temporary latex and dvi files in a temporary

directory, rather than in the user's current working directory - DSD

2006-04-05 Applied Ken's wx deprecation warning patch closing sf patch

#1465371 - JDH

2006-04-05 Added support for the new API in the postscript backend.

Allows values to be masked using nan's, and faster file creation - DSD

2006-04-05 Use python's subprocess module for usetex calls to

external programs. subprocess catches when they exit abnormally so an error can be raised. - DSD

2006-04-03 Fixed the bug in which widgets would not respond to

events. This regressed the twinx functionality, so I also updated subplots_adjust to update axes that share an x or y with a subplot instance. - CM

2006-04-02 Moved PBox class to transforms and deleted pbox.py;

made pylab axis command a thin wrapper for Axes.axis; more tweaks to aspect-ratio handling; fixed Axes.specgram to account for the new imshow default of unit aspect ratio; made contour set the Axes.dataLim. - EF

2006-03-31 Fixed the Qt "Underlying C/C++ object deleted" bug. - JRE

2006-03-31 Applied Vasily Sulatskov's Qt Navigation Toolbar enhancement. - JRE

2006-03-31 Ported Norbert's rewriting of Halldor's stineman_interp

algorithm to make it numerix compatible and added code to matplotlib.mlab. See examples/interp_demo.py - JDH

2006-03-30 Fixed a bug in aspect ratio handling; blocked potential

crashes when panning with button 3; added axis('image') support. - EF

2006-03-28 More changes to aspect ratio handling; new PBox class

in new file pbox.py to facilitate resizing and repositioning axes; made PolarAxes maintain unit aspect ratio. - EF

2006-03-23 Refactored TextWithDash class to inherit from, rather than

delegate to, the Text class. Improves object inspection and closes bug # 1357969 - DSD

2006-03-22 Improved aspect ratio handling, including pylab interface.

Interactive resizing, pan, zoom of images and plots (including panels with a shared axis) should work. Additions and possible refactoring are still likely. - EF

2006-03-21 Added another colorbrewer colormap (RdYlBu) - JSWHIT

2006-03-21 Fixed tickmarks for logscale plots over very large ranges.

Closes bug # 1232920 - DSD

2006-03-21 Added Rob Knight's arrow code; see examples/arrow_demo.py - JDH

2006-03-20 Added support for masking values with nan's, using ADS's

isnan module and the new API. Works for *Agg backends - DSD

2006-03-20 Added contour.negative_linestyle rcParam - ADS

2006-03-20 Added _isnan extension module to test for nan with Numeric

- ADS

2006-03-17 Added Paul and Alex's support for faceting with quadmesh

in sf patch 1411223 - JDH

2006-03-17 Added Charle Twardy's pie patch to support colors=None.

Closes sf patch 1387861 - JDH

2006-03-17 Applied sophana's patch to support overlapping axes with

toolbar navigation by toggling activation with the 'a' key. Closes sf patch 1432252 - JDH

2006-03-17 Applied Aarre's linestyle patch for backend EMF; closes sf

patch 1449279 - JDH

2006-03-17 Applied Jordan Dawe's patch to support kwarg properties

for grid lines in the grid command. Closes sf patch 1451661 - JDH

2006-03-17 Center postscript output on page when using usetex - DSD

2006-03-17 subprocess module built if Python <2.4 even if subprocess

can be imported from an egg - ADS

2006-03-17 Added _subprocess.c from Python upstream and hopefully

enabled building (without breaking) on Windows, although not tested. - ADS

2006-03-17 Updated subprocess.py to latest Python upstream and

reverted name back to subprocess.py - ADS

2006-03-16 Added John Porter's 3D handling code

2006-03-16 Released 0.87.2 at revision 2150

2006-03-15 Fixed bug in MaxNLocator revealed by daigos@infinito.it.

The main change is that Locator.nonsingular now adjusts vmin and vmax if they are nearly the same, not just if they are equal. A new kwarg, "tiny", sets the threshold. - EF

2006-03-14 Added import of compatibility library for newer numpy

linear_algebra - TEO

2006-03-12 Extended "load" function to support individual columns and

moved "load" and "save" into matplotlib.mlab so they can be used outside of pylab -- see examples/load_converter.py - JDH

2006-03-12 Added AutoDateFormatter and AutoDateLocator submitted

by James Evans. Try the load_converter.py example for a demo. - ADS

2006-03-11 Added subprocess module from python-2.4 - DSD

2006-03-11 Fixed landscape orientation support with the usetex

option. The backend_ps print_figure method was getting complicated, I added a _print_figure_tex method to maintain some degree of sanity - DSD

2006-03-11 Added "papertype" savefig kwarg for setting

postscript papersizes. papertype and ps.papersize rc setting can also be set to "auto" to autoscale pagesizes - DSD

2006-03-09 Apply P-J's patch to make pstoeps work on windows

patch report # 1445612 - DSD

2006-03-09 Make backend rc parameter case-insensitive - DSD

2006-03-07 Fixed bug in backend_ps related to C0-C6 papersizes,

which were causing problems with postscript viewers. Supported page sizes include letter, legal, ledger, A0-A10, and B0-B10 - DSD

2006-03-07 Released 0.87.1

2006-03-04 backend_cairo.py:

fix get_rgb() bug reported by Keith Briggs. Require pycairo 1.0.2. Support saving png to file-like objects. - SC

2006-03-03 Fixed pcolor handling of vmin, vmax - EF

2006-03-02 improve page sizing with usetex with the latex

geometry package. Closes bug # 1441629 - DSD

2006-03-02 Fixed dpi problem with usetex png output. Accepted a

modified version of patch # 1441809 - DSD

2006-03-01 Fixed axis('scaled') to deal with case xmax < xmin - JSWHIT

2006-03-01 Added reversed colormaps (with '_r' appended to name) - JSWHIT

2006-02-27 Improved eps bounding boxes with usetex - DSD

2006-02-27 Test svn commit, again!

2006-02-27 Fixed two dependency checking bugs related to usetex

on Windows - DSD

2006-02-27 Made the rc deprecation warnings a little more human

readable.

2006-02-26 Update the previous gtk.main_quit() bug fix to use gtk.main_level()

- SC

2006-02-24 Implemented alpha support in contour and contourf - EF

2006-02-22 Fixed gtk main quit bug when quit was called before

mainloop. - JDH

2006-02-22 Small change to colors.py to workaround apparent

bug in numpy masked array module - JSWHIT

2006-02-22 Fixed bug in ScalarMappable.to_rgba() reported by

Ray Jones, and fixed incorrect fix found by Jeff Whitaker - EF

2006-02-22 Released 0.87

2006-02-21 Fixed portrait/landscape orientation in postscript backend - DSD

2006-02-21 Fix bug introduced in yesterday's bug fix - SC

2006-02-20 backend_gtk.py FigureCanvasGTK.draw(): fix bug reported by

David Tremouilles - SC

2006-02-20 Remove the "pygtk.require('2.4')" error from

examples/embedding_in_gtk2.py - SC

2006-02-18 backend_gtk.py FigureCanvasGTK.draw(): simplify to use (rather than

duplicate) the expose_event() drawing code - SC

2006-02-12 Added stagger or waterfall plot capability to LineCollection;

illustrated in examples/collections.py. - EF

2006-02-11 Massive cleanup of the usetex code in the postscript backend. Possibly

fixed the clipping issue users were reporting with older versions of ghostscript - DSD

2006-02-11 Added autolim kwarg to axes.add_collection. Changed

collection get_verts() methods accordingly. - EF

2006-02-09 added a temporary rc parameter text.dvipnghack, to allow Mac users to get

results with the usetex option. - DSD

2006-02-09 Fixed a bug related to setting font sizes with the usetex option. - DSD

2006-02-09 Fixed a bug related to usetex's latex code. - DSD

2006-02-09 Modified behavior of font.size rc setting. You should define font.size in pts,

which will set the "medium" or default fontsize. Special text sizes like axis labels or tick labels can be given relative font sizes like small, large, x-large, etc. and will scale accordingly. - DSD

2006-02-08 Added py2exe specific datapath check again. Also added new

py2exe helper function get_py2exe_datafiles for use in py2exe setup.py scripts. - CM

2006-02-02 Added box function to pylab

2006-02-02 Fixed a problem in setupext.py, tk library formatted in unicode

caused build problems - DSD

2006-02-01 Dropped TeX engine support in usetex to focus on LaTeX. - DSD

2006-01-29 Improved usetex option to respect the serif, sans-serif, monospace,

and cursive rc settings. Removed the font.latex.package rc setting, it is no longer required - DSD

2006-01-29 Fixed tex's caching to include font.family rc information - DSD

2006-01-29 Fixed subpixel rendering bug in *Agg that was causing

uneven gridlines - JDH

2006-01-28 Added fontcmd to backend_ps's RendererPS.draw_tex, to support other

font families in eps output - DSD

2006-01-28 Added MaxNLocator to ticker.py, and changed contour.py to

use it by default. - EF

2006-01-28 Added fontcmd to backend_ps's RendererPS.draw_tex, to support other

font families in eps output - DSD

2006-01-27 Buffered reading of matplotlibrc parameters in order to allow

'verbose' settings to be processed first (allows verbose.report during rc validation process) - DSD

2006-01-27 Removed setuptools support from setup.py and created a

separate setupegg.py file to replace it. - CM

2006-01-26 Replaced the ugly datapath logic with a cleaner approach from

http://wiki.python.org/moin/DistutilsInstallDataScattered. Overrides the install_data command. - CM

2006-01-24 Don't use character typecodes in cntr.c --- changed to use

defined typenumbers instead. - TEO

2006-01-24 Fixed some bugs in usetex's and ps.usedistiller's dependency

2006-01-24 Added masked array support to scatter - EF

2006-01-24 Fixed some bugs in usetex's and ps.usedistiller's dependency

checking - DSD

2006-01-24 Released 0.86.2

2006-01-20 Added a converters dict to pylab load to convert selected

columns to float -- especially useful for files with date strings, uses a datestr2num converter - JDH

2006-01-20 Added datestr2num to matplotlib dates to convert a string

or sequence of strings to a matplotlib datenum

2006-01-18 Added quadrilateral pcolormesh patch 1409190 by Alex Mont

and Paul Kienzle -- this is *Agg only for now. See examples/quadmesh_demo.py - JDH

2006-01-18 Added Jouni's boxplot patch - JDH

2006-01-18 Added comma delimiter for pylab save - JDH

2006-01-12 Added Ryan's legend patch - JDH

2006-1-12 Fixed numpy / numeric to use .dtype.char to keep in SYNC with numpy SVN

2006-1-11 Released 0.86.1

2006-1-11 Fixed setup.py for win32 build and added rc template to the MANIFEST.in

2006-1-10 Added xpdf distiller option. matplotlibrc ps.usedistiller can now be

none, false, ghostscript, or xpdf. Validation checks for dependencies. This needs testing, but the xpdf option should produce the highest-quality output and small file sizes - DSD

2006-01-10 For the usetex option, backend_ps now does all the LaTeX work in the

os's temp directory - DSD

2006-1-10 Added checks for usetex dependencies. - DSD

2006-1-9 Released 0.86

2006-1-4 Changed to support numpy (new name for scipy_core) - TEO

2006-1-4 Added Mark's scaled axes patch for shared axis

2005-12-28 Added Chris Barker's build_wxagg patch - JDH

2005-12-27 Altered numerix/scipy to support new scipy package

structure - TEO

2005-12-20 Fixed Jame's Boyles date tick reversal problem - JDH

2005-12-20 Added Jouni's rc patch to support lists of keys to set on -

JDH

2005-12-12 Updated pyparsing and mathtext for some speed enhancements

(Thanks Paul McGuire) and minor fixes to scipy numerix and setuptools

2005-12-12 Matplotlib data is now installed as package_data in

the matplotlib module. This gets rid of checking the many possibilities in matplotlib._get_data_path() - CM

2005-12-11 Support for setuptools/pkg_resources to build and use

matplotlib as an egg. Still allows matplotlib to exist using a traditional distutils install. - ADS

2005-12-03 Modified setup to build matplotlibrc based on compile time

findings. It will set numerix in the order of scipy, numarray, Numeric depending on which are founds, and backend as in preference order GTKAgg, WXAgg, TkAgg, GTK, Agg, PS

2005-12-03 Modified scipy patch to support Numeric, scipy and numarray

Some work remains to be done because some of the scipy imports are broken if only the core is installed. e.g., apparently we need from scipy.basic.fftpack import * rather than from scipy.fftpack import *

2005-12-03 Applied some fixes to Nicholas Young's nonuniform image

patch

2005-12-01 Applied Alex Gontmakher hatch patch - PS only for now

2005-11-30 Added Rob McMullen's EMF patch

2005-11-30 Added Daishi's patch for scipy

2005-11-30 Fixed out of bounds draw markers segfault in agg

2005-11-28 Got TkAgg blitting working 100% (cross fingers) correctly. - CM

2005-11-27 Multiple changes in cm.py, colors.py, figure.py, image.py,

contour.py, contour_demo.py; new _cm.py, examples/image_masked.py. 1) Separated the color table data from cm.py out into a new file, _cm.py, to make it easier to find the actual code in cm.py and to add new colormaps. Also added some line breaks to the color data dictionaries. Everything from _cm.py is imported by cm.py, so the split should be transparent. 2) Enabled automatic generation of a colormap from a list of colors in contour; see modified examples/contour_demo.py. 3) Support for imshow of a masked array, with the ability to specify colors (or no color at all) for masked regions, and for regions that are above or below the normally mapped region. See examples/image_masked.py. 4) In support of the above, added two new classes, ListedColormap, and no_norm, to colors.py, and modified the Colormap class to include common functionality. Added a clip kwarg to the normalize class. Reworked color handling in contour.py, especially in the ContourLabeller mixin. - EF

2005-11-25 Changed text.py to ensure color is hashable. EF

2005-11-16 Released 0.85

2005-11-16 Changed the default default linewidth in rc to 1.0

2005-11-16 Replaced agg_to_gtk_drawable with pure pygtk pixbuf code in

backend_gtkagg. When the equivalent is doe for blit, the agg extension code will no longer be needed

2005-11-16 Added a maxdict item to cbook to prevent caches from

growing w/o bounds

2005-11-15 Fixed a colorup/colordown reversal bug in finance.py --

Thanks Gilles

2005-11-15 Applied Jouni K Steppanen's boxplot patch SF patch#1349997

- JDH

2005-11-09 added axisbelow attr for Axes to determine whether ticks and such

are above or below the actors

2005-11-08 Added Nicolas' irregularly spaced image patch

2005-11-08 Deprecated HorizontalSpanSelector and replaced with

SpanSelection that takes a third arg, direction. The new SpanSelector supports horizontal and vertical span selection, and the appropriate min/max is returned. - CM

2005-11-08 Added lineprops dialog for gtk

2005-11-03 Added FIFOBuffer class to mlab to support real time feeds

and examples/fifo_buffer.py

2005-11-01 Contributed Nickolas Young's patch for afm mathtext to

support mathtext based upon the standard postscript Symbol font when ps.usetex = True.

2005-10-26 Added support for scatter legends - thanks John Gill

2005-10-20 Fixed image clipping bug that made some tex labels

disappear. JDH

2005-10-14 Removed sqrt from dvipng 1.6 alpha channel mask.

2005-10-14 Added width kwarg to hist function

2005-10-10 Replaced all instances of os.rename with shutil.move

2005-10-05 Added Michael Brady's ydate patch

2005-10-04 Added rkern's texmanager patch

2005-09-25 contour.py modified to use a single ContourSet class

that handles filled contours, line contours, and labels; added keyword arg (clip_ends) to contourf. Colorbar modified to work with new ContourSet object; if the ContourSet has lines rather than polygons, the colorbar will follow suit. Fixed a bug introduced in 0.84, in which contourf(...,colors=...) was broken - EF

2005-09-19 Released 0.84

2005-09-14 Added a new 'resize_event' which triggers a callback with a

backend_bases.ResizeEvent object - JDH

2005-09-14 font_manager.py: removed chkfontpath from x11FontDirectory() - SC

2005-09-14 Factored out auto date locator/formatter factory code into

matplotlib.date.date_ticker_factory; applies John Bryne's quiver patch.

2005-09-13 Added Mark's axes positions history patch #1286915

2005-09-09 Added support for auto canvas resizing with

fig.set_figsize_inches(9,5,forward=True) # inches OR fig.resize(400,300) # pixels

2005-09-07 figure.py: update Figure.draw() to use the updated

renderer.draw_image() so that examples/figimage_demo.py works again. examples/stock_demo.py: remove data_clipping (which no longer exists) - SC

2005-09-06 Added Eric's tick.direction patch: in or out in rc

2005-09-06 Added Martin's rectangle selector widget

2005-09-04 Fixed a logic err in text.py that was preventing rgxsuper

from matching - JDH

2005-08-29 Committed Ken's wx blit patch #1275002

2005-08-26 colorbar modifications - now uses contourf instead of imshow

so that colors used by contourf are displayed correctly. Added two new keyword args (cspacing and clabels) that are only relevant for ContourMappable images - JSWHIT

2005-08-24 Fixed a PS image bug reported by Darren - JDH

2005-08-23 colors.py: change hex2color() to accept unicode strings as well as

normal strings. Use isinstance() instead of types.IntType etc - SC

2005-08-16 removed data_clipping line and rc property - JDH

2005-08-22 backend_svg.py: Remove redundant "x=0.0 y=0.0" from svg element.

Increase svg version from 1.0 to 1.1. Add viewBox attribute to svg element to allow SVG documents to scale-to-fit into an arbitrary viewport - SC

2005-08-16 Added Eric's dot marker patch - JDH

2005-08-08 Added blitting/animation for TkAgg - CM

2005-08-05 Fixed duplicate tickline bug - JDH

2005-08-05 Fixed a GTK animation bug that cropped up when doing

animations in gtk//gtkagg canvases that had widgets packed above them

2005-08-05 Added Clovis Goldemberg patch to the tk save dialog

2005-08-04 Removed origin kwarg from backend.draw_image. origin is

handled entirely by the frontend now.

2005-07-03 Fixed a bug related to TeX commands in backend_ps

2005-08-03 Fixed SVG images to respect upper and lower origins.

2005-08-03 Added flipud method to image and removed it from to_str.

2005-07-29 Modified figure.figaspect to take an array or number;

modified backend_svg to write utf-8 - JDH

2005-07-30 backend_svg.py: embed png image files in svg rather than linking

to a separate png file, fixes bug #1245306 (thanks to Norbert Nemec for the patch) - SC

2005-07-29 Released 0.83.2

2005-07-27 Applied SF patch 1242648: minor rounding error in

IndexDateFormatter in dates.py

2005-07-27 Applied sf patch 1244732: Scale axis such that circle

looks like circle - JDH

2005-07-29 Improved message reporting in texmanager and backend_ps - DSD

2005-07-28 backend_gtk.py: update FigureCanvasGTK.draw() (needed due to the

recent expose_event() change) so that examples/anim.py works in the usual way - SC

2005-07-26 Added new widgets Cursor and HorizontalSpanSelector to

matplotlib.widgets. See examples/widgets/cursor.py and examples/widgets/span_selector.py - JDH

2005-07-26 added draw event to mpl event hierarchy -- triggered on

figure.draw

2005-07-26 backend_gtk.py: allow 'f' key to toggle window fullscreen mode

2005-07-26 backend_svg.py: write "<.../>" elements all on one line and remove

surplus spaces - SC

2005-07-25 backend_svg.py: simplify code by deleting GraphicsContextSVG and

RendererSVG.new_gc(), and moving the gc.get_capstyle() code into RendererSVG._get_gc_props_svg() - SC

2005-07-24 backend_gtk.py: call FigureCanvasBase.motion_notify_event() on

all motion-notify-events, not just ones where a modifier key or button has been pressed (fixes bug report from Niklas Volbers) - SC

2005-07-24 backend_gtk.py: modify print_figure() use own pixmap, fixing

problems where print_figure() overwrites the display pixmap. return False from all button/key etc events - to allow the event to propagate further - SC

2005-07-23 backend_gtk.py: change expose_event from using set_back_pixmap();

clear() to draw_drawable() - SC

2005-07-23 backend_gtk.py: removed pygtk.require()

matplotlib/__init__.py: delete 'FROZEN' and 'McPLError' which are no longer used - SC

2005-07-22 backend_gdk.py: removed pygtk.require() - SC

2005-07-21 backend_svg.py: Remove unused imports. Remove methods doc strings

which just duplicate the docs from backend_bases.py. Rename draw_mathtext to _draw_mathtext. - SC

2005-07-17 examples/embedding_in_gtk3.py: new example demonstrating placing

a FigureCanvas in a gtk.ScrolledWindow - SC

2005-07-14 Fixed a Windows related bug (#1238412) in texmanager - DSD

2005-07-11 Fixed color kwarg bug, setting color=1 or 0 caused an

exception - DSD

2005-07-07 Added Eric's MA set_xdata Line2D fix - JDH

2005-07-06 Made HOME/.matplotlib the new config dir where the

matplotlibrc file, the ttf.cache, and the tex.cache live. The new default filenames in .matplotlib have no leading dot and are not hidden. e.g., the new names are matplotlibrc tex.cache ttffont.cache. This is how ipython does it so it must be right. If old files are found, a warning is issued and they are moved to the new location. Also fixed texmanager to put all files, including temp files in ~/.matplotlib/tex.cache, which allows you to usetex in non-writable dirs.

2005-07-05 Fixed bug #1231611 in subplots adjust layout. The problem

was that the text caching mechanism was not using the transformation affine in the key. - JDH

2005-07-05 Fixed default backend import problem when using API (SF bug

1209354 - see API_CHANGES for more info - JDH

2005-07-04 backend_gtk.py: require PyGTK version 2.0.0 or higher - SC

2005-06-30 setupext.py: added numarray_inc_dirs for building against

numarray when not installed in standard location - ADS

2005-06-27 backend_svg.py: write figure width, height as int, not float.

Update to fix some of the pychecker warnings - SC

2005-06-23 Updated examples/agg_test.py to demonstrate curved paths

and fills - JDH

2005-06-21 Moved some texmanager and backend_agg tex caching to class

level rather than instance level - JDH

2005-06-20 setupext.py: fix problem where _nc_backend_gdk is installed to the

wrong directory - SC

2005-06-19 Added 10.4 support for CocoaAgg. - CM

2005-06-18 Move Figure.get_width_height() to FigureCanvasBase and return

int instead of float. - SC

2005-06-18 Applied Ted Drain's QtAgg patch: 1) Changed the toolbar to

be a horizontal bar of push buttons instead of a QToolbar and updated the layout algorithms in the main window accordingly. This eliminates the ability to drag and drop the toolbar and detach it from the window. 2) Updated the resize algorithm in the main window to show the correct size for the plot widget as

requested. This works almost correctly right now. It looks to me like the final size of the widget is off by the border of the main window but I haven't figured out a way to get that information yet. We could just add a small margin to the new size but that seems a little hacky. 3) Changed the x/y location label to be in the toolbar like the Tk backend instead of as a status line at the bottom of the widget. 4) Changed the toolbar pixmaps to use the ppm files instead of the png files. I noticed that the Tk backend buttons looked much nicer and it uses the ppm files so I switched them.

2005-06-17 Modified the gtk backend to not queue mouse motion events.

This allows for live updates when dragging a slider. - CM

2005-06-17 Added starter CocoaAgg backend. Only works on OS 10.3 for

now and requires PyObjC. (10.4 is high priority) - CM

2005-06-17 Upgraded pyparsing and applied Paul McGuire's suggestions

for speeding things up. This more than doubles the speed of mathtext in my simple tests. JDH

2005-06-16 Applied David Cooke's subplot make_key patch

2005-06-15 0.82 released

2005-06-15 Added subplot config tool to GTK* backends -- note you must

now import the NavigationToolbar2 from your backend of choice rather than from backend_gtk because it needs to know about the backend specific canvas -- see examples/embedding_in_gtk2.py. Ditto for wx backend -- see examples/embedding_in_wxagg.py

2005-06-15 backend_cairo.py: updated to use pycairo 0.5.0 - SC

2005-06-14 Wrote some GUI neutral widgets (Button, Slider,

RadioButtons, CheckButtons) in matplotlib.widgets. See examples/widgets/*.py - JDH

2005-06-14 Exposed subplot parameters as rc vars and as the fig

SubplotParams instance subplotpars. See figure.SubplotParams, figure.Figure.subplots_adjust and the pylab method subplots_adjust and examples/subplots_adjust.py . Also added a GUI neutral widget for adjusting subplots, see examples/subplot_toolbar.py - JDH

2005-06-13 Exposed cap and join style for lines with new rc params and

line properties

lines.dash_joinstyle : miter # miter|round|bevel lines.dash_capstyle : butt # butt|round|projecting lines.solid_joinstyle : miter # miter|round|bevel lines.solid_capstyle : projecting # butt|round|projecting

2005-06-13 Added kwargs to Axes init

2005-06-13 Applied Baptiste's tick patch - JDH

2005-06-13 Fixed rc alias 'l' bug reported by Fernando by removing

aliases for mainlevel rc options. - JDH

2005-06-10 Fixed bug #1217637 in ticker.py - DSD

2005-06-07 Fixed a bug in texmanager.py: .aux files not being removed - DSD

2005-06-08 Added Sean Richard's hist binning fix -- see API_CHANGES - JDH

2005-06-07 Fixed a bug in texmanager.py: .aux files not being removed

- DSD

2005-06-07 matplotlib-0.81 released

2005-06-06 Added autoscale_on prop to axes

2005-06-06 Added Nick's picker "among" patch - JDH

2005-06-05 Fixed a TeX/LaTeX font discrepency in backend_ps. - DSD

2005-06-05 Added a ps.distill option in rc settings. If True, postscript

output will be distilled using ghostscript, which should trim the file size and allow it to load more quickly. Hopefully this will address the issue of large ps files due to font definitions. Tested with gnu-ghostscript-8.16. - DSD

2005-06-03 Improved support for tex handling of text in backend_ps. - DSD

2005-06-03 Added rc options to render text with tex or latex, and to select

the latex font package. - DSD

2005-06-03 Fixed a bug in ticker.py causing a ZeroDivisionError

2005-06-02 backend_gtk.py remove DBL_BUFFER, add line to expose_event to

try to fix pygtk 2.6 redraw problem - SC

2005-06-01 The default behavior of ScalarFormatter now renders scientific

notation and large numerical offsets in a label at the end of the axis. - DSD

2005-06-01 Added Nicholas' frombyte image patch - JDH

2005-05-31 Added vertical TeX support for agg - JDH

2005-05-31 Applied Eric's cntr patch - JDH

2005-05-27 Finally found the pesky agg bug (which Maxim was kind

enough to fix within hours) that was causing a segfault in the win32 cached marker drawing. Now windows users can get the enormouse performance benefits of caced markers w/o those occasional pesy screenshots. - JDH

2005-05-27 Got win32 build system working again, using a more recent

version of gtk and pygtk in the win32 build, gtk 2.6 from https://web.archive.org/web/20050527002647/https://www.gimp.org/~tml/gimp/win32/downloads.html (you will also need libpng12.dll to use these). I haven't tested whether this binary build of mpl for win32 will work with older gtk runtimes, so you may need to upgrade.

2005-05-27 Fixed bug where 2nd wxapp could be started if using wxagg

backend. - ADS

2005-05-26 Added Daishi text with dash patch -- see examples/dashtick.py

2005-05-26 Moved backend_latex functionality into backend_ps. If

text.usetex=True, the PostScript backend will use LaTeX to generate the .ps or .eps file. Ghostscript is required for eps output. - DSD

2005-05-24 Fixed alignment and color issues in latex backend. - DSD

2005-05-21 Fixed raster problem for small rasters with dvipng -- looks

like it was a premultipled alpha problem - JDH

2005-05-20 Added linewidth and faceted kwarg to scatter to control

edgewidth and color. Also added autolegend patch to inspect line segments.

2005-05-18 Added Orsay and JPL qt fixes - JDH

2005-05-17 Added a psfrag latex backend -- some alignment issues need

to be worked out. Run with -dLaTeX and a *.tex file and *.eps file are generated. latex and dvips the generated latex file to get ps output. Note xdvi *does* not work, you must generate ps.- JDH

2005-05-13 Added Florent Rougon's Axis set_label1

patch

2005-05-17 pcolor optimization, fixed bug in previous pcolor patch - JSWHIT

2005-05-16 Added support for masked arrays in pcolor - JSWHIT

2005-05-12 Started work on TeX text for antigrain using pngdvi -- see

examples/tex_demo.py and the new module matplotlib.texmanager. Rotated text not supported and rendering small glyps is not working right yet. BUt large fontsizes and/or high dpi saved figs work great.

2005-05-10 New image resize options interpolation options. New values

for the interp kwarg are

'nearest', 'bilinear', 'bicubic', 'spline16', 'spline36', 'hanning', 'hamming', 'hermite', 'kaiser', 'quadric', 'catrom', 'gaussian', 'bessel', 'mitchell', 'sinc', 'lanczos', 'blackman'

See help(imshow) for details, particularly the interpolation, filternorm and filterrad kwargs

2005-05-10 Applied Eric's contour mem leak fixes - JDH

2005-05-10 Extended python agg wrapper and started implementing

backend_agg2, an agg renderer based on the python wrapper. This will be more flexible and easier to extend than the current backend_agg. See also examples/agg_test.py - JDH

2005-05-09 Added Marcin's no legend patch to exclude lines from the

autolegend builder

plot(x, y, label='nolegend')

2005-05-05 Upgraded to agg23

2005-05-05 Added newscalarformatter_demo.py to examples. -DSD

2005-05-04 Added NewScalarFormatter. Improved formatting of ticklabels,

scientific notation, and the ability to plot large large numbers with small ranges, by determining a numerical offset. See ticker.NewScalarFormatter for more details. -DSD

2005-05-03 Added the option to specify a delimiter in pylab.load -DSD

2005-04-28 Added Darren's line collection example

2005-04-28 Fixed aa property in agg - JDH

2005-04-27 Set postscript page size in .matplotlibrc - DSD

2005-04-26 Added embedding in qt example. - JDH

2005-04-14 Applied Michael Brady's qt backend patch: 1) fix a bug

where keyboard input was grabbed by the figure and not released 2) turn on cursor changes 3) clean up a typo and commented-out print statement. - JDH

2005-04-14 Applied Eric Firing's masked data lines patch and contour

patch. Support for masked arrays has been added to the plot command and to the Line2D object. Only the valid points are plotted. A "valid_only" kwarg was added to the get_xdata() and get_ydata() methods of Line2D; by default it is False, so that the original data arrays are returned. Setting it to True returns the plottable points. - see examples/masked_demo.py - JDH

2005-04-13 Applied Tim Leslie's arrow key event handling patch - JDH

0.80 released

2005-04-11 Applied a variant of rick's xlim/ylim/axis patch. These

functions now take kwargs to let you selectively alter only the min or max if desired. e.g., xlim(xmin=2) or axis(ymax=3). They always return the new lim. - JDH

2005-04-11 Incorporated Werner's wx patch -- wx backend should be

compatible with wxpython2.4 and recent versions of 2.5. Some early versions of wxpython 2.5 will not work because there was a temporary change in the dc API that was rolled back to make it 2.4 compliant

2005-04-11 modified tkagg show so that new figure window pops up on

call to figure

2005-04-11 fixed wxapp init bug

2005-04-02 updated backend_ps.draw_lines, draw_markers for use with the

new API - DSD

2005-04-01 Added editable polygon example

2005-03-31 0.74 released

2005-03-30 Fixed and added checks for floating point inaccuracy in

ticker.Base - DSD

2005-03-30 updated /ellipse definition in backend_ps.py to address bug

#1122041 - DSD

2005-03-29 Added unicode support for Agg and PS - JDH

2005-03-28 Added Jarrod's svg patch for text - JDH

2005-03-28 Added Ludal's arrow and quiver patch - JDH

2005-03-28 Added label kwarg to Axes to facilitate forcing the

creation of new Axes with otherwise identical attributes

2005-03-28 Applied boxplot and OSX font search patches

2005-03-27 Added ft2font NULL check to fix Japanase font bug - JDH

2005-03-27 Added sprint legend patch plus John Gill's tests and fix --

see examples/legend_auto.py - JDH

2005-03-19 0.73.1 released

2005-03-19 Reverted wxapp handling because it crashed win32 - JDH

2005-03-18 Add .number attribute to figure objects returned by figure() - FP

2005-03-18 0.73 released

2005-03-16 Fixed labelsep bug

2005-03-16 Applied Darren's ticker fix for small ranges - JDH

2005-03-16 Fixed tick on horiz colorbar - JDH

2005-03-16 Added Japanese winreg patch - JDH

2005-03-15 backend_gtkagg.py: changed to use double buffering, this fixes

the problem reported Joachim Berdal Haga - "Parts of plot lagging from previous frame in animation". Tested with anim.py and it makes no noticeable difference to performance (23.7 before, 23.6 after) - SC

2005-03-14 add src/_backend_gdk.c extension to provide a substitute function

for pixbuf.get_pixels_array(). Currently pixbuf.get_pixels_array() only works with Numeric, and then only works if pygtk has been compiled with Numeric support. The change provides a function pixbuf_get_pixels_array() which works with Numeric and numarray and is always available. It means that backend_gtk should be able to display images and mathtext in all circumstances. - SC

2005-03-11 Upgraded CXX to 5.3.1

2005-03-10 remove GraphicsContextPS.set_linestyle()

and GraphicsContextSVG.set_linestyle() since they do no more than the base class GraphicsContext.set_linestyle() - SC

2005-03-09 Refactored contour functionality into dedicated module

2005-03-09 Added Eric's contourf updates and Nadia's clabel functionality

2005-03-09 Moved colorbar to figure.Figure to expose it for API developers

- JDH

2005-03-09 backend_cairo.py: implemented draw_markers() - SC

2005-03-09 cbook.py: only use enumerate() (the python version) if the builtin

version is not available.

Add new function 'izip' which is set to itertools.izip if available and the python equivalent if not available. - SC

2005-03-07 backend_gdk.py: remove PIXELS_PER_INCH from points_to_pixels(), but

still use it to adjust font sizes. This allows the GTK version of

line_styles.py to more closely match GTKAgg, previously the markers were being drawn too large. - SC

2005-03-01 Added Eric's contourf routines

2005-03-01 Added start of proper agg SWIG wrapper. I would like to

expose agg functionality directly a the user level and this module will serve that purpose eventually, and will hopefully take over most of the functionality of the current _image and _backend_agg modules. - JDH

2005-02-28 Fixed polyfit / polyval to convert input args to float

arrays - JDH

2005-02-25 Add experimental feature to backend_gtk.py to enable/disable

double buffering (DBL_BUFFER=True/False) - SC

2005-02-24 colors.py change ColorConverter.to_rgb() so it always returns rgb

(and not rgba), allow cnames keys to be cached, change the exception raised from RuntimeError to ValueError (like hex2color()) hex2color() use a regular expression to check the color string is valid - SC

2005-02-23 Added rc param ps.useafm so backend ps can use native afm

fonts or truetype. afme breaks mathtext but causes much smaller font sizes and may result in images that display better in some contexts (e.g., pdfs incorporated into latex docs viewed in acrobat reader). I would like to extend this approach to allow the user to use truetype only for mathtext, which should be easy.

2005-02-23 Used sequence protocol rather than tuple in agg collection

drawing routines for greater flexibility - JDH

2005-02-22 0.72.1 released

2005-02-21 fixed linestyles for collections -- contour now dashes for

levels <0

2005-02-21 fixed ps color bug - JDH

2005-02-15 fixed missing qt file

2005-02-15 banished error_msg and report_error. Internal backend

methods like error_msg_gtk are preserved. backend writers, check your backends, and diff against 0.72 to make sure I did the right thing! - JDH

2005-02-14 Added enthought traits to matplotlib tree - JDH

2005-02-14 0.72 released

2005-02-14 fix bug in cbook alltrue() and onetrue() - SC

2005-02-11 updated qtagg backend from Ted - JDH

2005-02-11 matshow fixes for figure numbering, return value and docs - FP

2005-02-09 new zorder example for fine control in zorder_demo.py - FP

2005-02-09 backend renderer draw_lines now has transform in backend,

as in draw_markers; use numerix in _backend_agg, aded small line optimization to agg

2005-02-09 subplot now deletes axes that it overlaps

2005-02-08 Added transparent support for gzipped files in load/save - Fernando

Perez (FP from now on).

2005-02-08 Small optimizations in PS backend. They may have a big impact for

large plots, otherwise they don't hurt - FP

2005-02-08 Added transparent support for gzipped files in load/save - Fernando

Perez (FP from now on).

2005-02-07 Added newstyle path drawing for markers - only implemented

in agg currently - JDH

2005-02-05 Some superscript text optimizations for ticking log plots

2005-02-05 Added some default key press events to pylab figures: 'g'

toggles grid - JDH

2005-02-05 Added some support for handling log switching for lines

that have nonpos data - JDH

2005-02-04 Added Nadia's contour patch - contour now has matlab

compatible syntax; this also fixed an unequal sized contour array bug- JDH

2005-02-04 Modified GTK backends to allow the FigureCanvas to be resized

smaller than its original size - SC

2005-02-02 Fixed a bug in dates mx2num - JDH

2005-02-02 Incorporated Fernando's matshow - JDH

2005-02-01 Added Fernando's figure num patch, including experimental

support for pylab backend switching, LineCOllection.color warns, savefig now a figure method, fixed a close(fig) bug - JDH

2005-01-31 updated datalim in contour - JDH

2005-01-30 Added backend_qtagg.py provided by Sigve Tjora - SC

2005-01-28 Added tk.inspect rc param to .matplotlibrc. IDLE users

should set tk.pythoninspect:True and interactive:True and backend:TkAgg

2005-01-28 Replaced examples/interactive.py with an updated script from

Fernando Perez - SC

2005-01-27 Added support for shared x or y axes. See

examples/shared_axis_demo.py and examples/ganged_plots.py

2005-01-27 Added Lee's patch for missing symbols leq and LEFTbracket

to _mathtext_data - JDH

2005-01-26 Added Baptiste's two scales patch -- see help(twinx) in the

pylab interface for more info. See also examples/two_scales.py

2005-01-24 Fixed a mathtext parser bug that prevented font changes in

sub/superscripts - JDH

2005-01-24 Fixed contour to work w/ interactive changes in colormaps,

clim, etc - JDH

2005-01-21 matplotlib-0.71 released

2005-01-21 Refactored numerix to solve vexing namespace issues - JDH

2005-01-21 Applied Nadia's contour bug fix - JDH

2005-01-20 Made some changes to the contour routine - particularly

region=1 seems t fix a lot of the zigzag strangeness. Added colormaps as default for contour - JDH

2005-01-19 Restored builtin names which were overridden (min, max,

abs, round, and sum) in pylab. This is a potentially significant change for those who were relying on an array version of those functions that previously overrode builtin function names. - ADS

2005-01-18 Added accents to mathtext: hat, breve, grave, bar,

acute, tilde, vec, dot, ddot. All of them have the same syntax, e.g., to make an overbar you do bar{o} or to make an o umlaut you do ddot{o}. The shortcuts are also provided, e.g., "o 'e `e ~n .x ^y - JDH

2005-01-18 Plugged image resize memory leaks - JDH

2005-01-18 Fixed some mathtext parser problems relating to superscripts

2005-01-17 Fixed a yticklabel problem for colorbars under change of

clim - JDH

2005-01-17 Cleaned up Destroy handling in wx reducing memleak/fig from
approx 800k to approx 6k- JDH

2005-01-17 Added kappa to latex_to_bakoma - JDH

2005-01-15 Support arbitrary colorbar axes and horizontal colorbars - JDH

2005-01-15 Fixed colormap number of colors bug so that the colorbar
has the same discretization as the image - JDH

2005-01-15 Added Nadia's x,y contour fix - JDH

2005-01-15 backend_cairo: added PDF support which requires pycairo 0.1.4.

Its not usable yet, but is ready for when the Cairo PDF backend matures - SC

2005-01-15 Added Nadia's x,y contour fix

2005-01-12 Fixed set clip_on bug in artist - JDH

2005-01-11 Reverted pythoninspect in tkagg - JDH

2005-01-09 Fixed a backend_bases event bug caused when an event is
triggered when location is None - JDH

2005-01-07 Add patch from Stephen Walton to fix bug in pylab.load()
when the % character is included in a comment. - ADS

2005-01-07 Added markerscale attribute to Legend class. This allows
the marker size in the legend to be adjusted relative to that in the plot. - ADS

2005-01-06 Add patch from Ben Vanhaeren to make the FigureManagerGTK vbox a

public attribute - SC

2004-12-30 Release 0.70

2004-12-28 Added coord location to key press and added a
examples/picker_demo.py

2004-12-28 Fixed coords notification in wx toolbar - JDH

2004-12-28 Moved connection and disconnection event handling to the
FigureCanvasBase. Backends now only need to connect one time for each of the button press, button release and key press/release functions. The base class deals with callbacks and multiple connections. This fixes flakiness on some backends (tk, wx) in the presence of multiple connections and/or disconnect - JDH

2004-12-27 Fixed PS mathtext bug where color was not set - Jochen
please verify correct - JDH

2004-12-27 Added Shadow class and added shadow kwarg to legend and pie

for shadow effect - JDH

2004-12-27 Added pie charts and new example/pie_demo.py

2004-12-23 Fixed an agg text rotation alignment bug, fixed some text

kwarg processing bugs, and added examples/text_rotation.py to explain and demonstrate how text rotations and alignment work in matplotlib. - JDH

2004-12-22 0.65.1 released - JDH

2004-12-22 Fixed colorbar bug which caused colorbar not to respond to

changes in colormap in some instances - JDH

2004-12-22 Refactored NavigationToolbar in tkagg to support app

embedding , init now takes (canvas, window) rather than (canvas, figman) - JDH

2004-12-21 Refactored axes and subplot management - removed

add_subplot and add_axes from the FigureManager. classic toolbar updates are done via an observer pattern on the figure using add_axobserver. Figure now maintains the axes stack (for gca) and supports axes deletion. Ported changes to GTK, Tk, Wx, and FLTK. Please test! Added delaxes - JDH

2004-12-21 Lots of image optimizations - 4x performance boost over

0.65 JDH

2004-12-20 Fixed a figimage bug where the axes is shown and modified

tkagg to move the destroy binding into the show method.

2004-12-18 Minor refactoring of NavigationToolbar2 to support

embedding in an application - JDH

2004-12-14 Added linestyle to collections (currently broken) - JDH

2004-12-14 Applied Nadia's setupext patch to fix libstdc++ link

problem with contour and solaris -JDH

2004-12-14 A number of pychecker inspired fixes, including removal of

True and False from cbook which I erroneously thought was needed for python2.2 - JDH

2004-12-14 Finished porting doc strings for set introspection.

Used silent_list for many get funcs that return lists. JDH

2004-12-13 dates.py: removed all timezone() calls, except for UTC - SC

2004-12-13 0.65 released - JDH

2004-12-13 colors.py: rgb2hex(), hex2color() made simpler (and faster), also

rgb2hex() - added round() instead of integer truncation hex2color() - changed 256.0 divisor to 255.0, so now '#ffffff' becomes (1.0,1.0,1.0) not (0.996,0.996,0.996) - SC

2004-12-11 Added ion and ioff to pylab interface - JDH

2004-12-11 backend_template.py: delete FigureCanvasTemplate.realize() - most

backends don't use it and its no longer needed

backend_ps.py, backend_svg.py: delete show() and draw_if_interactive() - they are not needed for image backends

backend_svg.py: write direct to file instead of StringIO - SC

2004-12-10 Added zorder to artists to control drawing order of lines,

patches and text in axes. See examples/zoder_demo.py - JDH

2004-12-10 Fixed colorbar bug with scatter - JDH

2004-12-10 Added Nadia Dencheva <dencheva@stsci.edu> contour code - JDH

2004-12-10 backend_cairo.py: got mathtext working - SC

2004-12-09 Added Norm Peterson's svg clipping patch

2004-12-09 Added Matthew Newville's wx printing patch

2004-12-09 Migrated matlab to pylab - JDH

2004-12-09 backend_gtk.py: split into two parts

- backend_gdk.py - an image backend
- backend_gtk.py - A GUI backend that uses GDK - SC

2004-12-08 backend_gtk.py: remove quit_after_print_xvfb(*args), show_xvfb(),

Dialog_MeasureTool(gtk.Dialog) one month after sending mail to matplotlib-users asking if anyone still uses these functions - SC

2004-12-02 backend_bases.py, backend_template.py: updated some of the method

documentation to make them consistent with each other - SC

2004-12-04 Fixed multiple bindings per event for TkAgg mpl_connect and

mpl_disconnect. Added a "test_disconnect" command line parameter to coords_demo.py JTM

2004-12-04 Fixed some legend bugs JDH

2004-11-30 Added over command for oneoff over plots. e.g., over(plot, x,
y, lw=2). Works with any plot function.

2004-11-30 Added bbox property to text - JDH

2004-11-29 Zoom to rect now respect reversed axes limits (for both
linear and log axes). - GL

2004-11-29 Added the over command to the matlab interface. over
allows you to add an overlay plot regardless of hold state. - JDH

2004-11-25 Added Printf to mplutils for printf style format string
formatting in C++ (should help write better exceptions)

2004-11-24 IMAGE_FORMAT: remove from agg and gtkagg backends as its no longer
used - SC

2004-11-23 Added matplotlib compatible set and get introspection. See
set_and_get.py

2004-11-23 applied Norbert's patched and exposed legend configuration
to kwargs - JDH

2004-11-23 backend_gtk.py: added a default exception handler - SC

2004-11-18 backend_gtk.py: change so that the backend knows about all image
formats and does not need to use IMAGE_FORMAT in other backends - SC

2004-11-18 Fixed some report_error bugs in string interpolation as
reported on SF bug tracker- JDH

2004-11-17 backend_gtkcairo.py: change so all print_figure() calls render using
Cairo and get saved using backend_gtk.print_figure() - SC

2004-11-13 backend_cairo.py: Discovered the magic number (96) required for
Cairo PS plots to come out the right size. Restored Cairo PS output and added support for landscape mode - SC

2004-11-13 Added ishold - JDH

2004-11-12 Added many new matlab colormaps - autumn bone cool copper
flag gray hot hsv jet pink prism spring summer winter - PG

2004-11-11 greatly simplify the emitted postscript code - JV

2004-11-12 Added new plotting functions spy, spy2 for sparse matrix

visualization - JDH

2004-11-11 Added rgrids, thetragrids for customizing the grid

locations and labels for polar plots - JDH

2004-11-11 make the Gtk backends build without an X-server connection - JV

2004-11-10 matplotlib/__init__.py: Added FROZEN to signal we are running under

py2exe (or similar) - is used by backend_gtk.py - SC

2004-11-09 backend_gtk.py: Made fix suggested by maffew@cat.org.au

to prevent problems when py2exe calls pygtk.require(). - SC

2004-11-09 backend_cairo.py: Added support for printing to a fileobject.

Disabled cairo PS output which is not working correctly. - SC

2004-11-08 matplotlib-0.64 released

2004-11-04 Changed -dbackend processing to only use known backends, so

we don't clobber other non-matplotlib uses of -d, like -debug.

2004-11-04 backend_agg.py: added IMAGE_FORMAT to list the formats that the

backend can save to. backend_gtkagg.py: added support for saving JPG files by using the GTK backend - SC

2004-10-31 backend_cairo.py: now produces png and ps files (although the figure

sizing needs some work). pycairo did not wrap all the necessary functions, so I wrapped them myself, they are included in the backend_cairo.py doc string. - SC

2004-10-31 backend_ps.py: clean up the generated PostScript code, use

the PostScript stack to hold itermediate values instead of storing them in the dictionary. - JV

2004-10-30 backend_ps.py, ft2font.cpp, ft2font.h: fix the position of

text in the PostScript output. The new FT2Font method get_descent gives the distance between the lower edge of the bounding box and the baseline of a string. In backend_ps the text is shifted upwards by this amount. - JV

2004-10-30 backend_ps.py: clean up the code a lot. Change the

PostScript output to be more DSC compliant. All definitions for the generated PostScript are now in a PostScript dictionary 'mpldict'. Moved the long comment about drawing ellipses from the PostScript output into a Python comment. - JV

2004-10-30 backend_gtk.py: removed FigureCanvasGTK.realize() as its no longer

needed. Merged ColorManager into GraphicsContext backend_bases.py: For set_capstyle/joinstyle() only set cap or joinstyle if there is no error. - SC

2004-10-30 backend_gtk.py: tidied up print_figure() and removed some of the

dependency on widget events - SC

2004-10-28 backend_cairo.py: The renderer is complete except for mathtext,

draw_image() and clipping. gtkcairo works reasonably well. cairo does not yet create any files since I can't figure how to set the 'target surface', I don't think pycairo wraps the required functions - SC

2004-10-28 backend_gtk.py: Improved the save dialog (GTK 2.4 only) so it

presents the user with a menu of supported image formats - SC

2004-10-28 backend_svg.py: change print_figure() to restore original face/edge

color backend_ps.py : change print_figure() to ensure original face/edge colors are restored even if there's an IOError - SC

2004-10-27 Applied Norbert's errorbar patch to support barsabove kwarg

2004-10-27 Applied Norbert's legend patch to support None handles

2004-10-27 Added two more backends: backend_cairo.py, backend_gtkcairo.py

They are not complete yet, currently backend_gtkcairo just renders polygons, rectangles and lines - SC

2004-10-21 Added polar axes and plots - JDH

2004-10-20 Fixed corrcoef bug exposed by corrcoef(X) where X is matrix

- JDH

2004-10-19 Added kwarg support to xticks and yticks to set ticklabel

text properties -- thanks to T. Edward Whalen for the suggestion

2004-10-19 Added support for PIL images in imshow(), image.py - ADS

2004-10-19 Re-worked exception handling in _image.py and _transforms.py

to avoid masking problems with shared libraries. - JTM

2004-10-16 Streamlined the matlab interface wrapper, removed the

noplot option to hist - just use mlab.hist instead.

2004-09-30 Added Andrew Dalke's strftime code to extend the range of

dates supported by the DateFormatter - JDH

2004-09-30 Added barh - JDH

2004-09-30 Removed fallback to alternate array package from numerix

so that ImportErrors are easier to debug. JTM

2004-09-30 Add GTK+ 2.4 support for the message in the toolbar. SC

2004-09-30 Made some changes to support python22 - lots of doc

fixes. - JDH

2004-09-29 Added a Verbose class for reporting - JDH

2004-09-28 Released 0.63.0

2004-09-28 Added save to file object for agg - see

examples/print_stdout.py

2004-09-24 Reorganized all py code to lib subdir

2004-09-24 Fixed axes resize image edge effects on interpolation -

required upgrade to agg22 which fixed an agg bug related to this problem

2004-09-20 Added toolbar2 message display for backend_tkagg. JTM

2004-09-17 Added coords formatter attributes. These must be callable,

and return a string for the x or y data. These will be used to format the x and y data for the coords box. Default is the axis major formatter. e.g.:

format the coords message box def price(x): return '$%1.2f'%x ax.format_xdata = DateFormatter('%Y-%m-%d') ax.format_ydata = price

2004-09-17 Total rewrite of dates handling to use python datetime with

num2date, date2num and drange. pytz for timezone handling, dateutils for sophisticated ticking. date ranges from 0001-9999 are supported. rrules allow arbitrary date ticking. examples/date_demo*.py converted to show new usage. new example examples/date_demo_rrule.py shows how to use rrules in date plots. The date locators are much more general and almost all of them have different constructors. See matplotlib.dates for more info.

2004-09-15 Applied Fernando's backend __init__ patch to support easier

backend maintenance. Added his numutils to mlab. JDH

2004-09-16 Re-designated all files in matplotlib/images as binary and

w/o keyword substitution using "cvs admin -kb *.svg ...". See binary files in "info cvs" under Linux. This was messing up builds from CVS on windows since CVS was doing lf -> cr/lf and keyword substitution on the bitmaps. - JTM

2004-09-15 Modified setup to build array-package-specific extensions

for those extensions which are array-aware. Setup builds extensions automatically for either Numeric, numarray, or both, depending on what you have installed. Python proxy modules for the array-aware extensions import the version optimized for numarray or Numeric determined by numerix. - JTM

2004-09-15 Moved definitions of infinity from mlab to numerix to avoid

divide by zero warnings for numarray - JTM

2004-09-09 Added axhline, axvline, axhspan and axvspan

2004-08-30 matplotlib 0.62.4 released

2004-08-30 Fixed a multiple images with different extent bug,

Fixed markerfacecolor as RGB tuple

2004-08-27 Mathtext now more than 5x faster. Thanks to Paul Mcguire

for fixes both to pyparsing and to the matplotlib grammar! mathtext broken on python2.2

2004-08-25 Exposed Darren's and Greg's log ticking and formatting

options to semilogx and friends

2004-08-23 Fixed grid w/o args to toggle grid state - JDH

2004-08-11 Added Gregory's log patches for major and minor ticking

2004-08-18 Some pixel edge effects fixes for images

2004-08-18 Fixed TTF files reads in backend_ps on win32.

2004-08-18 Added base and subs properties for logscale plots, user

modifiable using set_[x,y]scale('log',base=b,subs=[mt1,mt2,...]) - GL

2004-08-18 fixed a bug exposed by trying to find the HOME dir on win32

thanks to Alan Issac for pointing to the light - JDH

2004-08-18 fixed errorbar bug in setting ecolor - JDH

2004-08-12 Added Darren Dale's exponential ticking patch

2004-08-11 Added Gregory's fltkagg backend

2004-08-09 matplotlib-0.61.0 released

2004-08-08 backend_gtk.py: get rid of the final PyGTK deprecation warning by

replacing gtkOptionMenu with gtkMenu in the 2.4 version of the classic toolbar.

2004-08-06 Added Tk zoom to rect rectangle, proper idle drawing, and

keybinding - JDH

2004-08-05 Updated installing.html and INSTALL - JDH

2004-08-01 backend_gtk.py: move all drawing code into the expose_event()

2004-07-28 Added Greg's toolbar2 and backend_*agg patches - JDH

2004-07-28 Added image.imread with support for loading png into

numerix arrays

2004-07-28 Added key modifiers to events - implemented dynamic updates

and rubber banding for interactive pan/zoom - JDH

2004-07-27 did a readthrough of SVG, replacing all the string

additions with string interps for efficiency, fixed some layout problems, added font and image support (through external pngs) - JDH

2004-07-25 backend_gtk.py: modify toolbar2 to make it easier to support GTK+

2.4. Add GTK+ 2.4 toolbar support. - SC

2004-07-24 backend_gtk.py: Simplified classic toolbar creation - SC

2004-07-24 Added images/matplotlib.svg to be used when GTK+ windows are

minimised - SC

2004-07-22 Added right mouse click zoom for NavigationToolbar2 panning

mode. - JTM

2004-07-22 Added NavigationToolbar2 support to backend_tkagg.

Minor tweak to backend_bases. - JTM

2004-07-22 Incorporated Gergory's renderer cache and buffer object

cache - JDH

2004-07-22 Backend_gtk.py: Added support for GtkFileChooser, changed

FileSelection/FileChooser so that only one instance pops up, and made them both modal. - SC

2004-07-21 Applied backend_agg memory leak patch from hayden -

jocallo@online.no. Found and fixed a leak in binary operations on transforms. Moral of the story: never incref where you meant to decref! Fixed several leaks in ft2font: moral of story: almost always return Py::asObject over Py::Object - JDH

2004-07-21 Fixed a to string memory allocation bug in agg and image

modules - JDH

2004-07-21 Added mpl_connect and mpl_disconnect to matlab interface -

JDH

2004-07-21 Added beginnings of users_guide to CVS - JDH

2004-07-20 ported toolbar2 to wx

2004-07-20 upgraded to agg21 - JDH

2004-07-20 Added new icons for toolbar2 - JDH

2004-07-19 Added vertical mathtext for *Agg and GTK - thanks Jim

Benson! - JDH

2004-07-16 Added ps/eps/svg savefig options to wx and gtk JDH

2004-07-15 Fixed python framework tk finder in setupext.py - JDH

2004-07-14 Fixed layer images demo which was broken by the 07/12 image

extent fixes - JDH

2004-07-13 Modified line collections to handle arbitrary length

segments for each line segment. - JDH

2004-07-13 Fixed problems with image extent and origin -

set_image_extent deprecated. Use imshow(blah, blah, extent=(xmin, xmax, ymin, ymax) instead - JDH

2004-07-12 Added prototype for new nav bar with codifed event

handling. Use mpl_connect rather than connect for matplotlib event handling. toolbar style determined by rc toolbar param. backend status: gtk: prototype, wx: in progress, tk: not started - JDH

2004-07-11 backend_gtk.py: use builtin round() instead of redefining it.

- SC

2004-07-10 Added embedding_in_wx3 example - ADS

2004-07-09 Added dynamic_image_wxagg to examples - ADS

2004-07-09 added support for embedding TrueType fonts in PS files - PEB

2004-07-09 fixed a sfnt bug exposed if font cache is not built

2004-07-09 added default arg None to matplotlib.matlab grid command to

toggle current grid state

2004-07-08 0.60.2 released

2004-07-08 fixed a mathtext bug for '6'

2004-07-08 added some numarray bug workarounds

2004-07-07 0.60 released

2004-07-07 Fixed a bug in dynamic_demo_wx

2004-07-07 backend_gtk.py: raise SystemExit immediately if

'import pygtk' fails - SC

2004-07-05 Added new mathtext commands over{sym1}{sym2} and

under{sym1}{sym2}

2004-07-05 Unified image and patch collections colormapping and

scaling args. Updated docstrings for all - JDH

2004-07-05 Fixed a figure legend bug and added

examples/figlegend_demo.py - JDH

2004-07-01 Fixed a memory leak in image and agg to string methods

2004-06-25 Fixed fonts_demo spacing problems and added a kwargs

version of the fonts_demo fonts_demo_kw.py - JDH

2004-06-25 finance.py: handle case when urlopen() fails - SC

2004-06-24 Support for multiple images on axes and figure, with

blending. Support for upper and lower image origins. clim, jet and gray functions in matlab interface operate on current image - JDH

2004-06-23 ported code to Perry's new colormap and norm scheme. Added

new rc attributes image.aspect, image.interpolation, image.cmap, image.lut, image.origin

2004-06-20 backend_gtk.py: replace gtk.TRUE/FALSE with True/False.

simplified _make_axis_menu(). - SC

2004-06-19 anim_tk.py: Updated to use TkAgg by default (not GTK)

backend_gtk_py: Added '_' in front of private widget creation functions - SC

2004-06-17 backend_gtk.py: Create a GC once in realise(), not every

time draw() is called. - SC

2004-06-16 Added new py2exe FAQ entry and added frozen support in

get_data_path for py2exe - JDH

2004-06-16 Removed GTKGD, which was always just a proof-of-concept

backend - JDH

2004-06-16 backend_gtk.py updates to replace deprecated functions

gtk.mainquit(), gtk.mainloop().

Update NavigationToolbar to use the new GtkToolbar API - SC

2004-06-15 removed set_default_font from font_manager to unify font

customization using the new function rc. See API_CHANGES for more info. The examples fonts_demo.py and fonts_demo_kw.py are ported to the new API - JDH

2004-06-15 Improved (yet again!) axis scaling to properly handle

singleton plots - JDH

2004-06-15 Restored the old FigureCanvasGTK.draw() - SC

2004-06-11 More memory leak fixes in transforms and ft2font - JDH

2004-06-11 Eliminated numerix .numerix file and environment variable

NUMERIX. Fixed bug which prevented command line overrides: --numarray or --numeric. - JTM

2004-06-10 Added rc configuration function rc; deferred all rc param

setting until object creation time; added new rc attrs: lines.markerfacecolor, lines.markeredgecolor, lines.markeredgewidth, patch.linewidth, patch.facecolor, patch.edgecolor, patch.antialiased; see examples/customize_rc.py for usage - JDH

2004-06-09 0.54.2 released

2004-06-08 Rewrote ft2font using CXX as part of general memory leak

fixes; also fixed transform memory leaks - JDH

2004-06-07 Fixed several problems with log ticks and scaling - JDH

2004-06-07 Fixed width/height issues for images - JDH

2004-06-03 Fixed draw_if_interactive bug for semilogx;

2004-06-02 Fixed text clipping to clip to axes - JDH

2004-06-02 Fixed leading newline text and multiple newline text - JDH

2004-06-02 Fixed plot_date to return lines - JDH

2004-06-01 Fixed plot to work with x or y having shape N,1 or 1,N - JDH

2004-05-31 Added renderer markeredgewidth attribute of Line2D. - ADS

2004-05-29 Fixed tick label clipping to work with navigation.

2004-05-28 Added renderer grouping commands to support groups in

SVG/PS. - JDH

2004-05-28 Fixed, this time I really mean it, the singleton plot

plot([0]) scaling bug; Fixed Flavio's shape = N,1 bug - JDH

2004-05-28 added colorbar - JDH

2004-05-28 Made some changes to the matplotlib.colors.Colormap to

properly support clim - JDH

2004-05-27 0.54.1 released

2004-05-27 Lots of small bug fixes: rotated text at negative angles,

errorbar capsize and autoscaling, right tick label position, gtkagg on win98, alpha of figure background, singleton plots - JDH

2004-05-26 Added Gary's errorbar stuff and made some fixes for length

one plots and constant data plots - JDH

2004-05-25 Tweaked TkAgg backend so that canvas.draw() works

more like the other backends. Fixed a bug resulting in 2 draws per figure manager show(). - JTM

2004-05-19 0.54 released

2004-05-18 Added newline separated text with rotations to text.Text

layout - JDH

2004-05-16 Added fast pcolor using PolyCollections. - JDH

2004-05-14 Added fast polygon collections - changed scatter to use

them. Added multiple symbols to scatter. 10x speedup on large scatters using *Agg and 5X speedup for ps. - JDH

2004-05-14 On second thought... created an "nx" namespace in

in numerix which maps type names onto typecodes the same way for both numarray and Numeric. This undoes my previous change immediately below. To get a typename for Int16 usable in a Numeric extension: say nx.Int16. - JTM

2004-05-15 Rewrote transformation class in extension code, simplified

all the artist constructors - JDH

2004-05-14 Modified the type definitions in the numarray side of

numerix so that they are Numeric typecodes and can be used with Numeric compilex extensions. The original numarray types were renamed to type<old_name>. - JTM

2004-05-06 Gary Ruben sent me a bevy of new plot symbols and markers.

See matplotlib.matlab.plot - JDH

2004-05-06 Total rewrite of mathtext - factored ft2font stuff out of

layout engine and defined abstract class for font handling to lay groundwork for ps mathtext. Rewrote parser and made layout engine much more precise. Fixed all the layout hacks. Added spacing commands / and hspace. Added composite chars and defined angstrom. - JDH

2004-05-05 Refactored text instances out of backend; aligned

text with arbitrary rotations is now supported - JDH

2004-05-05 Added a Matrix capability for numarray to numerix. JTM

2004-05-04 Updated whats_new.html.template to use dictionary and

template loop, added anchors for all versions and items; updated goals.txt to use those for links. PG

2004-05-04 Added fonts_demo.py to backend_driver, and AFM and TTF font

caches to font_manager.py - PEB

2004-05-03 Redid goals.html.template to use a goals.txt file that

has a pseudo restructured text organization. PG

2004-05-03 Removed the close buttons on all GUIs and added the python

#! bang line to the examples following Steve Chaplin's advice on matplotlib dev

2004-04-29 Added CXX and rewrote backend_agg using it; tracked down

and fixed agg memory leak - JDH

2004-04-29 Added stem plot command - JDH

2004-04-28 Fixed PS scaling and centering bug - JDH

2004-04-26 Fixed errorbar autoscale problem - JDH

2004-04-22 Fixed copy tick attribute bug, fixed singular datalim

ticker bug; fixed mathtext fontsize interactive bug. - JDH

2004-04-21 Added calls to draw_if_interactive to axes(), legend(),

and pcolor(). Deleted duplicate pcolor(). - JTM

2004-04-21 matplotlib 0.53 release

2004-04-19 Fixed vertical alignment bug in PS backend - JDH

2004-04-17 Added support for two scales on the "same axes" with tick

different ticking and labeling left right or top bottom. See examples/two_scales.py - JDH

2004-04-17 Added default dirs as list rather than single dir in

setupext.py - JDH

2004-04-16 Fixed wx exception swallowing bug (and there was much

rejoicing!) - JDH

2004-04-16 Added new ticker locator a formatter, fixed default font

return - JDH

2004-04-16 Added get_name method to FontProperties class. Fixed font lookup

in GTK and WX backends. - PEB

2004-04-16 Added get- and set_fontstyle msethods. - PEB

2004-04-10 Mathtext fixes: scaling with dpi, - JDH

2004-04-09 Improved font detection algorithm. - PEB

2004-04-09 Move deprecation warnings from text.py to __init__.py - PEB

2004-04-09 Added default font customization - JDH

2004-04-08 Fixed viewlim set problem on axes and axis. - JDH

2004-04-07 Added validate_comma_sep_str and font properties parameters to

__init__. Removed font families and added rcParams to FontProperties __init__ arguments in font_manager. Added default font property parameters to .matplotlibrc file with descriptions. Added deprecation warnings to the get_ - and set_fontXXX methods of the Text object. - PEB

2004-04-06 Added load and save commands for ASCII data - JDH

2004-04-05 Improved font caching by not reading AFM fonts until needed.

Added better documentation. Changed the behaviour of the get_family, set_family, and set_name methods of FontProperties. - PEB

2004-04-05 Added WXAgg backend - JDH

2004-04-04 Improved font caching in backend_agg with changes to

font_manager - JDH

2004-03-29 Fixed fontdicts and kwargs to work with new font manager -

JDH

This is the Old, stale, never used changelog

2002-12-10 - Added a TODO file and CHANGELOG. Lots to do -- get

crackin'!

- Fixed y zoom tool bug
- Adopted a compromise fix for the y data clipping problem. The problem was that for solid lines, the y data clipping (as opposed to the gc clipping) caused artifactual horizontal solid lines near the ylim boundaries. I did a 5% offset hack in Axes set_ylim functions which helped, but didn't cure the problem for very high gain y zooms. So I disabled y data clipping for connected lines . If you need extensive y clipping, either plot(y,x) because x data clipping is always enabled, or change the _set_clip code to 'if 1' as indicated in the lines.py src. See _set_clip in lines.py and set_ylim in figure.py for more information.

2002-12-11 - Added a measurement dialog to the figure window to

measure axes position and the delta x delta y with a left mouse drag. These defaults can be overridden by deriving from Figure and overriding button_press_event, button_release_event, and motion_notify_event, and _dialog_measure_tool.

- fixed the navigation dialog so you can check the axes the navigation buttons apply to.

2003-04-23 Released matplotlib v0.1

2003-04-24 Added a new line style PixelLine2D which is the plots the

markers as pixels (as small as possible) with format symbol ','

Added a new class Patch with derived classes Rectangle, RegularPolygon and Circle

2003-04-25 Implemented new functions errorbar, scatter and hist

Added a new line type '|' which is a vline. syntax is plot(x, Y, '|') where y.shape = len(x),2 and each row gives the ymin,ymax for the respective values of x. Previously I had implemented vlines as a list of lines, but I needed the efficientcy of the numeric clipping for large numbers of vlines outside the viewport, so I wrote a dedicated class Vline2D which derives from Line2D

2003-05-01

Fixed ytick bug where grid and tick show outside axis viewport with gc clip

2003-05-14

Added new ways to specify colors 1) matlab format string 2) html-style hex string, 3) rgb tuple. See examples/color_demo.py

2003-05-28

Changed figure rendering to draw form a pixmap to reduce flicker. See examples/system_monitor.py for an example where the plot is continusouly updated w/o flicker. This example is meant to simulate a system monitor that shows free CPU, RAM, etc...

2003-08-04

Added Jon Anderson's GTK shell, which doesn't require pygtk to have threading built-in and looks nice!

2003-08-25

Fixed deprecation warnings for python2.3 and pygtk-1.99.18

2003-08-26

Added figure text with new example examples/figtext.py

2003-08-27

Fixed bugs i figure text with font override dictionairies and fig text that was placed outside the window bounding box

2003-09-1 through 2003-09-15

Added a postscript and a GD module backend

2003-09-16

Fixed font scaling and point scaling so circles, squares, etc on lines will scale with DPI as will fonts. Font scaling is not fully implemented on the gtk backend because I have not figured out how to scale fonts to arbitrary sizes with GTK

2003-09-17

Fixed figure text bug which crashed X windows on long figure text extending beyond display area. This was, I believe, due to the vestigial erase functionality that was no longer needed since I began rendering to a pixmap

2003-09-30 Added legend

2003-10-01 Fixed bug when colors are specified with rgb tuple or hex

string.

2003-10-21 Andrew Straw provided some legend code which I modified

and incorporated. Thanks Andrew!

2003-10-27 Fixed a bug in axis.get_view_distance that affected zoom in

versus out with interactive scrolling, and a bug in the axis text reset system that prevented the text from being redrawn on a interactive gtk view lim set with the widget

Fixed a bug in that prevented the manual setting of ticklabel strings from working properly

2003-11-02 - Do a nearest neighbor color pick on GD when

allocate fails

2003-11-02

- Added pcolor plot
- Added MRI example
- Fixed bug that screwed up label position if xticks or yticks were empty
- added nearest neighbor color picker when GD max colors exceeded
- fixed figure background color bug in GD backend

2003-11-10 - 2003-11-11

- major refactoring.
  - Ticks (with labels, lines and grid) handled by dedicated class
  - Artist now know bounding box and dpi
  - Bounding boxes and transforms handled by dedicated classes
  - legend in dedicated class. Does a better job of alignment and bordering. Can be initialized with specific line instances. See examples/legend_demo2.py

2003-11-14 Fixed legend positioning bug and added new position args

2003-11-16 Finished porting GD to new axes API

2003-11-20 - add TM for matlab on website and in docs

2003-11-20 - make a nice errorbar and scatter screenshot

2003-11-20 - auto line style cycling for multiple line types

broken

2003-11-18 (using inkrect) :logical rect too big on gtk backend

2003-11-18 ticks don't reach edge of axes in gtk mode --

rounding error?

2003-11-20 - port Gary's errorbar code to new API before 0.40

2003-11-20 - problem with stale _set_font. legend axes box

doesn't resize on save in GTK backend -- see htdocs legend_demo.py

2003-11-21 - make a dash-dot dict for the GC

2003-12-15 - fix install path bug

CHAPTER

NINE

LICENSE

Matplotlib only uses BSD compatible code, and its license is based on the PSF license. See the Open Source Initiative licenses page for details on individual licenses. Non-BSD compatible licenses (e.g., LGPL) are acceptable in matplotlib toolkits. For a discussion of the motivations behind the licencing choice, see *Licenses*.

9.1 Copyright Policy

John Hunter began matplotlib around 2003. Since shortly before his passing in 2012, Michael Droettboom has been the lead maintainer of matplotlib, but, as has always been the case, matplotlib is the work of many.

Prior to July of 2013, and the 1.3.0 release, the copyright of the source code was held by John Hunter. As of July 2013, and the 1.3.0 release, matplotlib has moved to a shared copyright model.

matplotlib uses a shared copyright model. Each contributor maintains copyright over their contributions to matplotlib. But, it is important to note that these contributions are typically only changes to the repositories. Thus, the matplotlib source code, in its entirety, is not the copyright of any single person or institution. Instead, it is the collective copyright of the entire matplotlib Development Team. If individual contributors want to maintain a record of what changes/contributions they have specific copyright on, they should indicate their copyright in the commit message of the change, when they commit the change to one of the matplotlib repositories.

The Matplotlib Development Team is the set of all contributors to the matplotlib project. A full list can be obtained from the git version control logs.

9.2 License agreement for matplotlib 3.3.3

1. This LICENSE AGREEMENT is between the Matplotlib Development Team ("MDT"), and the Individual or Organization ("Licensee") accessing and otherwise using matplotlib software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, MDT hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use matplotlib 3.3.3 alone or in any derivative version, provided, however, that MDT's License Agreement and MDT's notice of copyright, i.e., "Copyright (c) 2012-2013 Matplotlib Development Team; All Rights Reserved" are retained in matplotlib 3.3.3 alone or in any derivative version prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or incorporates matplotlib 3.3.3 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to matplotlib 3.3.3.

4. MDT is making matplotlib 3.3.3 available to Licensee on an "AS IS" basis. MDT MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, MDT MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF MATPLOTLIB 3.3.3 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. MDT SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF MATPLOTLIB 3.3.3 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING MATPLOTLIB 3.3.3, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between MDT and Licensee. This License Agreement does not grant permission to use MDT trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.

8. By copying, installing or otherwise using matplotlib 3.3.3, Licensee agrees to be bound by the terms and conditions of this License Agreement.

9.3 License agreement for matplotlib versions prior to 1.3.0

1. This LICENSE AGREEMENT is between John D. Hunter ("JDH"), and the Individual or Organization ("Licensee") accessing and otherwise using matplotlib software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, JDH hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use matplotlib 3.3.3 alone or in any derivative version, provided, however, that JDH's License Agreement and JDH's notice of copyright, i.e., "Copyright (c) 2002-2009 John D. Hunter; All Rights Reserved" are retained in matplotlib 3.3.3 alone or in any derivative version prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or incorporates matplotlib 3.3.3 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to matplotlib 3.3.3.

4. JDH is making matplotlib 3.3.3 available to Licensee on an "AS IS" basis. JDH MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, JDH MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF MATPLOTLIB 3.3.3 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. JDH SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF MATPLOTLIB 3.3.3 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING MATPLOTLIB 3.3.3, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between JDH and Licensee. This License Agreement does not grant permission to use JDH trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.

8. By copying, installing or otherwise using matplotlib 3.3.3, Licensee agrees to be bound by the terms and conditions of this License Agreement.

CHAPTER

TEN

CITING MATPLOTLIB

If Matplotlib contributes to a project that leads to a scientific publication, please acknowledge this fact by citing J. D. Hunter, "Matplotlib: A 2D Graphics Environment", Computing in Science & Engineering, vol. 9, no. 3, pp. 90-95, 2007.

```
@Article{Hunter:2007,
  Author    = {Hunter, J. D.},
  Title     = {Matplotlib: A 2D graphics environment},
  Journal   = {Computing in Science \& Engineering},
  Volume    = {9},
  Number    = {3},
  Pages     = {90--95},
  abstract  = {Matplotlib is a 2D graphics package used for Python for
  application development, interactive scripting, and publication-quality
  image generation across user interfaces and operating systems.},
  publisher = {IEEE COMPUTER SOC},
  doi       = {10.1109/MCSE.2007.55},
  year      = 2007
}
```

10.1 DOIs

The following DOI represents *all* Matplotlib versions. Please select a more specific DOI from the list below, referring to the version used for your publication.

10.1.1 By version

v3.3.3

v3.3.2

v3.3.1

DOI 10.5281/zenodo.3984190

v3.3.0

DOI 10.5281/zenodo.3948793

v3.2.2

DOI 10.5281/zenodo.3898017

v3.2.1

DOI 10.5281/zenodo.3714460

v3.2.0

DOI 10.5281/zenodo.3695547

v3.1.3

DOI 10.5281/zenodo.3633844

v3.1.2

DOI 10.5281/zenodo.3563226

v3.1.1

DOI 10.5281/zenodo.3264781

v3.1.0

DOI 10.5281/zenodo.2893252

v3.0.3

DOI 10.5281/zenodo.2577644

v3.0.2

DOI 10.5281/zenodo.1482099

v3.0.1

DOI 10.5281/zenodo.1482098

v2.2.5

DOI 10.5281/zenodo.3633833

v3.0.0

DOI 10.5281/zenodo.1420605

v2.2.4

DOI 10.5281/zenodo.2669103

v2.2.3

DOI 10.5281/zenodo.1343133

v2.2.2

DOI 10.5281/zenodo.1202077

v2.2.1

DOI 10.5281/zenodo.1202050

v2.2.0

DOI 10.5281/zenodo.1189358

v2.1.2

DOI 10.5281/zenodo.1154287

v2.1.1

DOI 10.5281/zenodo.1098480

v2.1.0

DOI 10.5281/zenodo.1004650

v2.0.2

DOI 10.5281/zenodo.573577

v2.0.1

DOI 10.5281/zenodo.570311

v2.0.0

DOI 10.5281/zenodo.248351

v1.5.3

DOI 10.5281/zenodo.61948

v1.5.2

DOI 10.5281/zenodo.56926

v1.5.1

DOI 10.5281/zenodo.44579

v1.5.0

DOI 10.5281/zenodo.32914

v1.4.3

DOI 10.5281/zenodo.15423

v1.4.2

DOI 10.5281/zenodo.12400

v1.4.1

DOI 10.5281/zenodo.12287

v1.4.0

DOI 10.5281/zenodo.11451

CHAPTER

ELEVEN

CREDITS

Matplotlib was written by John D. Hunter, with contributions from an ever-increasing number of users and developers. The current lead developer is Thomas A. Caswell, who is assisted by many active developers. Please also see our instructions on *Citing Matplotlib*.

The following is a list of contributors extracted from the git revision control history of the project:

4over7, 816-8055, Aaron Boushley, Aashil Patel, AbdealiJK, Abhinav Sagar, Abhinuv Nitin Pitale, Acanthostega, Adam, Adam Ginsburg, Adam Gomaa, Adam Heck, Adam J. Stewart, Adam Ortiz, Adam Paszke, Adam Ruszkowski, Adam Williamson, Adrian Price-Whelan, Adrien Chardon, Adrien F. Vincent, Ahmet Bakan, Akshay Nair, Alan Bernstein, Alan Du, Alberto, Alejandro Dubrovsky, Aleksey Bilogur, Alex C. Szatmary, Alex Loew, Alex Rothberg, Alex Rudy, AlexCav, Alexander Buchkovsky, Alexander Harnisch, Alexander Rudy, Alexander Taylor, Alexei Colin, Alexis Bienvenüe, Ali Mehdi, Ali Uneri, Alistair Muldal, Allan Haldane, Allen Downey, Alon Hershenhorn, Alvaro Sanchez, Amit Aronovitch, Amy, Amy Roberts, AmyTeegarden, AndersonDaniel, Andras Deak, Andrea Bedini, Andreas Gustafsson, Andreas Hilboll, Andreas Mayer, Andreas Mueller, Andreas Wallner, Andrew Dawson, Andrew Merrill, Andrew Nelson, Andrew Straw, Andy Mastbaum, Andy Zhu, Ankur Dedania, Anthony Scopatz, Anton Akhmerov, Antony Lee, Anubhav Shrimal, Ao Liu (frankliuao), Ardie Orden, Arie, Ariel Hernán Curiale, Arnaud Gardelein, Arpad Horvath, Arthur Paulino, Arvind, Aseem Bansal, Ashley Whetter, Atharva Khare, Avinash Sharma, Ayappan P, BHT, BTWS, Bas van Schaik, Bastian Bechtold, Behram Mistree, Ben, Ben Cohen, Ben Gamari, Ben Keller, Ben Root, Benedikt Daurer, Benjamin Bengfort, Benjamin Berg, Benjamin Congdon, Benjamin Reedlunn, Bernhard M. Wiedemann, Bharat123rox, Bianca Gibson, Binglin Chang, Bingyao Liu, Björn Dahlgren, Blaise Thompson, Boaz Mohar, Bradley M. Froehle, Brandon Liu, Brendan Zhang, Brennan Magee, Brett Cannon, Brett Graham, Brian Mattern, Brian McLaughlin, Brigitta Sipocz, Bruno Beltran, Bruno Pagani, Bruno Zohreh, CJ Carey, Cameron Bates, Cameron Davidson-Pilon, Cameron Fackler, Carissa Brittain, Carl Michal, Carsten Schelp, Carwyn Pelley, Casey Webster, Casper van der Wel, Charles Moad, Charles Ruan, Chen Karako, Cho Yin Yong, Chris, Chris Barnes, Chris Beaumont, Chris G, Chris Holdgraf, Chris Zimmerman, Christer Jensen, Christian Brodbeck, Christian Brueffer, Christian Stade-Schuldt, Christoph Dann, Christoph Deil, Christoph Gohlke, Christoph Hoffmann, Christoph Pohl, Christoph Reiter, Christopher Bradshaw, Cimarron Mittelsteadt, Clemens Brunner, Cody Scot, Colin, Colin Carroll, Cong Ma, Conner R. Phillips, Corey

Farwell, Craig Citro, Craig M, Craig Tenney, DaCoEx, Dakota Blair, Damian, Damon McDougall, Dan Hickstein, Dana, Daniel C. Marcu, Daniel Hyams, Daniel Laidig, Daniel O'Connor, DanielMatu, Daniele Nicolodi, Danny Hermes, Dara Adib, Darren Dale, DaveL17, David A, David Anderson, David Chudzicki, David Haberthür, David Huard, David Kaplan, David Kent, David Kua, David Stansby, David Trémouilles, Dean Malmgren, Deng Tian, Derek Kim, Derek Tropf, Devashish Deshpande, Diego Mora Cespedes, Dietmar Schwertberger, Dietrich Brunn, Divyam Madaan, Dmitry Lupyan, Dmitry Mottl, Dmitry Shachnev, Dominik Schmidt, DonaldSeo, Dora Fraeman Caswell, DoriekeMG, Dorota Jarecka, Doug Blank, Drew J. Sonne, Duncan Macleod, Dylan Evans, E. G. Patrick Bos, Edin Salkovic, Edoardo Pizzigoni, Egor Panfilov, Elan Ernest, Elena Glassman, Elias Pipping, Elijah Schutz, Elizabeth Seiver, Elliott Sales de Andrade, Elvis Stansvik, Emil Mikulic, Emlyn Price, Eric Dill, Eric Firing, Eric Larson, Eric Ma, Eric O. LEBIGOT (EOL), Eric Relson, Eric Wieser, Erik Bray, Erik M. Bray, Erin Pintozzi, Eugen Beck, Eugene Yurtsev, Evan Davey, Ezra Peisach, Fabian Kloosterman, Fabian-Robert Stöter, Fabien Maussion, Fabio Zanini, FedeMiorelli, Federico Ariza, Felipe, Felix, Felix Kohlgrüber, Felix Yan, Fernando Perez, Filip Dimitrovski, Filipe Fernandes, Florencia Noriega, Florian Le Bourdais, Florian Rhiem, Francesco Montesano, Francis Colas, Franco Vaccari, Francoise Provencher, Frank Sauerburger, Frank Yu, François Magimel, Gabe, Gabriel Munteanu, Gal Avineri, Galen Lynch, Gauravjeet, Gaute Hope, Gazing, Gellule Xg, Geoffrey Spear, Geoffroy Billotey, Georg Raiser, Gerald Storer, Gina, Giovanni, Graeme Smecher, Graham Poulter, Greg Lucas, Gregory Ashton, Gregory R. Lee, Grillard, Grégory Lielens, Guillaume Gay, Guillermo Breto, Gustavo Braganca, Gustavo Goretkin, HHest, Hajoon Choi, Hakan Kucukdereli, Hanno Rein, Hans Dembinski, Hans Meine, Hans Moritz Günther, Harnesser, Harshal Prakash Patankar, Harshit Patni, Hassan Kibirige, Hastings Greer, Heath Henley, Heiko Oberdiek, Helder, Henning Pohl, Herbert Kruitbosch, Holger Peters, Hubert Holin, Hugo van Kemenade, Ian Hincks, Ian Thomas, Ida Hjorth, Ignas Anikevicius (gns_ank), Ildar Akhmetgaleev, Ilia Kurenkov, Ilya Flyamer, ImSoErgodic, ImportanceOfBeingErnest, Inception95, Ingo Fründ, Ioannis Filippidis, Isa Hassen, Isaac Schwabacher, Isaac Slavitt, Ismo Toijala, J Alammar, J. Goutin, Jaap Versteegh, Jack Kelly, Jacob McDonald, Jacobson Okoro, Jae-Joon Lee, Jaime Fernandez, Jake Lee, Jake Vanderplas, James A. Bednar, James Adams, James Pallister, James R. Evans, JamesMakela, Jamie Nunez, Jan S. (Milania1), Jan Schlüter, Jan Schulz, Jan-Philip Gehrcke, Jan-willem De Bleser, Jarrod Millman, Jascha Ulrich, Jason Grout, Jason King, Jason Liw Yan Chong, Jason Miller, Jason Neal, Jason Zheng, Javad, JayP16, Jean-Benoist Leger, Jeff Lutgen, Jeff Whitaker, Jeffrey Bingham, Jeffrey Hokanson @ Loki, JelsB, Jens Hedegaard Nielsen, Jeremy Fix, Jeremy O'Donoghue, Jeremy Thurgood, Jeroonk, Jessica B. Hamrick, Jiahao Chen, Jim Radford, Jochen Voss, Jody Klymak, Joe Kington, Joel B. Mohler, Joel Frederico, Joel Wanner, Johannes H. Jensen, Johannes Wienke, John Hoffman, John Hunter, John Vandenberg, Johnny Gill, JojoBoulix, Jon Haitz Legarreta Gorroño, Jonas Camillus Jeppesen, Jonathan Waltman, Jorge Moraleda, Jorrit Wronski, Joscha Reimer, Josef Heinen, Joseph Albert, Joseph Fox-Rabinovitz, Joseph Jon Booker, Joseph Martinot-Lagarde, Joshua Taillon, José Ricardo, Jouni K. Seppänen, Joy Bhalla, Juan Nunez-Iglesias, Juanjo Bazán, Julia Sprenger, Julian Mehne, Julian Taylor, Julian V. Modesto, JulianCienfuegos, Julien Lhermitte, Julien Schueller, Julien Woillez, Julien-Charles Lévesque, Jun Tan, Justin Cai, Jörg Dietrich, Kacper Kowalik (Xarthisius), Kai Muehlbauer, Kanchana Ranasinghe, Kanwar245, Katrin Lein-

weber, Katy Huff, Kayla Ngan, Keerysanth Sribaskaran, Ken McIvor, Kenneth Ma, Kevin Chan, Kevin Davies, Kevin Ji, Kevin Keating, Kevin Mader, Kevin Rose, Kexuan Sun, Kieran Ramos, Kimmo Palin, Kjartan Myrdal, Kjell Le, Klara Gerlei, Konrad Förstner, Konstantin Tretyakov, Kristen M. Thyng, Kyle Bridgemohansingh, Kyle Sunden, Kyler Brown, Lance Hepler, Laptop11_ASPP2016, Larry Bradley, Laurent Thomas, Lawrence D'Anna, Leeonadoh, Lennart Fricke, Leo Singer, Leon Loopik, Leon Yin, LevN0, Levi Kilcher, Liam Brannigan, Lion Krischer, Lionel Miller, Lodato Luciano, Lori J, Loïc Estève, Loïc Séguin-C, Luca Verginer, Luis Pedro Coelho, Luke Davis, Maarten Baert, Maciej Dems, Magnus Nord, Maik Riechert, Majid alDosari, Maksym P, Manan, Manan Kevadiya, Manish Devgan, Manuel GOACOLOU, Manuel Jung, Manuel Metz, Manuel Nuno Melo, Maoz Gelbart, Marat K, Marc Abramowitz, Marcel Martin, Marco Gorelli, MarcoGorelli, Marcos Duarte, Marek Rudnicki, Marianne Corvellec, Marin Gilles, Mark Harfouche, Mark Wolf, Marko Baštovanović, Markus Roth, Markus Rothe, Martin Dengler, Martin Fitzpatrick, Martin Spacek, Martin Teichmann, Martin Thoma, Martin Ueding, Massimo Santini, Masud Rahman, Mathieu Duponchelle, Matt Giuca, Matt Hancock, Matt Klein, Matt Li, Matt Newville, Matt Shen, Matt Terry, Matthew Bell, Matthew Brett, Matthew Emmett, Matthias Bussonnier, Matthias Geier, Matthias Lüthi, Matthieu Caneill, Matthieu-Dartiailh, Matti Picus, Matěj Týč, Max Chen, Max Humber, Max Shinn, Maximilian Albert, Maximilian Maahn, Maximilian Nöthe, Maximilian Trescher, Meeseeks-Machine, Mellissa Cross, Mher Kazandjian, Michael, Michael Droettboom, Michael Jancsy, Michael Sarahan, Michael Scott Cuthbert, Michael Seifert, Michael Welter, Michaël Defferrard, Michele Mastropietro, Michiel de Hoon, Michka Popoff, Mike Henninger, Mike Jarvis, Mike Kaufman, Mikhail Korobov, MinRK, Mingkai Dong, Minty Zhang, MirandaXM, Miriam Sierig, Mitar, Molly Rossow, Moritz Boehle, Mudit Surana, Muhammad Mehdi, MuhammadFarooq1234, Mykola Dvornik, Naoya Kanai, Nathan Goldbaum, Nathan Musoke, Nathaniel M. Beaver, Neil, Neil Crighton, Nelle Varoquaux, Niall Robinson, Nic Eggert, Nicholas Devenish, Nick Forrington, Nick Garvey, Nick Papior, Nick Pope, Nick Semenkovich, Nico Schlömer, Nicolas Courtemanche, Nicolas P. Rougier, Nicolas Pinto, Nicolas Tessore, Nik Quibin, Nikita Kniazev, Niklas Koep, Nikolay Vyahhi, Nils Werner, Ninad Bhat, Norbert Nemec, Norman Fomferra, O. Castany, OceanWolf, Oleg Selivanov, Olga Botvinnik, Oliver Natt, Oliver Willekens, Olivier, Om Sitapara, Omar Chehab, Oriol Abril, Orso Meneghini, Osarumwense, Pankaj Pandey, Paramonov Andrey, Parfenov Sergey, Pascal Bugnion, Pastafarianist, Patrick Chen, Patrick Feiring, Patrick Marsh, Patrick Shriwise, Patrick-Feiring, Paul, Paul Barret, Paul Ganssle, Paul Gierz, Paul Hobson, Paul Hoffman, Paul Ivanov, Paul J. Koprowski, Paul Kirow, Paul Romano, Paul Seyfert, Pauli Virtanen, Pavel Fedin, Pavol Juhas, Per Parker, Perry Greenfield, Pete Bachant, Pete Huang, Pete Peterson, Peter Iannucci, Peter Mackenzie-Helnwein, Peter Mortensen, Peter Schutt, Peter St. John, Peter Würtz, Petr Danecek, Phil Elson, Phil Ruffwind, Philippe Pinard, Pierre Haessig, Pierre Thibault, Pierre de Buyl, Pim Schellart, Piti Ongmongkolkul, Po, Pranav Garg, Przemysław Dąbek, Puneeth Chaganti, QiCuiHub, Qingpeng "Q.P." Zhang, RAKOTOARISON Herilalaina, Ram Rachum, Ramiro Gómez, Randy Olson, Raphael, Rasmus Diederichsen, Ratin_Kumar, Rebecca W Perry, Reinier Heeres, Remi Rampin, Ricardo Mendes, Riccardo Di Maio, Richard Gowers, Richard Hattersley, Richard Ji-Cathriner, Richard Trieu, Ricky, Rishikesh, Rob Harrigan, Robert Johansson, Robin Dunn, Robin Neatherway, Robin Wilson, Rohan Walker, Roland Wirth, Roman Yurchak, Ronald Hartley-Davies, RoryIAngus,

Roy Smith, Rui Lopes, Russell Owen, RutgerK, Ryan, Ryan Blomberg, Ryan D'Souza, Ryan Dale, Ryan May, Ryan Morshead, Ryan Nelson, RyanPan, SBCV, Sairam Pillai, Saket Choudhary, Salganos, Salil Vanvari, Salinder Sidhu, Sam Vaughan, SamSchott, Sameer D'Costa, Samesh Lakhotia, Samson, Samuel St-Jean, Sander, Sandro Tosi, Scott Howard, Scott Lasley, Scott Lawrence, Scott Stevenson, Sean Farley, Sebastian Bullinger, Sebastian Pinnau, Sebastian Raschka, Sebastián Vanrell, Seraphim Alvanides, Sergey B Kirpichev, Sergey Kholodilov, Sergey Koposov, Seunghoon Park, Siddhesh Poyarekar, Sidharth Bansal, Silviu Tantos, Simon Cross, Simon Gibbons, Simon Legner, Skelpdar, Skipper Seabold, Slav Basharov, Snowhite, SojiroFukuda, Sourav Singh, Spencer McIntyre, Stanley, Simon, Stefan Lehmann, Stefan Mitic, Stefan Pfenninger, Stefan van der Walt, Stefano Rivera, Stephan Erb, Stephane Raynaud, Stephen Horst, Stephen-Chilcote, Sterling Smith, Steve Chaplin, Steve Dower, Steven G. Johnson, Steven Munn, Steven Silvester, Steven Tilley, Stuart Mumford, Tadeo Corradi, Taehoon Lee, Takafumi Arakaki, Takeshi Kanmae, Tamas Gal, Tanuj, Taras Kuzyo, Ted Drain, Ted Petrou, Terence Honles, Terrence J. Katzenbaer, Terrence Katzenbaer, The Gitter Badger, Thein Oo, Thomas A Caswell, Thomas Hisch, Thomas Kluyver, Thomas Lake, Thomas Levine, Thomas Mansencal, Thomas Robitaille, Thomas Spura, Thomas VINCENT, Thorsten Liebig, Tian Xia, Till Hoffmann, Till Stensitzki, Tim Hoffmann, Timo Vanwynsberghe, Tobia De Koninck, Tobias Froehlich, Tobias Hoppe, Tobias Megies, Todd Jennings, Todd Miller, Tom, Tom Augspurger, Tom Dupré la Tour, Tom Flannaghan, Tomas Kazmar, Tony S Yu, Tor Colvin, Travis Oliphant, Trevor Bekolay, Trish Gillett-Kawamoto, Truong Pham, Tuan Dung Tran, Tyler Makaro, Tyrone Xiong, Ulrich Dobramysl, Umair Idris, V. Armando Solé, V. R, Vadim Markovtsev, Valentin Haenel, Valentin Schmidt, Vedant Nanda, Venkada, Vidur Satija, Viktor Kerkez, Vincent L.M. Mazoyer, Viraj Mohile, Vitaly Buka, Vlad Seghete, Víctor Terrón, Víctor Zabalza, WANG Aiyong, Warren Weckesser, Wen Li, Wendell Smith, Werner F Bruhin, Wes Campaigne, Wieland Hoffmann, Will Handley, Will Silva, William Granados, William Mallard, William Manley, Wouter Overmeire, Xiaowen Tang, Xufeng Wang, Yann Tambouret, Yao-Yuan Mao, Yaron de Leeuw, Yu Feng, Yue Zhihan, Yunfei Yang, Yuri D'Elia, Yuval Langer, Yuxin Wu, Yuya, Zac Hatfield-Dodds, Zach Pincus, Zair Mubashar, Zbigniew Jędrzejewski-Szmek, Zhili (Jerry) Pan, Zulko, ahed87, akrherz, alcinos, alex, alvarosg, andrzejnovak, aneda, anykraus, aparamon, apodemus, arokem, as691454, aseagram, ash13, aszilagyi, azure-pipelines[bot], bblay, bduick, bev-a-tron, blackw1ng, blah blah, brut, btang02, buefox, burrbull, butterw, cammil, captainwhippet, cclauss, ch3rn0v, chadawagner, chaoyi1, chebee7i, chelseatroy, chuanzhu xu, cknd, cldssty, clintval, dabana, dahlbaek, danielballan, daronjp, davidovitch, daydreamt, deeenes, deepyaman, djdt, dlmccaffrey, domspad, donald, donchanee, drevicko, e-q, elpres, endolith, esvhd, et2010, fardal, ffteja, fgb, fibersnet, fourpoints, fredrik-1, frenchwr, fuzzythecat, fvgoto, gcallah, gitj, gluap, gnaggnoyil, goir, goldstarwebs, greg-roper, gregorybchris, gwin-zegal, hannah, helmiriawan, henryhu123, hugadams, ilivni, insertroar, itziakos, jacob-on-github, jb-leger, jbbrokaw, jbhopkins, jdollichon, jerrylui803, jess, jfbu, jhelie, jli, joaonsg, joelostblom, jonchar, juan.gonzalez, kcrisman, keithbriggs, kelsiegr, khyox, kikocorreoso, klaus, klonuo, kolibril13, kramer65, krishna katyal, ksafran, kshramt, lboogaard, legitz7, lepuchi, lichri12, limtaesu, lspvic, luftek, luz.paz, lzkelley, mamrehn, marky, masamson, mbyt, mcelrath, mcquin, mdipierro, mikhailov, miquelastein, mitch, mlub, mobando, mromanie, muahah, myyc, nathan78906, navdeep rana, nbrunett, nemanja, neok-m4700, nepix32, nickys-

tringer, njwhite, nmartensen, nwin, ob, pdubcali, pibion, pkienzle, productivememberofsociety666, profholzer, pupssman, rahiel, ranjanm, rebot, rhoef, rsnape, ruin, rvhbooth, s0vereign, s9w, saksmito, scls19fr, scott-vsi, sdementen, serv-inc, settheory, sfroid, shaunwbell, simon-kraeusel, simonpf, sindunuragarp, smheidrich, sohero, spiessbuerger, stahlous, stone, stonebig, switham, sxntxn, syngron, teresy, thoo, thuvejan, tmdavison, tomoemon, tonyyli, torfbolt, u55, ugurthemaster, ultra-andy, vab9, vbr, vishalBindal, vraelvrangr, watkinrt, woclass, xbtsw, xuanyuansen, y1thof, yeo, zhangeugenia, zhoubecky, Élie Gouzien, Андрей Парамонов

Some earlier contributors not included above are (with apologies to any we have missed):

Charles Twardy, Gary Ruben, John Gill, David Moore, Paul Barrett, Jared Wahlstrand, Jim Benson, Paul Mcguire, Andrew Dalke, Nadia Dencheva, Baptiste Carvello, Sigve Tjoraand, Ted Drain, James Amundson, Daishi Harada, Nicolas Young, Paul Kienzle, John Porter, and Jonathon Taylor.

Thanks to Tony Yu for the original logo design.

We also thank all who have reported bugs, commented on proposed changes, or otherwise contributed to Matplotlib's development and usefulness.

Part II

The Matplotlib FAQ

CHAPTER

TWELVE

INSTALLATION

Contents

12.1 Report a compilation problem

See *Getting help*.

12.2 Matplotlib compiled fine, but nothing shows up when I use it

The first thing to try is a *clean install* and see if that helps. If not, the best way to test your install is by running a script, rather than working interactively from a python shell or an integrated development environment such as `IDLE` which add additional complexities. Open up a UNIX shell or a DOS command prompt and run, for example:

```
python -c "from pylab import *; set_loglevel('debug'); plot(); show()"
```

This will give you additional information about which backends Matplotlib is loading, version information, and more. At this point you might want to make sure you understand Matplotlib's *configuration* process, governed by the `matplotlibrc` configuration file which contains instructions within and the concept of the Matplotlib backend.

If you are still having trouble, see *Getting help*.

12.3 How to completely remove Matplotlib

Occasionally, problems with Matplotlib can be solved with a clean installation of the package. In order to fully remove an installed Matplotlib:

1. Delete the caches from your *Matplotlib configuration directory*.
2. Delete any Matplotlib directories or eggs from your *installation directory*.

12.4 Linux Notes

To install Matplotlib at the system-level, we recommend that you use your distribution's package manager. This will guarantee that Matplotlib's dependencies will be installed as well.

If, for some reason, you cannot use the package manager, you may use the wheels available on PyPI:

```
python -mpip install matplotlib
```

or *build Matplotlib from source*.

12.5 OSX Notes

12.5.1 Which python for OSX?

Apple ships OSX with its own Python, in `/usr/bin/python`, and its own copy of Matplotlib. Unfortunately, the way Apple currently installs its own copies of NumPy, Scipy and Matplotlib means that these packages are difficult to upgrade (see system python packages). For that reason we strongly suggest that you install a fresh version of Python and use that as the basis for installing libraries such as NumPy and Matplotlib. One convenient way to install Matplotlib with other useful Python software is to use the Anaconda Python scientific software collection, which includes Python itself and a wide range of libraries; if you need a library that is not available from the collection, you can install it yourself using standard methods such as *pip*. See the Ananconda web page for installation support.

Other options for a fresh Python install are the standard installer from python.org, or installing Python using a general OSX package management system such as homebrew or macports. Power users on OSX will likely want one of homebrew or macports on their system to install open source software packages, but it is perfectly possible to use these systems with another source for your Python binary, such as Anaconda or Python.org Python.

12.5.2 Installing OSX binary wheels

If you are using Python from https://www.python.org, Homebrew, or Macports, then you can use the standard pip installer to install Matplotlib binaries in the form of wheels.

pip is installed by default with python.org and Homebrew Python, but needs to be manually installed on Macports with

```
sudo port install py36-pip
```

Once pip is installed, you can install Matplotlib and all its dependencies with from the Terminal.app command line:

```
python3 -mpip install matplotlib
```

(`sudo python3.6 ...` on Macports).

You might also want to install IPython or the Jupyter notebook (`python3 -mpip install ipython notebook`).

12.5.3 Checking your installation

The new version of Matplotlib should now be on your Python "path". Check this at the Terminal.app command line:

```
python3 -c 'import matplotlib; print(matplotlib.__version__, matplotlib.__file__)'
```

You should see something like

```
3.0.0 /Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/site-packages/
↪matplotlib/__init__.py
```

where `3.0.0` is the Matplotlib version you just installed, and the path following depends on whether you are using Python.org Python, Homebrew or Macports. If you see another version, or you get an error like

```
Traceback (most recent call last):
  File "<string>", line 1, in <module>
ImportError: No module named matplotlib
```

then check that the Python binary is the one you expected by running

```
which python3
```

If you get a result like `/usr/bin/python...`, then you are getting the Python installed with OSX, which is probably not what you want. Try closing and restarting Terminal.app before running the check again. If that doesn't fix the problem, depending on which Python you wanted to use, consider reinstalling Python.org Python, or check your homebrew or macports setup. Remember that the disk image installer only works for Python.org Python, and will not get picked up by other Pythons. If all these fail, please *let us know*.

12.6 Install from source

A C compiler is required. Typically, on Linux, you will need `gcc`, which should be installed using your distribution's package manager; on macOS, you will need xcode; on Windows, you will need Visual Studio 2015 or later.

Clone the main source using one of:

```
git clone git@github.com:matplotlib/matplotlib.git
```

or:

```
git clone git://github.com/matplotlib/matplotlib.git
```

and build and install with:

```
cd matplotlib
python -mpip install .
```

If you want to be able to follow the development branch as it changes just replace the last step with:

```
python -mpip install -e .
```

This creates links and installs the command line script in the appropriate places.

Then, if you want to update your Matplotlib at any time, just do:

```
git pull
```

When you run `git pull`, if the output shows that only Python files have been updated, you are all set. If C files have changed, you need to run `pip install -e .` again to compile them.

There is more information on *using git* in the developer docs.

CHAPTER

THIRTEEN

HOW-TO

Contents

13.1 How-to: Plotting

13.1.1 Plot `numpy.datetime64` values

As of Matplotlib 2.2, `numpy.datetime64` objects are handled the same way as `datetime.datetime` objects.

If you prefer the pandas converters and locators, you can register them. This is done automatically when calling a pandas plot function and may be unnecessary when using pandas instead of Matplotlib directly.

```
from pandas.plotting import register_matplotlib_converters
register_matplotlib_converters()
```

13.1.2 Check whether a figure is empty

Empty can actually mean different things. Does the figure contain any artists? Does a figure with an empty *Axes* still count as empty? Is the figure empty if it was rendered pure white (there may be artists present, but they could be outside the drawing area or transparent)?

For the purpose here, we define empty as: "The figure does not contain any artists except it's background patch." The exception for the background is necessary, because by default every figure contains a *Rectangle* as it's background patch. This definition could be checked via:

```
def is_empty(figure):
    """
    Return whether the figure contains no Artists (other than the default
    background patch).
    """
    contained_artists = figure.get_children()
    return len(contained_artists) <= 1
```

We've decided not to include this as a figure method because this is only one way of defining empty, and checking the above is only rarely necessary. Usually the user or program handling the figure know if they have added something to the figure.

Checking whether a figure would render empty cannot be reliably checked except by actually rendering the figure and investigating the rendered result.

13.1.3 Find all objects in a figure of a certain type

Every Matplotlib artist (see *Artist tutorial*) has a method called *findobj()* that can be used to recursively search the artist for any artists it may contain that meet some criteria (e.g., match all *Line2D* instances or match some arbitrary filter function). For example, the following snippet finds every object in the figure which has a `set_color` property and makes the object blue:

```
def myfunc(x):
    return hasattr(x, 'set_color')

for o in fig.findobj(myfunc):
    o.set_color('blue')
```

You can also filter on class instances:

```
import matplotlib.text as text
for o in fig.findobj(text.Text):
    o.set_fontstyle('italic')
```

13.1.4 How to prevent ticklabels from having an offset

The default formatter will use an offset to reduce the length of the ticklabels. To turn this feature off on a per-axis basis:

```
ax.get_xaxis().get_major_formatter().set_useOffset(False)
```

set `rcParams["axes.formatter.useoffset"]` (default: `True`), or use a different formatter. See *ticker* for details.

13.1.5 Save transparent figures

The *savefig()* command has a keyword argument *transparent* which, if 'True', will make the figure and axes backgrounds transparent when saving, but will not affect the displayed image on the screen.

If you need finer grained control, e.g., you do not want full transparency or you want to affect the screen displayed version as well, you can set the alpha properties directly. The figure has a *Rectangle* instance called *patch* and the axes has a Rectangle instance called *patch*. You can set any property on them directly (*facecolor*, *edgecolor*, *linewidth*, *linestyle*, *alpha*). e.g.:

```
fig = plt.figure()
fig.patch.set_alpha(0.5)
ax = fig.add_subplot(111)
ax.patch.set_alpha(0.5)
```

If you need *all* the figure elements to be transparent, there is currently no global alpha setting, but you can set the alpha channel on individual elements, e.g.:

```
ax.plot(x, y, alpha=0.5)
ax.set_xlabel('volts', alpha=0.5)
```

13.1.6 Save multiple plots to one pdf file

Many image file formats can only have one image per file, but some formats support multi-page files. Currently only the pdf backend has support for this. To make a multi-page pdf file, first initialize the file:

```
from matplotlib.backends.backend_pdf import PdfPages
pp = PdfPages('multipage.pdf')
```

You can give the *PdfPages* object to *savefig()*, but you have to specify the format:

```
plt.savefig(pp, format='pdf')
```

An easier way is to call *PdfPages.savefig*:

```
pp.savefig()
```

Finally, the multipage pdf object has to be closed:

```
pp.close()
```

The same can be done using the pgf backend:

```
from matplotlib.backends.backend_pgf import PdfPages
```

13.1.7 Move the edge of an axes to make room for tick labels

For subplots, you can control the default spacing on the left, right, bottom, and top as well as the horizontal and vertical spacing between multiple rows and columns using the *matplotlib.figure.Figure.subplots_adjust()* method (in pyplot it is *subplots_adjust()*). For example, to move the bottom of the subplots up to make room for some rotated x tick labels:

```
fig = plt.figure()
fig.subplots_adjust(bottom=0.2)
ax = fig.add_subplot(111)
```

You can control the defaults for these parameters in your `matplotlibrc` file; see *Customizing Matplotlib with style sheets and rcParams*. For example, to make the above setting permanent, you would set:

```
figure.subplot.bottom : 0.2   # the bottom of the subplots of the figure
```

The other parameters you can configure are, with their defaults

***left* = 0.125**

the left side of the subplots of the figure

***right* = 0.9**

the right side of the subplots of the figure

***bottom* = 0.1**

the bottom of the subplots of the figure

***top* = 0.9**

the top of the subplots of the figure

***wspace* = 0.2**

the amount of width reserved for space between subplots, expressed as a fraction of the average axis width

***hspace* = 0.2**

the amount of height reserved for space between subplots, expressed as a fraction of the average axis height

If you want additional control, you can create an *Axes* using the *axes()* command (or equivalently the figure *add_axes()* method), which allows you to specify the location explicitly:

```
ax = fig.add_axes([left, bottom, width, height])
```

where all values are in fractional (0 to 1) coordinates. See /gallery/subplots_axes_and_figures/axes_demo for an example of placing axes manually.

13.1.8 Automatically make room for tick labels

Note: This is now easier to handle than ever before. Calling *tight_layout()* or alternatively using `constrained_layout=True` argument in *subplots()* can fix many common layout issues. See the *Tight Layout guide* and *Constrained Layout Guide* for more details.

The information below is kept here in case it is useful for other purposes.

In most use cases, it is enough to simply change the subplots adjust parameters as described in *Move the edge of an axes to make room for tick labels*. But in some cases, you don't know ahead of time what your tick labels will be, or how large they will be

(data and labels outside your control may be being fed into your graphing application), and you may need to automatically adjust your subplot parameters based on the size of the tick labels. Any *Text* instance can report its extent in window coordinates (a negative x coordinate is outside the window), but there is a rub.

The *RendererBase* instance, which is used to calculate the text size, is not known until the figure is drawn (*draw()*). After the window is drawn and the text instance knows its renderer, you can call *get_window_extent()*. One way to solve this chicken and egg problem is to wait until the figure is draw by connecting (*mpl_connect()*) to the "on_draw" signal (*DrawEvent*) and get the window extent there, and then do something with it, e.g., move the left of the canvas over; see *Event handling and picking*.

Here is an example that gets a bounding box in relative figure coordinates (0..1) of each of the labels and uses it to move the left of the subplots over so that the tick labels fit in the figure:

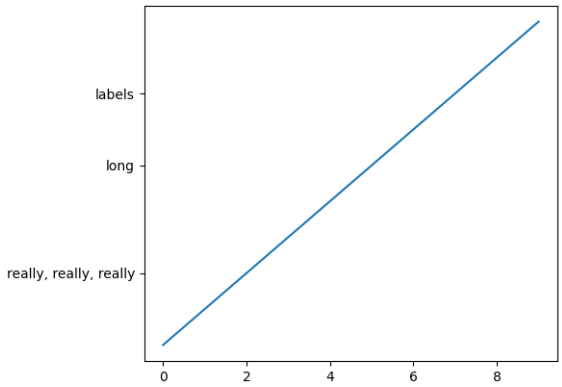


Fig. 1: Auto Subplots Adjust

13.1.9 Configure the tick widths

Wherever possible, it is recommended to use the *tick_params()* or *set_tick_params()* methods to modify tick properties:

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
ax.plot(range(10))

ax.tick_params(width=10)

plt.show()
```

For more control of tick properties that are not provided by the above methods, it is important to know that in Matplotlib, the ticks are *markers*. All *Line2D* objects support a line (solid, dashed, etc) and a marker (circle, square, tick). The tick width is

controlled by the "markeredgewidth" property, so the above effect can also be achieved by:

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
ax.plot(range(10))

for line in ax.get_xticklines() + ax.get_yticklines():
    line.set_markeredgewidth(10)

plt.show()
```

The other properties that control the tick marker, and all markers, are markerfacecolor, markeredgecolor, markeredgewidth, markersize. For more information on configuring ticks, see *Axis containers* and *Tick containers*.

13.1.10 Align my ylabels across multiple subplots

If you have multiple subplots over one another, and the y data have different scales, you can often get ylabels that do not align vertically across the multiple subplots, which can be unattractive. By default, Matplotlib positions the x location of the ylabel so that it does not overlap any of the y ticks. You can override this default behavior by specifying the coordinates of the label. The example below shows the default behavior in the left subplots, and the manual setting in the right subplots.

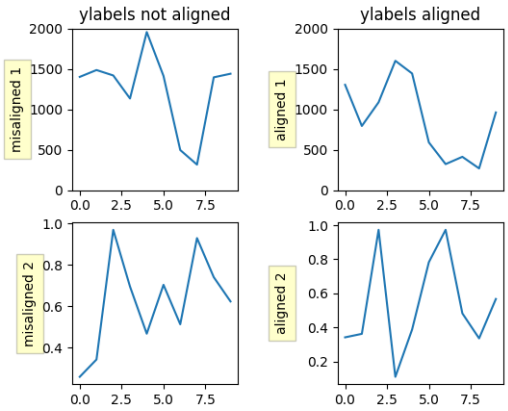


Fig. 2: Align Ylabels

13.1.11 Skip dates where there is no data

When plotting time series, e.g., financial time series, one often wants to leave out days on which there is no data, e.g., weekends. By passing in dates on the x-xaxis, you get large horizontal gaps on periods when there is not data. The solution is to pass in some proxy x-data, e.g., evenly sampled indices, and then use a custom formatter to format these as dates. /gallery/text_labels_and_annotations/date_index_formatter demonstrates how to use an 'index formatter' to achieve the desired plot.

13.1.12 Control the depth of plot elements

Within an axes, the order that the various lines, markers, text, collections, etc appear is determined by the *set_zorder()* property. The default order is patches, lines, text, with collections of lines and collections of patches appearing at the same level as regular lines and patches, respectively:

```
line, = ax.plot(x, y, zorder=10)
```

You can also use the Axes property *set_axisbelow()* to control whether the grid lines are placed above or below your other plot elements.

13.1.13 Make the aspect ratio for plots equal

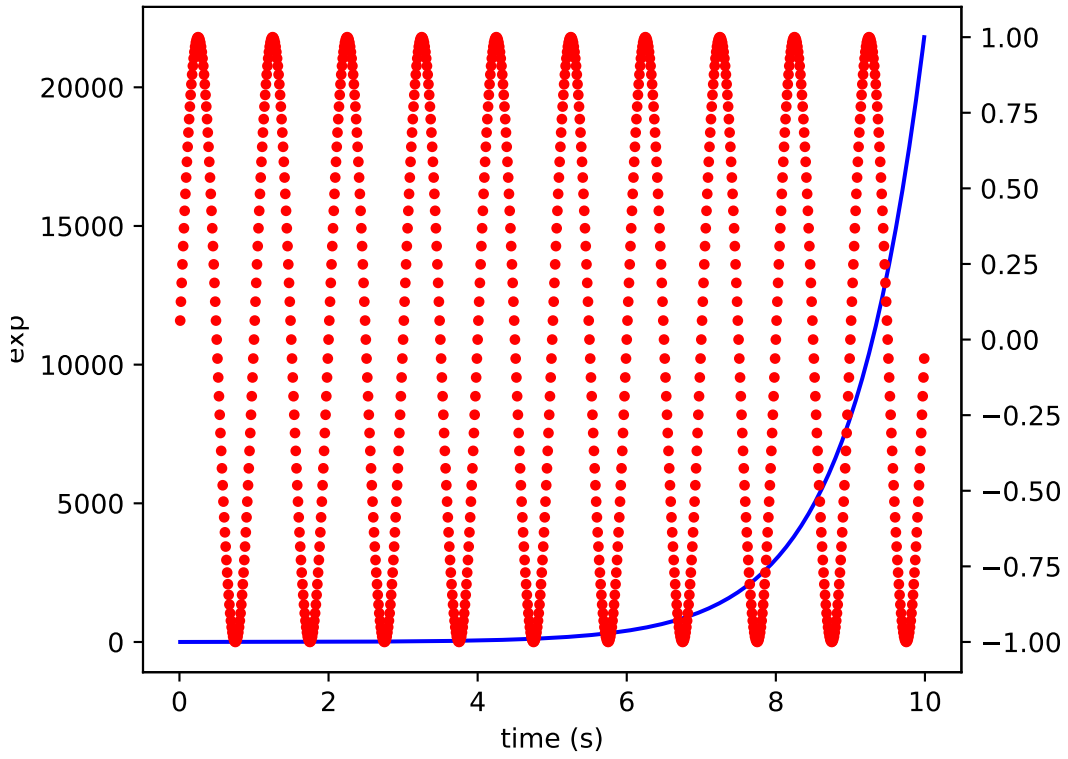
The Axes property *set_aspect()* controls the aspect ratio of the axes. You can set it to be 'auto', 'equal', or some ratio which controls the ratio:

```
ax = fig.add_subplot(111, aspect='equal')
```

13.1.14 Multiple y-axis scales

A frequent request is to have two scales for the left and right y-axis, which is possible using *twinx()* (more than two scales are not currently supported, though it is on the wish list). This works pretty well, though there are some quirks when you are trying to interactively pan and zoom, because both scales do not get the signals.

The approach uses *twinx()* (and its sister *twiny()*) to use *2 different axes*, turning the axes rectangular frame off on the 2nd axes to keep it from obscuring the first, and manually setting the tick locs and labels as desired. You can use separate `matplotlib.ticker` formatters and locators as desired because the two axes are independent.



13.1.15 Generate images without having a window appear

Simply do not call *show*, and directly save the figure to the desired format:

```
import matplotlib.pyplot as plt
plt.plot([1, 2, 3])
plt.savefig('myfig.png')
```

See also:

How to use Matplotlib in a web application server for information about running matplotlib inside of a web application.

13.1.16 Use `show()`

When you want to view your plots on your display, the user interface backend will need to start the GUI mainloop. This is what *show()* does. It tells Matplotlib to raise all of the figure windows created so far and start the mainloop. Because this mainloop is blocking by default (i.e., script execution is paused), you should only call this once per script, at the end. Script execution is resumed after the last window is closed. Therefore, if you are using Matplotlib to generate only images and do not want a user

interface window, you do not need to call `show` (see *Generate images without having a window appear* and *What is a backend?*).

Note: Because closing a figure window unregisters it from pyplot, you must call *`savefig`* *before* calling `show` if you wish to save the figure as well as view it.

Whether `show` blocks further execution of the script or the python interpreter depends on whether Matplotlib is set to use interactive mode. In non-interactive mode (the default setting), execution is paused until the last figure window is closed. In interactive mode, the execution is not paused, which allows you to create additional figures (but the script won't finish until the last figure window is closed).

Because it is expensive to draw, you typically will not want Matplotlib to redraw a figure many times in a script such as the following:

```
plot([1, 2, 3])          # draw here?
xlabel('time')           # and here?
ylabel('volts')          # and here?
title('a simple plot')   # and here?
show()
```

However, it is *possible* to force Matplotlib to draw after every command, which might be what you want when working interactively at the python console (see *Interactive Figures*), but in a script you want to defer all drawing until the call to `show`. This is especially important for complex figures that take some time to draw. *`show()`* is designed to tell Matplotlib that you're all done issuing commands and you want to draw the figure now.

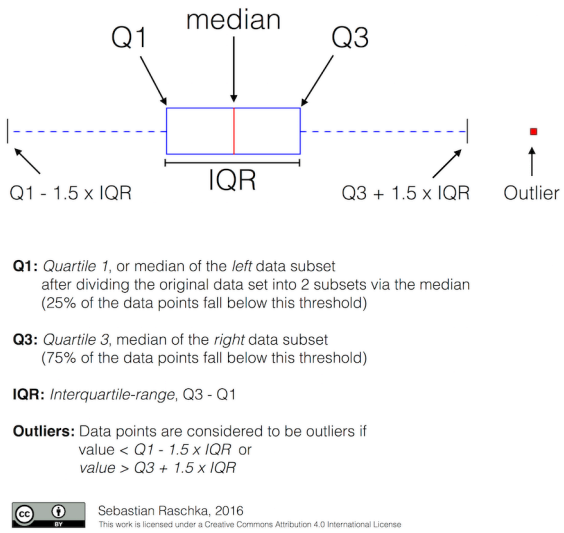
Note: *`show()`* should typically only be called at most once per script and it should be the last line of your script. At that point, the GUI takes control of the interpreter. If you want to force a figure draw, use *`draw()`* instead.

New in version v1.0.0: Matplotlib 1.0.0 and 1.0.1 added support for calling `show` multiple times per script, and harmonized the behavior of interactive mode, across most backends.

13.1.17 Interpreting box plots and violin plots

Tukey's box plots (Robert McGill, John W. Tukey and Wayne A. Larsen: "The American Statistician" Vol. 32, No. 1, Feb., 1978, pp. 12-16) are statistical plots that provide useful information about the data distribution such as skewness. However, bar plots with error bars are still the common standard in most scientific literature, and thus, the interpretation of box plots can be challenging for the unfamiliar reader. The figure below illustrates the different visual features of a box plot.

Violin plots are closely related to box plots but add useful information such as the



distribution of the sample data (density trace). Violin plots were added in Matplotlib 1.4.

13.1.18 Working with threads

Matplotlib is not thread-safe: in fact, there are known race conditions that affect certain artists. Hence, if you work with threads, it is your responsibility to set up the proper locks to serialize access to Matplotlib artists.

You may be able to work on separate figures from separate threads. However, you must in that case use a *non-interactive backend* (typically Agg), because most GUI backends *require* being run from the main thread as well.

13.2 How-to: Contributing

13.2.1 Request a new feature

Is there a feature you wish Matplotlib had? Then ask! The best way to get started is to email the developer mailing list for discussion. This is an open source project developed primarily in the contributors free time, so there is no guarantee that your feature will be added. The *best* way to get the feature you need added is to contribute it your self.

13.2.2 Reporting a bug or submitting a patch

The development of Matplotlib is organized through github. If you would like to report a bug or submit a patch please use that interface.

To report a bug create an issue on github (this requires having a github account). Please include a Short, Self Contained, Correct (Compilable), Example demonstrating what the bug is. Including a clear, easy to test example makes it easy for the developers to evaluate the bug. Expect that the bug reports will be a conversation. If you do not want to register with github, please email bug reports to the mailing list.

The easiest way to submit patches to Matplotlib is through pull requests on github. Please see the *The Matplotlib Developers' Guide* for the details.

13.2.3 Contribute to Matplotlib documentation

Matplotlib is a big library, which is used in many ways, and the documentation has only scratched the surface of everything it can do. So far, the place most people have learned all these features are through studying the examples-index, which is a recommended and great way to learn, but it would be nice to have more official narrative documentation guiding people through all the dark corners. This is where you come in.

There is a good chance you know more about Matplotlib usage in some areas, the stuff you do every day, than many of the core developers who wrote most of the documentation. Just pulled your hair out compiling Matplotlib for Windows? Write a FAQ or a section for the *Installation* page. Are you a digital signal processing wizard? Write a tutorial on the signal analysis plotting functions like *xcorr()*, *psd()* and *specgram()*. Do you use Matplotlib with django or other popular web application servers? Write a FAQ or tutorial and we'll find a place for it in the *User's Guide*. And so on... I think you get the idea.

Matplotlib is documented using the sphinx extensions to restructured text (ReST). sphinx is an extensible python framework for documentation projects which generates HTML and PDF, and is pretty easy to write; you can see the source for this document or any page on this site by clicking on the *Show Source* link at the end of the page in the sidebar.

The sphinx website is a good resource for learning sphinx, but we have put together a cheat-sheet at *Writing documentation* which shows you how to get started, and outlines the Matplotlib conventions and extensions, e.g., for including plots directly from external code in your documents.

Once your documentation contributions are working (and hopefully tested by actually *building* the docs) you can submit them as a patch against git. See *Install git* and *Reporting a bug or submitting a patch*. Looking for something to do? Search for TODO or look at the open issues on github.

13.3 How to use Matplotlib in a web application server

In general, the simplest solution when using Matplotlib in a web server is to completely avoid using pyplot (pyplot maintains references to the opened figures to make *show* work, but this will cause memory leaks unless the figures are properly closed). Since Matplotlib 3.1, one can directly create figures using the *Figure* constructor and save them to in-memory buffers. The following example uses Flask, but other frameworks work similarly:

```
import base64
from io import BytesIO

from flask import Flask
from matplotlib.figure import Figure

app = Flask(__name__)

@app.route("/")
def hello():
    # Generate the figure **without using pyplot**.
    fig = Figure()
    ax = fig.subplots()
    ax.plot([1, 2])
    # Save it to a temporary buffer.
    buf = BytesIO()
    fig.savefig(buf, format="png")
    # Embed the result in the html output.
    data = base64.b64encode(buf.getbuffer()).decode("ascii")
    return f"<img src='{data}'/>"
```

When using Matplotlib versions older than 3.1, it is necessary to explicitly instantiate an Agg canvas; see e.g. /gallery/user_interfaces/canvasagg.

13.3.1 Clickable images for HTML

Andrew Dalke of Dalke Scientific has written a nice article on how to make html click maps with Matplotlib agg PNGs. We would also like to add this functionality to SVG. If you are interested in contributing to these efforts that would be great.

CHAPTER

FOURTEEN

TROUBLESHOOTING

Contents

14.1 Obtaining Matplotlib version

To find out your Matplotlib version number, import it and print the `__version__` attribute:

```
>>> import matplotlib
>>> matplotlib.__version__
'0.98.0'
```

14.2 `matplotlib` install location

You can find what directory Matplotlib is installed in by importing it and printing the `__file__` attribute:

```
>>> import matplotlib
>>> matplotlib.__file__
'/home/jdhunter/dev/lib64/python2.5/site-packages/matplotlib/__init__.pyc'
```

14.3 `matplotlib` configuration and cache directory locations

Each user has a Matplotlib configuration directory which may contain a *matplotlibrc* file. To locate your `matplotlib/` configuration directory, use *matplotlib.get_configdir()*:

```
>>> import matplotlib as mpl
>>> mpl.get_configdir()
'/home/darren/.config/matplotlib'
```

On unix-like systems, this directory is generally located in your *HOME* directory under the `.config/` directory.

In addition, users have a cache directory. On unix-like systems, this is separate from the configuration directory by default. To locate your `.cache/` directory, use *matplotlib.get_cachedir()*:

```
>>> import matplotlib as mpl
>>> mpl.get_cachedir()
'/home/darren/.cache/matplotlib'
```

On windows, both the config directory and the cache directory are the same and are in your `Documents and Settings` or `Users` directory by default:

```
>>> import matplotlib as mpl
>>> mpl.get_configdir()
'C:\\Documents and Settings\\jdhunter\\.matplotlib'
>>> mpl.get_cachedir()
'C:\\Documents and Settings\\jdhunter\\.matplotlib'
```

If you would like to use a different configuration directory, you can do so by specifying the location in your *MPLCONFIGDIR* environment variable -- see *Setting environment variables in Linux and macOS*. Note that *MPLCONFIGDIR* sets the location of both the configuration directory and the cache directory.

14.4 Getting help

There are a number of good resources for getting help with Matplotlib. There is a good chance your question has already been asked:

- The mailing list archive.
- GitHub issues.
- Stackoverflow questions tagged matplotlib.

If you are unable to find an answer to your question through search, please provide the following information in your e-mail to the mailing list:

- Your operating system (Linux/UNIX users: post the output of `uname -a`).

- Matplotlib version:

```
python -c "import matplotlib; print(matplotlib.__version__)"
```

- Where you obtained Matplotlib (e.g., your Linux distribution's packages, GitHub, PyPI, or Anaconda).
- Any customizations to your `matplotlibrc` file (see *Customizing Matplotlib with style sheets and rcParams*).
- If the problem is reproducible, please try to provide a *minimal*, standalone Python script that demonstrates the problem. This is *the* critical step. If you can't post a piece of code that we can run and reproduce your error, the chances of getting help are significantly diminished. Very often, the mere act of trying to minimize your code to the smallest bit that produces the error will help you find a bug in *your* code that is causing the problem.
- Matplotlib provides debugging information through the `logging` library, and a helper function to set the logging level: one can call

```
plt.set_loglevel("info")  # or "debug" for more info
```

  to obtain this debugging information.

  Standard functions from the `logging` module are also applicable; e.g. one could call `logging.basicConfig(level="DEBUG")` even before importing Matplotlib (this is in particular necessary to get the logging info emitted during Matplotlib's import), or attach a custom handler to the "matplotlib" logger. This may be useful if you use a custom logging configuration.

If you compiled Matplotlib yourself, please also provide:

- any changes you have made to `setup.py` or `setupext.py`.
- the output of:

```
rm -rf build
python setup.py build
```

  The beginning of the build output contains lots of details about your platform that are useful for the Matplotlib developers to diagnose your problem.

- your compiler version -- e.g., `gcc --version`.

Including this information in your first e-mail to the mailing list will save a lot of time.

You will likely get a faster response writing to the mailing list than filing a bug in the bug tracker. Most developers check the bug tracker only periodically. If your problem has been determined to be a bug and can not be quickly solved, you may be asked to file a bug in the tracker so the issue doesn't get lost.

14.5 Problems with recent git versions

First make sure you have a clean build and install (see *How to completely remove Matplotlib*), get the latest git update, install it and run a simple test script in debug mode:

```
rm -rf /path/to/site-packages/matplotlib*
git clean -xdf
git pull
python -mpip install -v . > build.out
python -c "from pylab import *; set_loglevel('debug'); plot(); show()" > run.out
```

and post `build.out` and `run.out` to the matplotlib-devel mailing list (please do not post git problems to the users list).

Of course, you will want to clearly describe your problem, what you are expecting and what you are getting, but often a clean build and install will help. See also *Getting help*.

CHAPTER

FIFTEEN

ENVIRONMENT VARIABLES

Contents

`DISPLAY`
: The server and screen on which to place windows. This is interpreted by GUI toolkits in a backend-specific manner, but generally refers to an X.org display name.

`HOME`
: The user's home directory. On Linux, `~` is shorthand for *`HOME`*.

`MPLBACKEND`
: This optional variable can be set to choose the Matplotlib backend. See *What is a backend?*.

`MPLCONFIGDIR`
: This is the directory used to store user customizations to Matplotlib, as well as some caches to improve performance. If *`MPLCONFIGDIR`* is not defined, *`HOME`*`/.config/matplotlib` and *`HOME`*`/.cache/matplotlib` are used on Linux, and *`HOME`*`/.matplotlib` on other platforms, if they are writable. Otherwise, the Python standard library's `tempfile.gettempdir` is used to find a base directory in which the `matplotlib` subdirectory is created.

`PATH`
: The list of directories searched to find executable programs.

`PYTHONPATH`
: The list of directories that are added to Python's standard search list when importing packages and modules.

`QT_API`
: The Python Qt wrapper to prefer when using Qt-based backends. See *the entry in the usage guide* for more information.

15.1 Setting environment variables in Linux and macOS

To list the current value of *PYTHONPATH*, which may be empty, try:

```
echo $PYTHONPATH
```

The procedure for setting environment variables in depends on what your default shell is. Common shells include `bash` and `csh`. You should be able to determine which by running at the command prompt:

```
echo $SHELL
```

To create a new environment variable:

```
export PYTHONPATH=~/Python  # bash/ksh
setenv PYTHONPATH ~/Python  # csh/tcsh
```

To prepend to an existing environment variable:

```
export PATH=~/bin:${PATH}  # bash/ksh
setenv PATH ~/bin:${PATH}  # csh/tcsh
```

The search order may be important to you, do you want `~/bin` to be searched first or last? To append to an existing environment variable:

```
export PATH=${PATH}:~/bin  # bash/ksh
setenv PATH ${PATH}:~/bin  # csh/tcsh
```

To make your changes available in the future, add the commands to your `~/.bashrc`/`.cshrc` file.

15.2 Setting environment variables in Windows

Open the `Control Panel` (*Start* ▸ *Control Panel*), start the `System` program. Click the *Advanced* tab and select the *Environment Variables* button. You can edit or add to the *User Variables*.

Part III

API Overview

CHAPTER

SIXTEEN

API CHANGES

If updating Matplotlib breaks your scripts, this list may help you figure out what caused the breakage and how to fix it by updating your code.

For API changes in older versions see

16.1 Old API Changes

16.1.1 API Changes for 3.3.1

Deprecations

Reverted deprecation of `num2epoch` and `epoch2num`

These two functions were deprecated in 3.3.0, and did not return an accurate Matplotlib datenum relative to the new Matplotlib epoch handling (*`get_epoch`* and `rcParams["date.epoch"]` (default: `'1970-01-01T00:00:00'`)). This version reverts the deprecation.

Functions `epoch2num` and `dates.julian2num` use `date.epoch` rcParam

Now *`epoch2num`* and (undocumented) `julian2num` return floating point days since *`get_epoch`* as set by `rcParams["date.epoch"]` (default: `'1970-01-01T00:00:00'`), instead of floating point days since the old epoch of "0000-12-31T00:00:00". If needed, you can translate from the new to old values as `old = new + mdates.date2num(np.datetime64('0000-12-31'))`

16.1.2 API Changes for 3.3.0

Behaviour changes

`Formatter.fix_minus`

`Formatter.fix_minus` now performs hyphen-to-unicode-minus replacement whenever `rcParams["axes.unicode_minus"]` (default: `True`) is True; i.e. its behavior matches the one of `ScalarFormatter.fix_minus` (*`ScalarFormatter`* now just inherits that implementation).

This replacement is now used by the `format_data_short` method of the various builtin formatter classes, which affects the cursor value in the GUI toolbars.

`FigureCanvasBase` now always has a `manager` attribute, which may be None

Previously, it did not necessarily have such an attribute. A check for `hasattr(figure.canvas, "manager")` should now be replaced by `figure.canvas.manager is not None` (or `getattr(figure.canvas, "manager", None) is not None` for back-compatibility).

`cbook.CallbackRegistry` now propagates exceptions when no GUI event loop is running

`cbook.CallbackRegistry` now defaults to propagating exceptions thrown by callbacks when no interactive GUI event loop is running. If a GUI event loop *is* running, *`cbook.CallbackRegistry`* still defaults to just printing a traceback, as unhandled exceptions can make the program completely `abort()` in that case.

`Axes.locator_params()` validates `axis` parameter

axes.Axes.locator_params used to accept any value for `axis` and silently did nothing, when passed an unsupported value. It now raises a `ValueError`.

`Axis.set_tick_params()` validates `which` parameter

Axis.set_tick_params (and the higher level *axes.Axes.tick_params* and *pyplot.tick_params*) used to accept any value for `which` and silently did nothing, when passed an unsupported value. It now raises a `ValueError`.

`Axis.set_ticklabels()` must match `FixedLocator.locs`

If an axis is using a *ticker.FixedLocator*, typically set by a call to *Axis.set_ticks*, then the number of ticklabels supplied must match the number of locations available (`FixedFormattor.locs`). If not, a `ValueError` is raised.

`backend_pgf.LatexManager.latex`

`backend_pgf.LatexManager.latex` is now created with `encoding="utf-8"`, so its `stdin`, `stdout`, and `stderr` attributes are utf8-encoded.

`pyplot.xticks()` and `pyplot.yticks()`

Previously, passing labels without passing the ticks to either *pyplot.xticks* and *pyplot.yticks* would result in

> TypeError: object of type 'NoneType' has no len()

It now raises a `TypeError` with a proper description of the error.

Setting the same property under multiple aliases now raises a TypeError

Previously, calling e.g. `plot(..., color=somecolor, c=othercolor)` would emit a warning because `color` and `c` actually map to the same Artist property. This now raises a TypeError.

`FileMovieWriter` temporary frames directory

FileMovieWriter now defaults to writing temporary frames in a temporary directory, which is always cleared at exit. In order to keep the individual frames saved on the filesystem, pass an explicit *frame_prefix*.

`Axes.plot` no longer accepts *x* and *y* being both 2D and with different numbers of columns

Previously, calling *Axes.plot* e.g. with *x* of shape `(n, 3)` and *y* of shape `(n, 2)` would plot the first column of *x* against the first column of *y*, the second column of *x* against the second column of *y*, **and** the first column of *x* against the third column of *y*. This now raises an error instead.

`Text.update_from` now copies usetex state from the source Text

`stem` now defaults to `use_line_collection=True`

This creates the stem plot as a *LineCollection* rather than individual *Line2D* objects, greatly improving performance.

rcParams color validator is now stricter

Previously, rcParams entries whose values were color-like accepted "spurious" extra letters or characters in the "middle" of the string, e.g. `"(0, 1a, '0.5')"` would be interpreted as `(0, 1, 0.5)`. These extra characters (including the internal quotes) now cause a ValueError to be raised.

`SymLogNorm` now has a *base* parameter

Previously, *SymLogNorm* had no *base* keyword argument, and defaulted to `base=np.e` whereas the documentation said it was `base=10`. In preparation to make the default 10, calling *SymLogNorm* without the new *base* keyword argument emits a deprecation warning.

`errorbar` now color cycles when only errorbar color is set

Previously setting the *ecolor* would turn off automatic color cycling for the plot, leading to the the lines and markers defaulting to whatever the first color in the color cycle was in the case of multiple plot calls.

`rcsetup.validate_color_for_prop_cycle` now always raises TypeError for bytes input

It previously raised `TypeError`, **except** when the input was of the form `b"C[number]"` in which case it raised a ValueError.

`FigureCanvasPS.print_ps` and `FigureCanvasPS.print_eps` no longer apply edgecolor and facecolor

These methods now assume that the figure edge and facecolor have been correctly applied by *`FigureCanvasBase.print_figure`*, as they are normally called through it.

This behavior is consistent with other figure saving methods (*`FigureCanvasAgg.print_png`*, *`FigureCanvasPdf.print_pdf`*, *`FigureCanvasSVG.print_svg`*).

`pyplot.subplot()` now raises TypeError when given an incorrect number of arguments

This is consistent with other signature mismatch errors. Previously a ValueError was raised.

Shortcut for closing all figures

Shortcuts for closing all figures now also work for the classic toolbar. There is no default shortcut any more because unintentionally closing all figures by a key press might happen too easily. You can configure the shortcut yourself using `rcParams["keymap.quit_all"]` (default: `[]`).

Autoscale for arrow

Calling ax.arrow() will now autoscale the axes.

`set_tick_params(label1On=False)` now also makes the offset text (if any) invisible

... because the offset text can rarely be interpreted without tick labels anyways.

`Axes.annotate` and `pyplot.annotate` parameter name changed

The parameter `s` to *Axes.annotate* and *pyplot.annotate* is renamed to `text`, matching *Annotation*.

The old parameter name remains supported, but support for it will be dropped in a future Matplotlib release.

`font_manager.json_dump` now locks the font manager dump file

... to prevent multiple processes from writing to it at the same time.

`pyplot.rgrids` and `pyplot.thetagrids` now act as setters also when called with only kwargs

Previously, keyword arguments were silently ignored when no positional arguments were given.

`Axis.get_minorticklabels` and `Axis.get_majorticklabels` now returns plain list

Previously, *Axis.get_minorticklabels* and *Axis.get_majorticklabels* returns silent_list. Their return type is now changed to normal list. *get_xminorticklabels*, *get_yminorticklabels*, *get_zminorticklabels*, *Axis.get_ticklabels*, *get_xmajorticklabels*, *get_ymajorticklabels* and *get_zmajorticklabels* methods will be affected by this change.

Default slider formatter

The default method used to format *Slider* values has been changed to use a *ScalarFormatter* adapted the slider values limits. This should ensure that values are displayed with an appropriate number of significant digits even if they are much smaller or much bigger than 1. To restore the old behavior, explicitly pass a "%1.2f" as the *valfmt* parameter to *Slider*.

Add *normalize* keyword argument to `Axes.pie`

`pie()` used to draw a partial pie if the sum of the values was < 1. This behavior is deprecated and will change to always normalizing the values to a full pie by default. If you want to draw a partial pie, please pass `normalize=False` explicitly.

`table.CustomCell` is now an alias for `table.Cell`

All the functionality of `CustomCell` has been moved to its base class *Cell*.

wx Timer interval

Setting the timer interval on a not-yet-started `TimerWx` won't start it anymore.

"step"-type histograms default to the zorder of `Line2D`

This ensures that they go above gridlines by default. The old `zorder` can be kept by passing it as a keyword argument to *Axes.hist*.

`Legend` and `OffsetBox` visibility

Legend and *OffsetBox* subclasses (*PaddedBox*, *AnchoredOffsetbox*, and *AnnotationBbox*) no longer directly keep track of the visibility of their underlying *Patch* artist, but instead pass that flag down to the *Patch*.

`Legend` and `Table` no longer allow invalid locations

This affects legends produced on an Axes (*Axes.legend* and *pyplot.legend*) and on a Figure (*Figure.legend* and *pyplot.figlegend*). Figure legends also no longer accept the unsupported `'best'` location. Previously, invalid Axes locations would use `'best'` and invalid Figure locations would used `'upper right'`.

Passing Line2D's *drawstyle* together with *linestyle* is removed

Instead of `plt.plot(..., linestyle="steps--")`, use `plt.plot(..., linestyle="--", drawstyle="steps")`. `ds` is also an alias for `drawstyle`.

Upper case color strings

Support for passing single-letter colors (one of "rgbcmykw") as UPPERCASE characters is removed; these colors are now case-sensitive (lowercase).

tight/constrained_layout no longer worry about titles that are too wide

tight_layout and *constrained_layout* shrink axes to accommodate "decorations" on the axes. However, if an xlabel or title is too long in the x direction, making the axes smaller in the x-direction doesn't help. The behavior of both has been changed to ignore the width of the title and xlabel and the height of the ylabel in the layout logic.

This also means there is a new keyword argument for *axes.Axes.get_tightbbox* and *axis.Axis.get_tightbbox*: `for_layout_only`, which defaults to *False*, but if *True* returns a bounding box using the rules above.

`rcParams["savefig.facecolor"]` (default: `'auto'`) and `rcParams["savefig.edgecolor"]` (default: `'auto'`) now default to "auto"

This newly allowed value for `rcParams["savefig.facecolor"]` (default: `'auto'`) and `rcParams["savefig.edgecolor"]` (default: `'auto'`), as well as the *facecolor* and *edgecolor* parameters to *Figure.savefig*, means "use whatever facecolor and edgecolor the figure current has".

When using a single dataset, `Axes.hist` no longer wraps the added artist in a `silent_list`

When *Axes.hist* is called with a single dataset, it adds to the axes either a *BarContainer* object (when `histtype="bar"` or `"barstacked"`), or a *Polygon* object (when `histype="step"` or `"stepfilled"`) -- the latter being wrapped in a list-of-one-element. Previously, either artist would be wrapped in a *silent_list*. This is no longer the case: the *BarContainer* is now returned as is (this is an API breaking change if you were directly relying on the concrete `list` API; however, *BarContainer* inherits from `tuple` so most common operations remain available), and the list-of-one *Polygon* is returned as is. This makes the `repr` of the returned artist more accurate: it is now

```
<BarContainer object of of 10 artists>  # "bar", "barstacked"
[<matplotlib.patches.Polygon object at 0xdeadbeef>]  # "step", "stepfilled"
```

instead of

```
<a list of 10 Patch objects>  # "bar", "barstacked"
<a list of 1 Patch objects>  # "step", "stepfilled"
```

When *Axes.hist* is called with multiple artists, it still wraps its return value in a *silent_list*, but uses more accurate type information

```
<a list of 3 BarContainer objects>  # "bar", "barstacked"
<a list of 3 List[Polygon] objects>  # "step", "stepfilled"
```

instead of

```
<a list of 3 Lists of Patches objects>  # "bar", "barstacked"
<a list of 3 Lists of Patches objects>  # "step", "stepfilled"
```

Qt and wx backends no longer create a status bar by default

The coordinates information is now displayed in the toolbar, consistently with the other backends. This is intended to simplify embedding of Matplotlib in larger GUIs, where Matplotlib may control the toolbar but not the status bar.

`rcParams["text.hinting"]` (default: `'force_autohint'`) now supports names mapping to FreeType flags

`rcParams["text.hinting"]` (default: `'force_autohint'`) now supports the values "default", "no_autohint", "force_autohint", and "no_hinting", which directly map to the FreeType flags FT_LOAD_DEFAULT, etc. The old synonyms (respectively "either", "native", "auto", and "none") are still supported, but their use is discouraged. To get normalized values, use *backend_agg.get_hinting_flag*, which returns integer flag values.

`cbook.get_sample_data` auto-loads numpy arrays

When *cbook.get_sample_data* is used to load a npy or npz file and the keyword-only parameter `np_load` is True, the file is automatically loaded using `numpy.load`. `np_load` defaults to False for backwards compatibility, but will become True in a later release.

`get_text_width_height_descent` now checks `ismath` rather than `rcParams["text.usetex"]` (default: `False`)

... to determine whether a string should be passed to the usetex machinery or not. This allows single strings to be marked as not-usetex even when the rcParam is True.

`Axes.vlines`, `Axes.hlines`, `pyplot.vlines` and `pyplot.hlines` *colors* parameter default change

The *colors* parameter will now default to `rcParams["lines.color"]` (default: `'C0'`), while previously it defaulted to 'k'.

Aggressively autoscale clim in `ScalerMappable` classes

Previously some plotting methods would defer autoscaling until the first draw if only one of the *vmin* or *vmax* keyword arguments were passed (*Axes.scatter*, *Axes.hexbin*, *Axes.imshow*, *Axes.pcolorfast*) but would scale based on the passed data if neither was passed (independent of the *norm* keyword arguments). Other methods (*Axes.pcolor*, *Axes.pcolormesh*) always autoscaled base on the initial data.

All of the plotting methods now resolve the unset *vmin* or *vmax* at the initial call time using the data passed in.

If you were relying on exactly one of the *vmin* or *vmax* remaining unset between the time when the method is called and the first time the figure is rendered you get back the old behavior by manually setting the relevant limit back to `None`

```
cm_obj.norm.vmin = None
# or
cm_obj.norm.vmax = None
```

which will be resolved during the draw process.

Deprecations

`figure.add_axes()` without arguments

Calling `fig.add_axes()` with no arguments currently does nothing. This call will raise an error in the future. Adding a free-floating axes needs a position rectangle. If you want a figure-filling single axes, use `add_subplot()` instead.

`backend_wx.DEBUG_MSG`

`backend_wx.DEBUG_MSG` is deprecated. The wx backends now use regular logging.

`Colorbar.config_axis()`

`Colorbar.config_axis()` is considered internal. Its use is deprecated.

`NonUniformImage.is_grayscale` and `PcolorImage.is_grayscale`

These attributes are deprecated, for consistency with `AxesImage.is_grayscale`, which was removed back in Matplotlib 2.0.0. (Note that previously, these attributes were only available *after rendering the image*).

`den` parameter and attribute to `mpl_toolkits.axisartist.angle_helper`

For all locator classes defined in *mpl_toolkits.axisartist.angle_helper*, the `den` parameter has been renamed to `nbins`, and the `den` attribute deprecated in favor of its (pre-existing) synonym `nbins`, for consistency with locator classes defined in *matplotlib.ticker*.

`backend_pgf.LatexManager.latex_stdin_utf8`

`backend_pgf.LatexManager.latex` is now created with `encoding="utf-8"`, so its `stdin` attribute is already utf8-encoded; the `latex_stdin_utf8` attribute is thus deprecated.

Flags containing "U" passed to `cbook.to_filehandle` and `cbook.open_file_cm`

Please remove "U" from flags passed to *cbook.to_filehandle* and *cbook.open_file_cm*. This is consistent with their removal from `open` in Python 3.9.

PDF and PS character tracking internals

The `used_characters` attribute and `track_characters` and `merge_used_characters` methods of *RendererPdf*, *PdfFile*, and *RendererPS* are deprecated.

Case-insensitive capstyles and joinstyles

Please pass capstyles ("miter", "round", "bevel") and joinstyles ("butt", "round", "projecting") as lowercase.

Passing raw data to `register_cmap()`

Passing raw data via parameters *data* and *lut* to *`register_cmap()`* is deprecated. Instead, explicitly create a *`LinearSegmentedColormap`* and pass it via the *cmap* parameter: `register_cmap(cmap=LinearSegmentedColormap(name, data, lut))`.

`DateFormatter.illegal_s`

This attribute is unused and deprecated.

`widgets.TextBox.params_to_disable`

This attribute is deprecated.

Revert deprecation *min, *max keyword arguments to `set_x/y/zlim_3d()`

These keyword arguments were deprecated in 3.0, alongside with the respective parameters in `set_xlim()` / `set_ylim()`. The deprecations of the 2D versions were already reverted in in 3.1.

`cbook.local_over_kwdict`

This function is deprecated. Use *`cbook.normalize_kwargs`* instead.

Passing both singular and plural *colors*, *linewidths*, *linestyles* to `Axes.eventplot`

Passing e.g. both *linewidth* and *linewidths* will raise a TypeError in the future.

Setting `rcParams["text.latex.preamble"]` (default: `''`) or `rcParams["pdf.preamble"]` to non-strings

These rcParams should be set to string values. Support for None (meaning the empty string) and lists of strings (implicitly joined with newlines) is deprecated.

Parameters *norm* and *vmin*/*vmax* should not be used simultaneously

Passing parameters *norm* and *vmin*/*vmax* simultaneously to functions using colormapping such as `scatter()` and `imshow()` is deprecated. Inestead of `norm=LogNorm()`, `vmin=min_val, vmax=max_val` pass `norm=LogNorm(min_val, max_val)`. *vmin* and *vmax* should only be used without setting *norm*.

Effectless parameters of `Figure.colorbar` and `matplotlib.colorbar.Colorbar`

The *cmap* and *norm* parameters of *Figure.colorbar* and *matplotlib.colorbar.Colorbar* have no effect because they are always overridden by the mappable's colormap and norm; they are thus deprecated. Likewise, passing the *alpha*, *boundaries*, *values*, *extend*, or *filled* parameters with a *ContourSet* mappable, or the *alpha* parameter with an *Artist* mappable, is deprecated, as the mappable would likewise override them.

`args_key` and `exec_key` attributes of builtin `MovieWriters`

These attributes are deprecated.

Unused parameters

The following parameters do not have any effect and are deprecated:

- arbitrary keyword arguments to `StreamplotSet`
- parameter *quantize* of *Path.cleaned()*
- parameter *s* of *AnnotationBbox.get_fontsize()*
- parameter *label* of *Tick*

Passing *props* to `Shadow`

The parameter *props* of *Shadow* is deprecated. Use keyword arguments instead.

`Axes.update_datalim_bounds`

This method is deprecated. Use `ax.dataLim.set(Bbox.union([ax.dataLim, bounds]))` instead.

{,Symmetrical}LogScale.{,Inverted}LogTransform

LogScale.LogTransform, LogScale.InvertedLogTransform, SymmetricalScale.SymmetricalTransform and SymmetricalScale.InvertedSymmetricalTransform are deprecated. Directly access the transform classes from the *scale* module.

TexManager.cachedir, TexManager.rgba_arrayd

Use *matplotlib.get_cachedir()* instead for the former; there is no replacement for the latter.

Setting Line2D's pickradius via Line2D.set_picker

Setting a *Line2D*'s pickradius (i.e. the tolerance for pick events and containment checks) via *Line2D.set_picker* is deprecated. Use *Line2D.set_pickradius* instead.

Line2D.set_picker no longer sets the artist's custom-contain() check.

Artist.set_contains, Artist.get_contains

Setting a custom method overriding *Artist.contains* is deprecated. There is no replacement, but you may still customize pick events using *Artist.set_picker*.

Colorbar methods

The on_mappable_changed and update_bruteforce methods of *Colorbar* are deprecated; both can be replaced by calls to *update_normal*.

OldScalarFormatter, IndexFormatter and DateIndexFormatter

These formatters are deprecated. Their functionality can be implemented using e.g. *FuncFormatter*.

OldAutoLocator

This ticker is deprecated.

required, *forbidden* and *allowed* parameters of cbook.normalize_kwargs

These parameters are deprecated.

The TTFPATH and AFMPATH environment variables

Support for the (undocumented) TTFPATH and AFMPATH environment variables is deprecated. Additional fonts may be registered using matplotlib.font_manager.fontManager.addfont().

matplotlib.compat

This module is deprecated.

matplotlib.backends.qt_editor.formsubplottool

This module is deprecated. Use matplotlib.backends.backend_qt5.SubplotToolQt instead.

AVConv animation writer deprecated

The AVConvBase, AVConvWriter and AVConvFileWriter classes, and the associated animation.avconv_path and animation.avconv_args rcParams are deprecated.

Debian 8 (2015, EOL 06/2020) and Ubuntu 14.04 (EOL 04/2019) were the last versions of Debian and Ubuntu to ship avconv. It remains possible to force the use of avconv by using the ffmpeg-based writers with rcParams["animation.ffmpeg_path"] (default: 'ffmpeg') set to "avconv".

log/symlog scale base, ticks, and nonpos specification

semilogx, *semilogy*, *loglog*, *LogScale*, and *SymmetricalLogScale* used to take keyword arguments that depends on the axis orientation ("basex" vs "basey", "subsx" vs "subsy", "nonposx" vs "nonposy"); these parameter names are now deprecated in favor of "base", "subs", "nonpositive". This deprecation also affects e.g. `ax.set_yscale("log", basey=...)` which must now be spelled `ax.set_yscale("log", base=...)`.

The change from "nonpos" to "nonpositive" also affects *LogTransform*, *InvertedLogTransform*, *SymmetricalLogTransform*, etc.

To use *different* bases for the x-axis and y-axis of a *loglog* plot, use e.g. `ax.set_xscale("log", base=10); ax.set_yscale("log", base=2)`.

`DraggableBase.artist_picker`

This method is deprecated. If you previously reimplemented it in a subclass, set the artist's picker instead with *Artist.set_picker*.

clear_temp parameter and attribute of `FileMovieWriter`

The *clear_temp* parameter and attribute of *FileMovieWriter* is deprecated. In the future, files placed in a temporary directory (using `frame_prefix=None`, the default) will be cleared; files placed elsewhere will not.

Deprecated rcParams validators

The following validators, defined in *rcsetup*, are deprecated: `validate_fontset`, `validate_mathtext_default`, `validate_alignment`, `validate_svg_fontset`, `validate_pgf_texsystem`, `validate_movie_frame_fmt`, `validate_axis_locator`, `validate_movie_html_fmt`, `validate_grid_axis`, `validate_axes_titlelocation`, `validate_toolbar`, `validate_ps_papersize`, `validate_legend_loc`, `validate_bool_maybe_none`, `validate_hinting`, `validate_movie_writers`, `validate_webagg_address`, `validate_nseq_float`, `validate_nseq_int`. To test whether an rcParam value would be acceptable, one can test e.g. `rc = RcParams(); rc[k] = v` raises an exception.

Stricter rcParam validation

`rcParams["axes.axisbelow"]` (default: `'line'`) currently normalizes all strings starting with "line" (case-insensitive) to the option "line". This is deprecated; in a future version only the exact string "line" (case-sensitive) will be supported.

`add_subplot()` validates its inputs

In particular, for `add_subplot(rows, cols, index)`, all parameters must be integral. Previously strings and floats were accepted and converted to int. This will now emit a deprecation warning.

Toggling axes navigation from the keyboard using "a" and digit keys

Axes navigation can still be toggled programmatically using *Axes.set_navigate*.

The following related APIs are also deprecated: `backend_tools.ToolEnableAllNavigation`, `backend_tools.ToolEnableNavigation`, and `rcParams["keymap.all_axes"]`.

`matplotlib.test(recursionlimit=...)`

The *recursionlimit* parameter of `matplotlib.test` is deprecated.

mathtext glues

The *copy* parameter of `mathtext.Glue` is deprecated (the underlying glue spec is now immutable). `mathtext.GlueSpec` is deprecated.

Signatures of `Artist.draw` and `Axes.draw`

The *inframe* parameter to *Axes.draw* is deprecated. Use *Axes.redraw_in_frame* instead.

Not passing the *renderer* parameter to *Axes.draw* is deprecated. Use `axes.draw_artist(axes)` instead.

These changes make the signature of the `draw` (`artist.draw(renderer)`) method consistent across all artists; thus, additional parameters to *Artist.draw* are deprecated.

DraggableBase.on_motion_blit

This method is deprecated. *DraggableBase.on_motion* now handles both the blitting and the non-blitting cases.

Passing the dash offset as None

Fine control of dash patterns can be achieved by passing an `(offset, (on-length, off-length, on-length, off-length, ...))` pair as the linestyle property of *Line2D* and *LineCollection*. Previously, certain APIs would accept `offset = None` as a synonym for `offset = 0`, but this was never universally implemented, e.g. for vector output. Support for `offset = None` is deprecated, set the offset to 0 instead.

RendererCairo.fontweights, RendererCairo.fontangles

... are deprecated.

autofmt_xdate(which=None)

This is deprecated, use its more explicit synonym, `which="major"`, instead.

JPEG options

The *quality*, *optimize*, and *progressive* keyword arguments to *savefig*, which were only used when saving to JPEG, are deprecated. `rcParams["savefig.jpeg_quality"]` (default: `95`) is likewise deprecated.

Such options should now be directly passed to Pillow using `savefig(..., pil_kwargs={"quality": ..., "optimize": ..., "progressive": ...})`.

dviread.Encoding

This class was (mostly) broken and is deprecated.

Axis and Locator `pan` and `zoom`

The unused `pan` and `zoom` methods of *Axis* and *Locator* are deprecated. Panning and zooming are now implemented using the `start_pan`, `drag_pan`, and `end_pan` methods of *Axes*.

Passing None to various Axes subclass factories

Support for passing `None` as base class to *axes.subplot_class_factory*, `axes_grid1.parasite_axes.host_axes_class_factory`, `axes_grid1.parasite_axes.host_subplot_class_factory`, `axes_grid1.parasite_axes.parasite_axes_class_factory`, and `axes_grid1.parasite_axes.parasite_axes_auxtrans_class_factory` is deprecated. Explicitly pass the correct base `Axes` class instead.

axes_rgb

In *mpl_toolkits.axes_grid1.axes_rgb*, `imshow_rgb` is deprecated (use `ax.imshow(np.dstack([r, g, b]))` instead); `RGBAxesBase` is deprecated (use `RGBAxes` instead); `RGBAxes.add_RGB_to_figure` is deprecated (it was an internal helper).

Substitution.from_params

This method is deprecated. If needed, directly assign to the `params` attribute of the Substitution object.

PGF backend cleanups

The *dummy* parameter of *RendererPgf* is deprecated.

GraphicsContextPgf is deprecated (use *GraphicsContextBase* instead).

`set_factor` method of `mpl_toolkits.axisartist` locators

The `set_factor` method of *mpl_toolkits.axisartist* locators (which are different from "standard" Matplotlib tick locators) is deprecated.

`widgets.SubplotTool` callbacks and axes

The `funcleft`, `funcright`, `funcbottom`, `functop`, `funcwspace`, and `funchspace` methods of *`widgets.SubplotTool`* are deprecated.

The `axleft`, `axright`, `axbottom`, `axtop`, `axwspace`, and `axhspace` attributes of *`widgets.SubplotTool`* are deprecated. Access the `ax` attribute of the corresponding slider, if needed.

mathtext `Glue` helper classes

The `Fil`, `Fill`, `Filll`, `NegFil`, `NegFill`, `NegFilll`, and `SsGlue` classes in the *`matplotlib.mathtext`* module are deprecated. As an alternative, directly construct glue instances with `Glue("fil")`, etc.

NavigationToolbar2._init_toolbar

Overriding this method to initialize third-party toolbars is deprecated. Instead, the toolbar should be initialized in the `__init__` method of the subclass (which should call the base-class' `__init__` as appropriate). To keep back-compatibility with earlier versions of Matplotlib (which *required* `_init_toolbar` to be overridden), a fully empty implementation (`def _init_toolbar(self): pass`) may be kept and will not trigger the deprecation warning.

NavigationToolbar2QT.parent and .basedir

These attributes are deprecated. In order to access the parent window, use `toolbar.canvas.parent()`. Once the deprecation period is elapsed, it will also be accessible as `toolbar.parent()`. The base directory to the icons is `os.path.join(mpl.get_data_path(), "images")`.

NavigationToolbar2QT.ctx

This attribute is deprecated.

NavigationToolbar2Wx attributes

The `prevZoomRect`, `retinaFix`, `savedRetinaImage`, `wxoverlay`, `zoomAxes`, `zoomStartX`, and `zoomStartY` attributes are deprecated.

NavigationToolbar2.press and .release

These methods were called when pressing or releasing a mouse button, but *only* when an interactive pan or zoom was occurring (contrary to what the docs stated). They are deprecated; if you write a backend which needs to customize such events, please directly override `press_pan`/`press_zoom`/`release_pan`/`release_zoom` instead.

FigureCanvasGTK3._renderer_init

Overriding this method to initialize renderers for GTK3 canvases is deprecated. Instead, the renderer should be initialized in the `__init__` method of the subclass (which should call the base-class' `__init__` as appropriate). To keep back-compatibility with earlier versions of Matplotlib (which *required* `_renderer_init` to be overridden), a fully empty implementation (`def _renderer_init(self): pass`) may be kept and will not trigger the deprecation warning.

Path helpers in `bezier`

`bezier.make_path_regular` is deprecated. Use `Path.cleaned()` (or `Path.cleaned(curves=True)`, etc.) instead (but note that these methods add a `STOP` code at the end of the path).

`bezier.concatenate_paths` is deprecated. Use `Path.make_compound_path()` instead.

`animation.html_args` rcParam

The unused `animation.html_args` rcParam and `animation.HTMLWriter.args_key` attribute are deprecated.

`text.latex.preview` rcParam

This rcParam, which controlled the use of the preview.sty LaTeX package to align TeX string baselines, is deprecated, as Matplotlib's own dvi parser now computes baselines just as well as preview.sty.

`SubplotSpec.get_rows_columns`

This method is deprecated. Use the `GridSpec.nrows`, `GridSpec.ncols`, `SubplotSpec.rowspan`, and `SubplotSpec.colspan` properties instead.

Qt4-based backends

The qt4agg and qt4cairo backends are deprecated. Qt4 has reached its end-of-life in 2015 and there are no releases for recent versions of Python. Please consider switching to Qt5.

fontdict and *minor* parameters of `Axes.set_xticklabels` and `Axes.set_yticklabels` will become keyword-only

All parameters of `Figure.subplots` except *nrows* and *ncols* will become keyword-only

This avoids typing e.g. `subplots(1, 1, 1)` when meaning `subplot(1, 1, 1)`, but actually getting `subplots(1, 1, sharex=1)`.

`RendererWx.get_gc`

This method is deprecated. Access the `gc` attribute directly instead.

add_all parameter in `axes_grid`

The *add_all* parameter of *axes_grid1.axes_grid.Grid*, *axes_grid1.axes_grid.ImageGrid*, *axes_grid1.axes_rgb.make_rgb_axes* and *axes_grid1.axes_rgb.RGBAxes* is deprecated. Axes are now always added to the parent figure, though they can be later removed with `ax.remove()`.

BboxBase.inverse_transformed

.BboxBase.inverse_transformed is deprecated (call *BboxBase.transformed* on the *inverted()* transform instead).

orientation of eventplot() and EventCollection

Setting the *orientation* of an eventplot() or *EventCollection* to "none" or None is deprecated; set it to "horizontal" instead. Moreover, the two orientations ("horizontal" and "vertical") will become case-sensitive in the future.

minor kwarg to Axis.get_ticklocs will become keyword-only

Passing this argument positionally is deprecated.

Case-insensitive properties

Normalization of upper or mixed-case property names to lowercase in *Artist.set* and *Artist.update* is deprecated. In the future, property names will be passed as is, allowing one to pass names such as *patchA* or *UVC*.

ContourSet.ax, Quiver.ax

These attributes are deprecated in favor of ContourSet.axes and Quiver.axes, for consistency with other artists.

Locator.refresh() and associated methods

Locator.refresh() is deprecated. This method was called at certain places to let locators update their internal state, typically based on the axis limits. Locators should now always consult the axis limits when called, if needed.

The associated helper methods NavigationToolbar2.draw() and ToolViewsPositions.refresh_locators() are deprecated, and should be replaced by calls to draw_idle() on the corresponding canvas.

`ScalarMappable` checkers

The `add_checker` and `check_update` methods and `update_dict` attribute of *`ScalarMappable`* are deprecated.

`pyplot.tight_layout` and `ColorbarBase` parameters will become keyword-only

All parameters of *`pyplot.tight_layout`* and all parameters of `ColorbarBase` except for the first (*ax*) will become keyword-only, consistently with *`Figure.tight_layout`* and `Colorbar`, respectively.

`Axes.pie` radius and startangle

Passing `None` as either the `radius` or `startangle` of an *`Axes.pie`* is deprecated; use the explicit defaults of 1 and 0, respectively, instead.

`AxisArtist.dpi_transform`

... is deprecated. Scale `Figure.dpi_scale_trans` by 1/72 to achieve the same effect.

`offset_position` property of `Collection`

The `offset_position` property of *`Collection`* is deprecated. In the future, *`Collection`*s will always behave as if `offset_position` is set to "screen" (the default).

Support for passing `offset_position="data"` to the `draw_path_collection` of all renderer classes is deprecated.

`transforms.AffineDeltaTransform` can be used as a replacement. This API is experimental and may change in the future.

`testing.compare.make_external_conversion_command`

... is deprecated.

`epoch2num` and `num2epoch` are deprecated

These are unused and can be easily reproduced by other date tools. *`get_epoch`* will return Matplotlib's epoch.

`axes_grid1.CbarAxes` attributes

The `cbid` and `locator` attribute are deprecated. Use `mappable.colorbar_cid` and `colorbar.locator`, as for standard colorbars.

`qt_compat.is_pyqt5`

This function is deprecated in prevision of the future release of PyQt6. The Qt version can be checked using `QtCore.qVersion()`.

Reordering of parameters by `Artist.set`

In a future version, `Artist.set` will apply artist properties in the order in which they are given. This only affects the interaction between the *color*, *edgecolor*, *facecolor*, and, for *`Collection`*s, *alpha* properties: the *color* property now needs to be passed first in order not to override the other properties. This is consistent with e.g. *`Artist.update`*, which did not reorder the properties passed to it.

Passing multiple keys as a single comma-separated string or multiple arguments to `ToolManager.update_keymap`

This is deprecated; pass keys as a list of strings instead.

Statusbar classes and attributes

The `statusbar` attribute of *`FigureManagerBase`*, *`StatusbarBase`* and all its subclasses, and `StatusBarWx`, are deprecated, as messages are now displayed in the toolbar instead.

`ismath` parameter of `draw_tex`

The `ismath` parameter of the `draw_tex` method of all renderer classes is deprecated (as a call to `draw_tex` -- not to be confused with `draw_text`! -- means that the entire string should be passed to the `usetex` machinery anyways). Likewise, the text machinery will no longer pass the `ismath` parameter when calling `draw_tex` (this should only matter for backend implementers).

Passing `ismath="TeX!"` to *`RendererAgg.get_text_width_height_descent`* is deprecated. Pass `ismath="TeX"` instead, consistently with other low-level APIs which support the values True, False, and "TeX" for `ismath`.

`matplotlib.ttconv`

This module is deprecated.

Stricter PDF metadata keys in PGF

Saving metadata in PDF with the PGF backend currently normalizes all keys to lowercase, unlike the PDF backend, which only accepts the canonical case. This is deprecated; in a future version, only the canonically cased keys listed in the PDF specification (and the *`PdfPages`* documentation) will be accepted.

Qt modifier keys

The `MODIFIER_KEYS`, `SUPER`, `ALT`, `CTRL`, and `SHIFT` global variables of the `matplotlib.backends.backend_qt4agg`, `matplotlib.backends.backend_qt4cairo`, `matplotlib.backends.backend_qt5agg` and `matplotlib.backends.backend_qt5cairo` modules are deprecated.

`TexManager`

The `TexManager.serif`, `TexManager.sans_serif`, `TexManager.cursive` and `TexManager.monospace` attributes are deprecated.

Removals

The following deprecated APIs have been removed:

Modules

- `backends.qt_editor.formlayout` (use the formlayout module available on PyPI instead).

Classes, methods and attributes

- `artist.Artist.aname` property (no replacement)
- `axis.Axis.iter_ticks` (no replacement)
- Support for custom backends that do not provide a `backend_bases.GraphicsContextBase.set_hatch_color` method
- `backend_bases.RendererBase.strip_math()` (use `cbook.strip_math()` instead)
- `backend_wx.debug_on_error()` (no replacement)
- `backend_wx.raise_msg_to_str()` (no replacement)
- `backend_wx.fake_stderr` (no replacement)
- `backend_wx.MenuButtonWx` (no replacement)
- `backend_wx.PrintoutWx` (no replacement)
- `_backend_tk.NavigationToolbar2Tk.set_active()` (no replacement)
- `backend_ps.PsBackendHelper.gs_exe` property (no replacement)
- `backend_ps.PsBackendHelper.gs_version` property (no replacement)
- `backend_ps.PsBackendHelper.supports_ps2write` property (no replacement)
- `backend_ps.RendererPS.afmfontd` property (no replacement)
- `backend_ps.GraphicsContextPS.shouldstroke` property (no replacement)
- `backend_gtk3.FileChooserDialog` (no replacement)
- `backend_gtk3.SaveFigureGTK3.get_filechooser()` (no replacement)
- `backend_gtk3.NavigationToolbar2GTK3.get_filechooser()` (no replacement)
- `backend_gtk3cairo.FigureManagerGTK3Cairo` (use `backend_gtk3.FigureManagerGTK3` instead)
- `backend_pdf.RendererPdf.afm_font_cache` property (no replacement)
- `backend_pgf.LatexManagerFactory` (no replacement)
- `backend_qt5.NavigationToolbar2QT.buttons` property (no replacement)

- `backend_qt5.NavigationToolbar2QT.adj_window` property (no replacement)
- `bezier.find_r_to_boundary_of_closedpath()` (no replacement)
- `cbook.dedent()` (use `inspect.cleandoc` instead)
- `cbook.get_label()` (no replacement)
- `cbook.is_hashable()` (use `isinstance(..., collections.abc.Hashable)` instead)
- `cbook.iterable()` (use `numpy.iterable()` instead)
- `cbook.safezip()` (no replacement)
- `colorbar.ColorbarBase.get_cmap` (use `ScalarMappable.get_cmap` instead)
- `colorbar.ColorbarBase.set_cmap` (use `ScalarMappable.set_cmap` instead)
- `colorbar.ColorbarBase.get_clim` (use `ScalarMappable.get_clim` instead)
- `colorbar.ColorbarBase.set_clim` (use `ScalarMappable.set_clim` instead)
- `colorbar.ColorbarBase.set_norm` (use `ScalarMappable.set_norm` instead)
- `dates.seconds()` (no replacement)
- `dates.minutes()` (no replacement)
- `dates.hours()` (no replacement)
- `dates.weeks()` (no replacement)
- `dates.strpdate2num` and `dates.bytespdate2num` (use `time.strptime` or `dateutil.parser.parse` or *`dates.datestr2num`* instead)
- `docstring.Appender` (no replacement)
- `docstring.dedent()` (use `inspect.getdoc` instead)
- `docstring.copy_dedent()` (use `docstring.copy()` and `inspect.getdoc` instead)
- `font_manager.OSXInstalledFonts()` (no replacement)
- `image.BboxImage.interp_at_native` property (no replacement)
- `lines.Line2D.verticalOffset` property (no replacement)
- `matplotlib.checkdep_dvipng` (no replacement)
- `matplotlib.checkdep_ghostscript` (no replacement)
- `matplotlib.checkdep_pdftops` (no replacement)
- `matplotlib.checkdep_inkscape` (no replacement)
- `matplotlib.get_py2exe_datafiles` (no replacement)
- `matplotlib.tk_window_focus` (use `rcParams['tk.window_focus']` instead)
- `mlab.demean()` (use `mlab.detrend_mean()` instead)
- `path.get_paths_extents()` (use `path.get_path_collection_extents()` instead)

- `path.Path.has_nonfinite()` (use `not np.isfinite(self.vertices).all()` instead)
- `projections.process_projection_requirements()` (no replacement)
- `pyplot.plotfile()` (Instead, load the data using `pandas.read_csv` or `numpy.loadtxt` or similar and use regular pyplot functions to plot the loaded data.)
- `quiver.Quiver.color()` (use `Quiver.get_facecolor()` instead)
- `quiver.Quiver.keyvec` property (no replacement)
- `quiver.Quiver.keytext` property (no replacement)
- `rcsetup.validate_qt4()` (no replacement)
- `rcsetup.validate_qt5()` (no replacement)
- `rcsetup.validate_verbose()` (no replacement)
- `rcsetup.ValidateInterval` (no replacement)
- `scale.LogTransformBase` (use `scale.LogTransform` instead)
- `scale.InvertedLogTransformBase` (use `scale.InvertedLogTransform` instead)
- `scale.Log10Transform` (use `scale.LogTransform` instead)
- `scale.InvertedLog10Transform` (use `scale.InvertedLogTransform` instead)
- `scale.Log2Transform` (use `scale.LogTransform` instead)
- `scale.InvertedLog2Transform` (use `scale.InvertedLogTransform` instead)
- `scale.NaturalLogTransform` (use `scale.LogTransform` instead)
- `scale.InvertedNaturalLogTransform` (use `scale.InvertedLogTransform` instead)
- `scale.get_scale_docs()` (no replacement)
- `sphinxext.plot_directive.plot_directive()` (use the class `PlotDirective` instead)
- `sphinxext.mathmpl.math_directive()` (use the class `MathDirective` instead)
- `spines.Spine.is_frame_like()` (no replacement)
- `testing.decorators.switch_backend()` (use `@pytest.mark.backend` decorator instead)
- `text.Text.is_math_text()` (use `cbook.is_math_text()` instead)
- `text.TextWithDash()` (use `text.Annotation` instead)
- `textpath.TextPath.is_math_text()` (use `cbook.is_math_text()` instead)
- `textpath.TextPath.text_get_vertices_codes()` (use `textpath.text_to_path.get_text_path()` instead)
- `textpath.TextToPath.glyph_to_path()` (use `font.get_path()` and manual translation of the vertices instead)
- `ticker.OldScalarFormatter.pprint_val()` (no replacement)

- `ticker.ScalarFormatter.pprint_val()` (no replacement)
- `ticker.LogFormatter.pprint_val()` (no replacement)
- `ticker.decade_down()` (no replacement)
- `ticker.decade_up()` (no replacement)
- `Tick` properties `gridOn`, `tick1On`, `tick2On`, `label1On`, `label2On` (use `set_visible()` / `get_visible()` on `Tick.gridline`, `Tick.tick1line`, `Tick.tick2line`, `Tick.label1`, `Tick.label2` instead)
- `widgets.SpanSelector.buttonDown` property (no replacement)
- `mplot3d.proj3d.line2d()` (no replacement)
- `mplot3d.proj3d.line2d_dist()` (no replacement)
- `mplot3d.proj3d.line2d_seg_dist()` (no replacement)
- `mplot3d.proj3d.mod()` (use `numpy.linalg.norm` instead)
- `mplot3d.proj3d.proj_transform_vec()` (no replacement)
- `mplot3d.proj3d.proj_transform_vec_clip()` (no replacement)
- `mplot3d.proj3d.vec_pad_ones()` (no replacement)
- `mplot3d.proj3d.proj_trans_clip_points()` (no replacement)
- `mplot3d.art3d.norm_angle()` (no replacement)
- `mplot3d.art3d.norm_text_angle()` (no replacement)
- `mplot3d.art3d.path_to_3d_segment()` (no replacement)
- `mplot3d.art3d.paths_to_3d_segments()` (no replacement)
- `mplot3d.art3d.path_to_3d_segment_with_codes()` (no replacement)
- `mplot3d.art3d.paths_to_3d_segments_with_codes()` (no replacement)
- `mplot3d.art3d.get_patch_verts()` (no replacement)
- `mplot3d.art3d.get_colors()` (no replacement)
- `mplot3d.art3d.zalpha()` (no replacement)
- `mplot3d.axis3d.get_flip_min_max()` (no replacement)
- `mplot3d.axis3d.Axis.get_tick_positions()` (no replacement)
- `axisartist.axis_artist.UnimplementedException` (no replacement)
- `axisartist.axislines.SimpleChainedObjects` (use `axis_grid1.mpl_axes.SimpleChainedObjects` instead)
- `axisartist.axislines.Axes.AxisDict` (use `axis_grid1.mpl_axes.Axes.AxisDict` instead)

Arguments

- `Axes.text()` / `pyplot.text()` do not support the parameter `withdash` anymore. Use `Axes.annotate()` and `pyplot.annotate()` instead.
- The first parameter of *matplotlib.use* has been renamed from `arg` to `backend` (only relevant if you pass by keyword).
- The parameter `warn` of *matplotlib.use* has been removed. A failure to switch the backend will now always raise an `ImportError` if `force` is set; catch that error if necessary.
- All parameters of *matplotlib.use* except the first one are now keyword-only.
- The unused parameters `shape` and `imlim` of *imshow()* are now removed. All parameters beyond `extent` are now keyword-only.
- The unused parameter `interp_at_native` of *BboxImage* has been removed.
- The parameter `usetex` of *TextToPath.get_text_path* has been removed. Use `ismath='TeX'` instead.
- The parameter `block` of `show()` is now keyword-only, and arbitrary arguments or keyword arguments are no longer accepted.
- The parameter `frameon` of *Figure.savefig* has been removed. Use `facecolor="none"` to get a transparent background.
- Passing a `wx.EvtHandler` as the first argument to `backend_wx.TimerWx` is not supported anymore; the signature of `TimerWx` is now consistent with *TimerBase*.
- The `manage_xticks` parameter of *boxplot* and *bxp* has been renamed to `manage_ticks`.
- The `normed` parameter of *hist2d* has been renamed to `density`.
- The `s` parameter of *Annotation* has been renamed to `text`.
- For all functions in *bezier* that supported a `tolerence` parameter, this parameter has been renamed to `tolerance`.
- `axis("normal")` is not supported anymore. Use the equivalent `axis("auto")` instead.
- `axis()` does not accept arbitrary keyword arguments anymore.
- `Axis.set_ticklabels()` does not accept arbitrary positional arguments other than `ticklabels`.
- `mpl_toolkits.mplot3d.art3d.Poly3DCollection.set_zsort` does not accept the value `True` anymore. Pass the equivalent value 'average' instead.
- *AnchoredText* no longer accepts `horizontalalignment` or `verticalalignment` keyword arguments.
- *ConnectionPatch* no longer accepts the `arrow_transmuter` and `connector` keyword arguments, which did nothing since 3.0.

- *FancyArrowPatch* no longer accepts the `arrow_transmuter` and `connector` keyword arguments, which did nothing since 3.0.
- *TextPath* no longer accepts arbitrary positional or keyword arguments.
- *MaxNLocator.set_params()* no longer accepts arbitrary keyword arguments.
- *pie* no longer accepts and squeezes non-1D inputs; pass 1D input to the `x` argument.
- Passing (n, 1)-shaped error arrays to *Axes.errorbar()* is no longer supported; pass a 1D array instead.

rcParams

- The `text.latex.unicode` rcParam has been removed, with no replacement. Matplotlib now always supports unicode in usetex.
- The `savefig.frameon` rcParam has been removed. Set `rcParams["savefig.facecolor"]` (default: `'auto'`) to "none" to get a transparent background.
- The `pgf.debug`, `verbose.fileo` and `verbose.verbose.level` rcParams, which had no effect, have been removed.
- Support for setting `rcParams["mathtext.default"]` (default: `'it'`) to "circled" has been removed.

Environment variables

- `MATPLOTLIBDATA` (no replacement).

mathtext

- The `\stackrel` command (which behaved differently from its LaTeX version) has been removed. Use `\genfrac` instead.
- The `\mathcircled` command has been removed. Directly use Unicode characters, such as `'\N{CIRCLED LATIN CAPITAL LETTER A}'`, instead.

Development changes

Matplotlib now requires numpy>=1.15

Matplotlib now uses Pillow to save and read pngs

The builtin png encoder and decoder has been removed, and Pillow is now a dependency. Note that when reading 16-bit RGB(A) images, Pillow truncates them to 8-bit precision, whereas the old builtin decoder kept the full precision.

The deprecated wx backend (not wxagg!) now always uses wx's builtin jpeg and tiff support rather than relying on Pillow for writing these formats; this behavior is consistent with wx's png output.

16.1.3 API Changes for 3.2.0

Behavior changes

Reduced default value of `rcParams["axes.formatter.limits"]` (default: `[-5, 6]`)

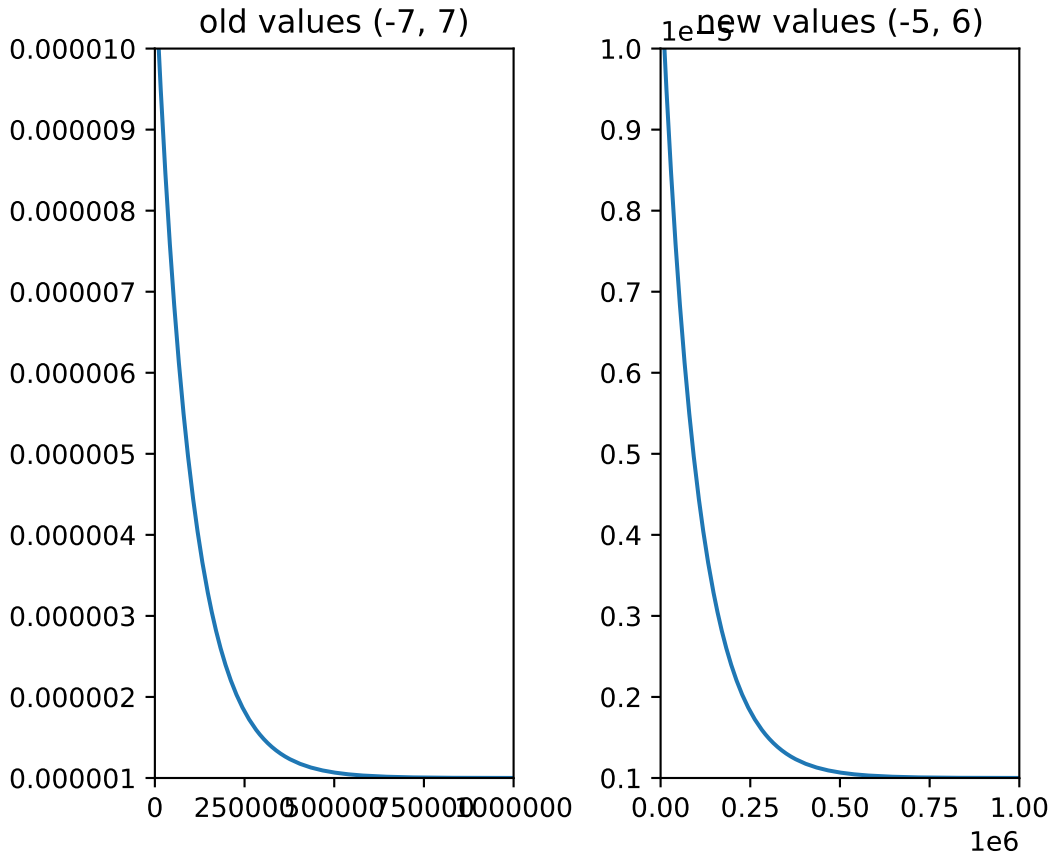
Changed the default value of `rcParams["axes.formatter.limits"]` (default: `[-5, 6]`) from -7, 7 to -5, 6 for better readability.

`matplotlib.colorbar.Colorbar` uses un-normalized axes for all mappables

Before 3.0, *`matplotlib.colorbar.Colorbar`* (*`colorbar`*) normalized all axes limits between 0 and 1 and had custom tickers to handle the labelling of the colorbar ticks. After 3.0, colorbars constructed from mappables that were *not* contours were constructed with axes that had limits between `vmin` and `vmax` of the mappable's norm, and the tickers were made children of the normal axes tickers.

This version of Matplotlib extends that to mappables made by contours, and allows the axes to run between the lowest boundary in the contour and the highest.

Code that worked around the normalization between 0 and 1 will need to be modified.



`MovieWriterRegistry`

`MovieWriterRegistry` now always checks the availability of the writer classes before returning them. If one wishes, for example, to get the first available writer, without performing the availability check on subsequent writers, it is now possible to iterate over the registry, which will yield the names of the available classes.

Autoscaling

Matplotlib used to recompute autoscaled limits after every plotting (`plot()`, `bar()`, etc.) call. It now only does so when actually rendering the canvas, or when the user queries the Axes limits. This is a major performance improvement for plots with a large number of artists.

In particular, this means that artists added manually with *`Axes.add_line`*, *`Axes.add_patch`*, etc. will be taken into account by the autoscale, even without an explicit call to *`Axes.autoscale_view`*.

In some cases, this can result in different limits being reported. If this is an issue, consider triggering a draw with `fig.canvas.draw()`.

Autoscaling has also changed for artists that are based on the *`Collection`* class. Pre-

viously, the method that calculates the automatic limits *Collection.get_datalim* tried to take into account the size of objects in the collection and make the limits large enough to not clip any of the object, i.e., for *Axes.scatter* it would make the limits large enough to not clip any markers in the scatter. This is problematic when the object size is specified in physical space, or figure-relative space, because the transform from physical units to data limits requires knowing the data limits, and becomes invalid when the new limits are applied. This is an inverse problem that is theoretically solvable (if the object is physically smaller than the axes), but the extra complexity was not deemed worth it, particularly as the most common use case is for markers in scatter that are usually small enough to be accommodated by the default data limit margins.

While the new behavior is algorithmically simpler, it is conditional on properties of the *Collection* object:

1. `offsets = None`, `transform` is a child of `Axes.transData`: use the paths for the automatic limits (i.e. for *LineCollection* in *Axes.streamplot*).
2. `offsets != None`, and `offset_transform` is child of `Axes.transData`:

a) **`transform` is child of `Axes.transData`: use the `path + offset` for** limits (i.e., for *Axes.bar*).

b) **`transform` is not a child of `Axes.transData`: just use the offsets** for the limits (i.e. for scatter)

3. otherwise return a null *Bbox*.

While this seems complicated, the logic is simply to use the information from the object that are in data space for the limits, but not information that is in physical units.

log-scale bar() / hist() autolimits

The autolimits computation in *bar* and *hist* when the axes already uses log-scale has changed to match the computation when the axes is switched to log-scale after the call to *bar* and *hist*, and when calling `bar(..., log=True)` / `hist(..., log=True)`: if there are at least two different bar heights, add the normal axes margins to them (in log-scale); if there is only a single bar height, expand the axes limits by one order of magnitude around it and then apply axes margins.

Axes labels spanning multiple rows/columns

`Axes.label_outer` now correctly keep the x labels and tick labels visible for Axes spanning multiple rows, as long as they cover the last row of the Axes grid. (This is consistent with keeping the y labels and tick labels visible for Axes spanning multiple columns as long as they cover the first column of the Axes grid.)

The `Axes.is_last_row` and `Axes.is_last_col` methods now correctly return True for Axes spanning multiple rows, as long as they cover the last row or column respectively. Again this is consistent with the behavior for axes covering the first row or column.

The `Axes.rowNum` and `Axes.colNum` attributes are deprecated, as they only refer to the first grid cell covered by the Axes. Instead, use the new `ax.get_subplotspec().rowspan` and `ax.get_subplotspec().colspan` properties, which are `range` objects indicating the whole span of rows and columns covered by the subplot.

(Note that all methods and attributes mentioned here actually only exist on the `Subplot` subclass of *Axes*, which is used for grid-positioned Axes but not for Axes positioned directly in absolute coordinates.)

The *GridSpec* class gained the `nrows` and `ncols` properties as more explicit synonyms for the parameters returned by `GridSpec.get_geometry`.

Locators

When more than *Locator.MAXTICKS* ticks are generated, the behavior of *Locator.raise_if_exceeds* changed from raising a RuntimeError to emitting a log at WARNING level.

nonsingular Locators

`Locator.nonsingular` (introduced in mpl 3.1), `DateLocator.nonsingular`, and `AutoDateLocator.nonsingular` now returns a range `v0, v1` with `v0 <= v1`. This behavior is consistent with the implementation of `nonsingular` by the `LogLocator` and `LogitLocator` subclasses.

get_data_ratio

`Axes.get_data_ratio` now takes the axes scale into account (linear, log, logit, etc.) before computing the y-to-x ratio. This change allows fixed aspects to be applied to any combination of x and y scales.

Artist sticky edges

Previously, the `sticky_edges` attribute of artists was a list of values such that if an axis limit coincides with a sticky edge, it would not be expanded by the axes margins (this is the mechanism that e.g. prevents margins from being added around images).

`sticky_edges` now have an additional effect on margins application: even if an axis limit did not coincide with a sticky edge, it cannot *cross* a sticky edge through margin application -- instead, the margins will only expand the axis limit until it bumps against the sticky edge.

This change improves the margins of axes displaying a *`streamplot`*:

- if the streamplot goes all the way to the edges of the vector field, then the axis limits are set to match exactly the vector field limits (whereas they would sometimes be off by a small floating point error previously).
- if the streamplot does not reach the edges of the vector field (e.g., due to the use of `start_points` and `maxlength`), then margins expansion will not cross the vector field limits anymore.

This change is also used internally to ensure that polar plots don't display negative *r* values unless the user really passes in a negative value.

`gid` in svg output

Previously, if a figure, axis, legend or some other artists had a custom `gid` set (e.g. via `.set_gid()`), this would not be reflected in the svg output. Instead a default gid, like `figure_1` would be shown. This is now fixed, such that e.g. `fig.set_gid("myfigure")` correctly shows up as `<g id="myfigure">` in the svg file. If you relied on the gid having the default format, you now need to make sure not to set the `gid` parameter of the artists.

Fonts

Font weight guessing now first checks for the presence of the FT_STYLE_BOLD_FLAG before trying to match substrings in the font name. In particular, this means that Times New Roman Bold is now correctly detected as bold, not normal weight.

Color-like checking

matplotlib.colors.is_color_like used to return True for all string representations of floats. However, only those with values in 0-1 are valid colors (representing grayscale values). *is_color_like* now returns False for string representations of floats outside 0-1.

Default image interpolation

Images displayed in Matplotlib previously used nearest-neighbor interpolation, leading to aliasing effects for downscaling and non-integer upscaling.

New default for `rcParams["image.interpolation"]` (default: `'antialiased'`) is the new option "antialiased". `imshow(A, interpolation='antialiased')` will apply a Hanning filter when resampling the data in A for display (or saving to file) *if* the upsample rate is less than a factor of three, and not an integer; downsampled data is always smoothed at resampling.

To get the old behavior, set `rcParams["image.interpolation"]` (default: `'antialiased'`) to the old default "nearest" (or specify the `interpolation` kwarg of *Axes.imshow*)

To always get the anti-aliasing behavior, no matter what the up/down sample rate, set `rcParams["image.interpolation"]` (default: `'antialiased'`) to "hanning" (or one of the other filters available).

Note that the "hanning" filter was chosen because it has only a modest performance penalty. Anti-aliasing can be improved with other filters.

rcParams

When using *RendererSVG* with `rcParams["svg.image_inline"] == True`, externally written images now use a single counter even if the `renderer.basename` attribute is overwritten, rather than a counter per basename.

This change will only affect you if you used `rcParams["svg.image_inline"] = True` (the default is False) *and* manually modified `renderer.basename`.

Changed the default value of `rcParams["axes.formatter.limits"]` (default: `[-5, 6]`) from -7, 7 to -5, 6 for better readability.

add_subplot()

Figure.add_subplot() and *pyplot.subplot()* do not accept a *figure* keyword argument anymore. It only used to work anyway if the passed figure was `self` or the current figure, respectively.

indicate_inset()

In <= 3.1.0, *indicate_inset* and *indicate_inset_zoom* were documented as returning a 4-tuple of *ConnectionPatch*, where in fact they returned a 4-length list.

They now correctly return a 4-tuple. *indicate_inset* would previously raise an error if the optional *inset_ax* was not supplied; it now completes successfully, and returns *None* instead of the tuple of `ConnectionPatch`.

PGF backend

The pgf backend's get_canvas_width_height now returns the canvas size in display units rather than in inches, which it previously did. The new behavior is the correct one given the uses of `get_canvas_width_height` in the rest of the codebase.

The pgf backend now includes images using `\includegraphics` instead of `\pgfimage` if the version of `graphicx` is recent enough to support the `interpolate` option (this is detected automatically).

cbook

The default value of the "obj_type" parameter to `cbook.warn_deprecated` has been changed from "attribute" (a default that was never used internally) to the empty string.

Testing

The test suite no longer turns on the Python fault handler by default. Set the standard `PYTHONFAULTHANDLER` environment variable to do so.

Backend `supports_blit`

Backends do not need to explicitly define the flag `supports_blit` anymore. This is only relevant for backend developers. Backends had to define the flag `supports_blit`. This is not needed anymore because the blitting capability is now automatically detected.

Exception changes

Various APIs that raised a `ValueError` for incorrectly typed inputs now raise `TypeError` instead: *backend_bases.GraphicsContextBase.set_clip_path*, `blocking_input.BlockingInput.__call__`, *cm.register_cmap*, *dviread.DviFont*, *rcsetup.validate_hatch*, *rcsetup.validate_animation_writer_path*, *spines.Spine*, many classes in the *matplotlib.transforms* module and *matplotlib.tri* package, and Axes methods that take a `norm` parameter.

If extra kwargs are passed to *LogScale*, `TypeError` will now be raised instead of `ValueError`.

mplot3d auto-registration

mpl_toolkits.mplot3d is always registered by default now. It is no longer necessary to import mplot3d to create 3d axes with

```
ax = fig.add_subplot(111, projection="3d")
```

`SymLogNorm` now has a *base* parameter

Previously, *SymLogNorm* had no *base* keyword argument and the base was hard-coded to `base=np.e`. This was inconsistent with the default behavior of *SymmetricalLogScale* (which defaults to `base=10`) and the use of the word "decade" in the documentation.

In preparation for changing the default base to 10, calling *SymLogNorm* without the new *base* keyword argument emits a deprecation warning.

Deprecations

`matplotlib.use`

The `warn` parameter to *matplotlib.use()* is deprecated (catch the `ImportError` emitted on backend switch failure and reemit a warning yourself if so desired).

plotfile

`.pyplot.plotfile` is deprecated in favor of separately loading and plotting the data. Use pandas or NumPy to load data, and pandas or matplotlib to plot the resulting data.

axes and axis

Setting `Axis.major.locator`, `Axis.minor.locator`, `Axis.major.formatter` or `Axis.minor.formatter` to an object that is not a subclass of *Locator* or *Formatter* (respectively) is deprecated. Note that these attributes should usually be set using *Axis.set_major_locator*, *Axis.set_minor_locator*, etc. which already raise an exception when an object of the wrong class is passed.

Passing more than one positional argument or unsupported keyword arguments to *axis()* is deprecated (such arguments used to be silently ignored).

`minor` argument will become keyword-only

Using the parameter `minor` to `get_*ticks()` / `set_*ticks()` as a positional parameter is deprecated. It will become keyword-only in future versions.

axes_grid1

The `mpl_toolkits.axes_grid1.colorbar` module and its colorbar implementation are deprecated in favor of *matplotlib.colorbar*, as the former is essentially abandoned and the latter is a more featureful replacement with a nearly compatible API (for example, the following additional keywords are supported: `panchor`, `extendfrac`, `extendrect`).

The main differences are:

- Setting the ticks on the colorbar is done by calling `colorbar.set_ticks` rather than `colorbar.cbar_axis.set_xticks` or `colorbar.cbar_axis.set_yticks`; the `locator` parameter to `colorbar()` is deprecated in favor of its synonym `ticks` (which already existed previously, and is consistent with *matplotlib.colorbar*).
- The colorbar's long axis is accessed with `colorbar.xaxis` or `colorbar.yaxis` depending on the orientation, rather than `colorbar.cbar_axis`.
- The default ticker is no longer `MaxNLocator(5)`, but a `_ColorbarAutoLocator`.
- Overdrawing multiple colorbars on top of one another in a single Axes (e.g. when using the `cax` attribute of *ImageGrid* elements) is not supported; if you previously relied on the second colorbar being drawn over the first, you can call `cax.cla()` to clear the axes before drawing the second colorbar.

During the deprecation period, the `mpl_toolkits.legacy_colorbar` rcParam can be set to True to use `mpl_toolkits.axes_grid1.colorbar` in *mpl_toolkits.axes_grid1* code with a deprecation warning (the default), or to False to use `matplotlib.colorbar`.

Passing a `pad` size of `None` (the default) as a synonym for zero to the `append_axes`, `new_horizontal` and `new_vertical` methods of *axes_grid1.axes_divider.AxesDivider* is deprecated. In a future release, the default value of `None` will mean "use `rcParams["figure.subplot.wspace"]` (default: `0.2`) or `rcParams["figure.subplot.hspace"]` (default: `0.2`)" (depending on the orientation). Explicitly pass `pad=0` to keep the old behavior.

Axes3D

`mplot3d.axis3d.get_flip_min_max` is deprecated.

`axes3d.unit_bbox` is deprecated (use `Bbox.unit` instead).

`axes3d.Axes3D.w_xaxis`, `.w_yaxis`, and `.w_zaxis` are deprecated (use `.xaxis`, `.yaxis`, and `.zaxis` instead).

matplotlib.cm

`cm.revcmap` is deprecated. Use *Colormap.reversed* to reverse a colormap.

`cm.datad` no longer contains entries for reversed colormaps in their "unconverted" form.

axisartist

`mpl_toolkits.axisartist.grid_finder.GridFinderBase` is deprecated (its only use is to be inherited by the *GridFinder* class which just provides more defaults in the constructor and directly sets the transforms, so `GridFinderBase`'s methods were just moved to *GridFinder*).

`axisartist.axis_artist.BezierPath` is deprecated (use *patches.PathPatch* to draw arbitrary Paths).

`AxisArtist.line` is now a *patches.PathPatch* instance instead of a `BezierPath` instance.

Returning a factor equal to None from axisartist Locators (which are **not** the same as "standard" tick Locators), or passing a factor equal to None to axisartist Formatters (which are **not** the same as "standard" tick Formatters) is deprecated. Pass a factor equal to 1 instead.

For the *mpl_toolkits.axisartist.axis_artist.AttributeCopier* class, the constructor and the `set_ref_artist` method, and the *default_value* parameter of `get_attribute_from_ref_artist`, are deprecated.

Deprecation of the constructor means that classes inheriting from *AttributeCopier* should no longer call its constructor.

Locators

The unused *Locator.autoscale* method is deprecated (pass the axis limits to *Locator.view_limits* instead).

Animation

The following methods and attributes of the `MovieWriterRegistry` class are deprecated: `set_dirty`, `ensure_not_dirty`, `reset_available_writers`, `avail`.

`smart_bounds()`

The "smart_bounds" functionality is deprecated. This includes `Axis.set_smart_bounds()`, `Axis.get_smart_bounds()`, `Spine.set_smart_bounds()`, and `Spine.get_smart_bounds()`.

`boxplot()`

Setting the `whis` parameter of *Axes.boxplot* and *cbook.boxplot_stats* to "range" to mean "the whole data range" is deprecated; set it to (0, 100) (which gets interpreted as percentiles) to achieve the same effect.

`fill_between()`

Passing scalars to parameter *where* in `fill_between()` and `fill_betweenx()` is deprecated. While the documentation already states that *where* must be of the same size as *x* (or *y*), scalars were accepted and broadcasted to the size of *x*. Non-matching sizes will raise a `ValueError` in the future.

`tight_layout()`

The `renderer` parameter to *Figure.tight_layout* is deprecated; this method now always uses the renderer instance cached on the *Figure*.

rcParams

The `rcsetup.validate_animation_writer_path` function is deprecated.

Setting `rcParams["savefig.format"]` (default: `'png'`) to "auto" is deprecated; use its synonym "png" instead.

Setting `rcParams["text.hinting"]` (default: `'force_autohint'`) to True or False is deprecated; use their synonyms "auto" or "none" instead.

`rcsetup.update_savefig_format` is deprecated.

`rcsetup.validate_path_exists` is deprecated (use `os.path.exists` to check whether a path exists).

`rcsetup.ValidateInterval` is deprecated.

Dates

`dates.mx2num` is deprecated.

TK

`NavigationToolbar2Tk.set_active` is deprecated, as it has no (observable) effect.

WX

`FigureFrameWx.statusbar` and `NavigationToolbar2Wx.statbar` are deprecated. The status bar can be retrieved by calling standard wx methods (`frame.GetStatusBar()` and `toolbar.GetTopLevelParent().GetStatusBar()`).

`backend_wx.ConfigureSubplotsWx.configure_subplots` and `backend_wx.ConfigureSubplotsWx.get_canvas` are deprecated.

PGF

`backend_pgf.repl_escapetext` and `backend_pgf.repl_mathdefault` are deprecated.

`RendererPgf.latexManager` is deprecated.

FigureCanvas

`FigureCanvasBase.draw_cursor` (which has never done anything and has never been overridden in any backend) is deprecated.

`FigureCanvasMac.invalidate` is deprecated in favor of its synonym, `FigureCanvasMac.draw_idle`.

The `dryrun` parameter to the various `FigureCanvasFoo.print_foo` methods is deprecated.

QuiverKey doc

`quiver.QuiverKey.quiverkey_doc` is deprecated; use `quiver.QuiverKey.__init__.__doc__` instead.

matplotlib.mlab

`mlab.apply_window` and `mlab.stride_repeat` are deprecated.

Fonts

`font_manager.JSONEncoder` is deprecated. Use *font_manager.json_dump* to dump a *FontManager* instance.

`font_manager.createFontList` is deprecated. *font_manager.FontManager.addfont* is now available to register a font at a given path.

The `as_str`, `as_rgba_str`, `as_array`, `get_width` and `get_height` methods of `matplotlib.ft2font.FT2Image` are deprecated. Convert the `FT2Image` to a NumPy array with `np.asarray` before processing it.

Colors

The function *matplotlib.colors.makeMappingArray* is not considered part of the public API any longer. Thus, it's deprecated.

Using a string of single-character colors as a color sequence (e.g. "rgb") is deprecated. Use an explicit list instead.

Scales

Passing unsupported keyword arguments to *ScaleBase*, and its subclasses *LinearScale* and *SymmetricalLogScale*, is deprecated and will raise a TypeError in 3.3.

If extra keyword arguments are passed to *LogScale*, TypeError will now be raised instead of ValueError.

Testing

The matplotlib.testing.disable_internet module is deprecated. Use (for example) pytest-remotedata instead.

Support in *matplotlib.testing* for nose-based tests is deprecated (a deprecation is emitted if using e.g. the decorators from that module while both 1) matplotlib's conftests have not been called and 2) nose is in sys.modules).

testing.is_called_from_pytest is deprecated.

During the deprecation period, to force the generation of nose base tests, import nose first.

The switch_backend_warn parameter to matplotlib.test has no effect and is deprecated.

testing.jpl_units.UnitDbl.UnitDbl.checkUnits is deprecated.

DivergingNorm renamed to TwoSlopeNorm

DivergingNorm was a misleading name; although the norm was developed with the idea that it would likely be used with diverging colormaps, the word 'diverging' does not describe or evoke the norm's mapping function. Since that function is monotonic, continuous, and piece-wise linear with two segments, the norm has been renamed to *TwoSlopeNorm*

Misc

matplotlib.get_home is deprecated (use e.g. os.path.expanduser("~")) instead.

matplotlib.compare_versions is deprecated (use comparison of distutils.version.LooseVersions instead).

matplotlib.checkdep_ps_distiller is deprecated.

matplotlib.figure.AxesStack is considered private API and will be removed from the public API in future versions.

BboxBase.is_unit is deprecated (check the Bbox extents if needed).

`Affine2DBase.matrix_from_values(...)` is deprecated. Use (for example) `Affine2D.from_values(...).get_matrix()` instead.

`style.core.is_style_file` and `style.core.iter_style_files` are deprecated.

The `datapath` rcParam

Use *`get_data_path`* instead. (The rcParam is deprecated because it cannot be meaningfully set by an end user.) The rcParam had no effect from 3.2.0, but was deprecated only in 3.2.1. In 3.2.1+ if `'datapath'` is set in a `matplotlibrc` file it will be respected, but this behavior will be removed in 3.3.

Removals

The `matplotlib.testing.determinism` module, which exposes no public API, has been deleted.

The following API elements have been removed:

- `backend_gtk3.PIXELS_PER_INCH`
- `backend_pgf.re_escapetext`, `backend_pgf.re_mathdefault`.
- the `matplotlib.backends.tkagg`, `matplotlib.backends.windowing`, `matplotlib.backends.wx_compat`, and `matplotlib.compat.subprocess` modules
- `RcParams.msg_depr`, `RcParams.msg_depr_ignore`, `RcParams.msg_depr_set`, `RcParams.msg_obsolete`, `RcParams.msg_backend_obsolete`
- `afm.parse_afm` (use `afm.AFM` instead)
- `axes.Axes.mouseover_set`
- `backend_cairo.ArrayWrapper`, `backend_cairo.RendererCairo.convert_path`
- `backend_gtk3.FileChooserDialog.sorted_filetypes` (use `sorted(self.filetypes.items())` instead)
- `backend_pgf.get_texcommand`
- `backend_pdf.PdfFile.texFontMap`
- `backend_ps.get_bbox`
- `backend_qt.FigureCanvasQt.keyAutoRepeat` (use `event.guiEvent.isAutoRepeat` instead), `backend_qt.error_msg_qt`, `backend_qt.exception_handler`
- `backend_wx.FigureCanvasWx.macros`
- `backends.pylab_setup`

- cbook.Bunch (use types.SimpleNamespace instead), cbook.Locked, cbook.unicode_safe, cbook.is_numlike (use isinstance(..., numbers.Number) instead), cbook.mkdirs (use os.makedirs(..., exist_ok=True) instead), cbook.GetRealpathAndStat (use cbook.get_realpath_and_stat instead), cbook.listFiles
- container.Container.set_remove_method
- contour.ContourLabeler.cl, contour.ContourLabeler.cl_xy, contour.ContourLabeler.cl_cvalues (use labelTexts, labelXYs, labelCValues instead)
- dates.DateFormatter.strftime, dates.DateFormatter.strftime_pre_1900
- font_manager.TempCache, font_manager.FontManager.ttffiles, font_manager.FontManager.afmfiles
- mathtext.unichr_safe (use chr instead)
- patches.YAArrow (use patches.FancyArrowPatch instead)
- sphinxext.plot_directive.remove_coding
- table.Table.get_child_artists
- testing.compare.compare_float, testing.decorators.CleanupTest, testing.decorators.ImageComparisonTest, testing.decorators.skip_if_command_unavailable, support for nose-based tests
- text.Annotation.arrow (use text.Annotation.arrow_patch instead)
- textpath.TextToPath.tex_font_map
- ticker.Base, ticker.closeto, ticker.nearest_long
- axes_grid1.axes_divider.LocatableAxesBase, axes_grid1.axes_divider.locatable_axes_factory, axes_grid1.axes_divider.Axes (use axes_grid1.mpl_axes.Axes instead), axes_grid1.axes_divider.LocatableAxes (use axes_grid1.mpl_axes.Axes instead)
- axisartist.axes_divider.Axes, axisartist.axes_divider.LocatableAxes (use axisartist.axislines.Axes instead)
- the normed kwarg to hist (use density instead)
- the verts parameter to scatter (use marker instead)
- passing (verts, 0) or (..., 3) when specifying a marker to specify a path or a circle, respectively (instead, use verts or "o", respectively)
- rcParams["examples.directory"]

The following members of matplotlib.backends.backend_pdf.PdfFile were removed:

- nextObject
- nextFont
- nextAlphaState
- nextHatch

- nextImage
- alphaStateObject

The required_interactive_framework attribute of backend modules introduced in Matplotlib 3.0 has been moved to the FigureCanvas class, in order to let it be inherited by third-party canvas subclasses and to make it easier to know what interactive framework is required by a canvas class.

backend_qt4.FigureCanvasQT5, which is an alias for backend_qt5.FigureCanvasQT (but only exists under that name in backend_qt4), has been removed.

Development changes

Windows build

Previously, when building the matplotlib._png extension, the build script would add "png" and "z" to the extensions .libraries attribute (if pkg-config information is not available, which is in particular the case on Windows).

In particular, this implies that the Windows build would look up files named png.lib and z.lib; but neither libpng upstream nor zlib upstream provides these files by default. (On Linux, this would look up libpng.so and libz.so, which are indeed standard names.)

Instead, on Windows, we now look up libpng16.lib and zlib.lib, which *are* the upstream names for the shared libraries (as of libpng 1.6.x).

For a statically-linked build, the upstream names are libpng16_static.lib and zlibstatic.lib; one still needs to manually rename them if such a build is desired.

Packaging DLLs

Previously, it was possible to package Windows DLLs into the Maptlotlib wheel (or sdist) by copying them into the source tree and setting the package_data.dlls entry in setup.cfg.

DLLs copied in the source tree are now always packaged; the package_data.dlls entry has no effect anymore. If you do not want to include the DLLs, don't copy them into the source tree.

16.1.4 API Changes for 3.1.1

Behavior changes

Locator.nonsingular return order

Locator.nonsingular (introduced in mpl 3.1) now returns a range `v0, v1` with `v0 <= v1`. This behavior is consistent with the implementation of `nonsingular` by the *LogLocator* and *LogitLocator* subclasses.

16.1.5 API Changes for 3.1.0

Behavior changes

Matplotlib.use

Switching backends via *matplotlib.use* is now allowed by default, regardless of whether *matplotlib.pyplot* has been imported. If the user tries to switch from an already-started interactive backend to a different interactive backend, an ImportError will be raised.

Invalid points in PathCollections

PathCollections created with *scatter* now keep track of invalid points. Previously, points with nonfinite (infinite or nan) coordinates would not be included in the offsets (as returned by *PathCollection.get_offsets*) of a *PathCollection* created by *scatter*, and points with nonfinite values (as specified by the *c* kwarg) would not be included in the array (as returned by *PathCollection.get_array*)

Such points are now included, but masked out by returning a masked array.

If the *plotnonfinite* kwarg to *scatter* is set, then points with nonfinite values are plotted using the bad color of the *collections.PathCollection*'s colormap (as set by *colors.Colormap.set_bad()*).

Alpha blending in imshow of RBGA input

The alpha-channel of RBGA images is now re-sampled independently of RGB channels. While this is a bug fix, it does change the output and may result in some down-stream image comparison tests to fail.

Autoscaling

On log-axes where a single value is plotted at a "full" decade (1, 10, 100, etc.), the autoscaling now expands the axis symmetrically around that point, instead of adding a decade only to the right.

Log-scaled axes

When the default *LogLocator* would generate no ticks for an axis (e.g., an axis with limits from 0.31 to 0.39) or only a single tick, it now instead falls back on the linear *AutoLocator* to pick reasonable tick positions.

`Figure.add_subplot` with no arguments

Calling *Figure.add_subplot()* with no positional arguments used to do nothing; this now is equivalent to calling `add_subplot(111)` instead.

`bxp` and rcparams

bxp now respects `rcParams["boxplot.boxprops.linewidth"]` (default: `1.0`) even when *patch_artist* is set. Previously, when the *patch_artist* parameter was set, *bxp* would ignore `rcParams["boxplot.boxprops.linewidth"]` (default: `1.0`). This was an oversight -- in particular, *boxplot* did not ignore it.

Major/minor tick collisions

Minor ticks that collide with major ticks are now hidden by default. Previously, certain locator classes (*LogLocator*, *AutoMinorLocator*) contained custom logic to avoid emitting tick locations that collided with major ticks when they were used as minor locators. This logic has now moved to the *Axis* class, and is used regardless of the locator class. You can control this behavior via the `remove_overlaping_locs` attribute on *Axis*.

If you were relying on both the major and minor tick labels to appear on the same tick, you may need to update your code. For example, the following snippet

```
import numpy as np
import matplotlib.dates as mdates
import matplotlib.pyplot as plt

t = np.arange("2018-11-03", "2018-11-06", dtype="datetime64")
x = np.random.rand(len(t))

fig, ax = plt.subplots()
ax.plot(t, x)
ax.xaxis.set(
    major_locator=mdates.DayLocator(),
    major_formatter=mdates.DateFormatter("\n%a"),
    minor_locator=mdates.HourLocator((0, 6, 12, 18)),
    minor_formatter=mdates.DateFormatter("%H:%M"),
)
# disable removing overlapping locations
ax.xaxis.remove_overlapping_locs = False
plt.show()
```

labeled days using major ticks, and hours and minutes using minor ticks and added a newline to the major ticks labels to avoid them crashing into the minor tick labels. Setting the *remove_overlapping_locs* property (also accessible via *set_remove_overlapping_locs* / *get_remove_overlapping_locs* and *setp*) disables removing overlapping tick locations.

The major tick labels could also be adjusted include hours and minutes, as the minor ticks are gone, so the `major_formatter` would be:

```
mdates.DateFormatter("%H:%M\n%a")
```

usetex support

Previously, if `rcParams["text.usetex"]` (default: `False`) was True, then constructing a *TextPath* on a non-mathtext string with `usetex=False` would rely on the mathtext parser (but not on usetex support!) to parse the string. The mathtext parser is not invoked anymore, which may cause slight changes in glyph positioning.

get_window_extents

matplotlib.axes.Axes.get_window_extent used to return a bounding box that was slightly larger than the axes, presumably to take into account the ticks that may be on a spine. However, it was not scaling the tick sizes according to the dpi of the canvas, and it did not check if the ticks were visible, or on the spine.

Now *matplotlib.axes.Axes.get_window_extent* just returns the axes extent with no padding for ticks.

This affects *matplotlib.axes.Axes.get_tightbbox* in cases where there are outward ticks with no tick labels, and it also removes the (small) pad around axes in that case.

spines.Spine.get_window_extent now takes into account ticks that are on the spine.

Sankey

Previously, *Sankey.add* would only accept a single string as the *labels* argument if its length is equal to the number of flows, in which case it would use one character of the string for each flow.

The behavior has been changed to match the documented one: when a single string is passed, it is used to label all the flows.

`FontManager` scores

font_manager.FontManager.score_weight is now more strict with its inputs. Previously, when a weight string was passed to *font_manager.FontManager.score_weight*,

- if the weight was the string representation of an integer, it would be converted to that integer,
- otherwise, if the weight was not a standard weight name, it would be silently replaced by a value of 500 ("normal" weight).

font_manager.FontManager.score_weight now raises an exception on such inputs.

Text alignment

Text alignment was previously incorrect, in particular for multiline text objects with large descenders (i.e. subscripts) and rotated text. These have been fixed and made more consistent, but could make old code that has compensated for this no longer have the correct alignment.

Upper case color strings

Support for passing single-letter colors (one of "rgbcmykw") as UPPERCASE characters is deprecated; these colors will become case-sensitive (lowercase) after the deprecation period has passed.

The goal is to decrease the number of ambiguous cases when using the `data` keyword to plotting methods; e.g. `plot("X", "Y", data={"X": ..., "Y": ...})` will not warn about "Y" possibly being a color anymore after the deprecation period has passed.

Degenerate limits

When bounds passed to *set_xlim* are degenerate (i.e. the lower and upper value are equal), the method used to "expand" the bounds now matches the expansion behavior of autoscaling when the plot contains a single x-value, and should in particular produce nicer limits for non-linear scales.

`plot` format string parsing

In certain cases, *`plot`* would previously accept format strings specifying more than one linestyle (e.g. `"---."` which specifies both `"--"` and `"-."`); only use one of them would be used. This now raises a `ValueError` instead.

HTMLWriter

The HTMLWriter constructor is more strict: it no longer normalizes unknown values of *default_mode* to 'loop', but errors out instead.

AFM parsing

In accordance with the AFM spec, the AFM parser no longer truncates the `UnderlinePosition` and `UnderlineThickness` fields to integers.

The `Notice` field (which can only be publicly accessed by the deprecated `afm.parse_afm` API) is no longer decoded to a `str`, but instead kept as `bytes`, to support non-conformant AFM files that use non-ASCII characters in that field.

`Artist.set` keyword normalisation

`Artist.set` now normalizes keywords before sorting them. Previously it sorted its keyword arguments in reverse alphabetical order (with a special-case to put `color` at the end) before applying them.

It now normalizes aliases (and, as above, emits a warning on duplicate properties) before doing the sorting (so `c` goes to the end too).

`Axes.tick_params` argument checking

Previously *`Axes.tick_params`* silently did nothing when an invalid *axis* parameter was supplied. This behavior has been changed to raise a `ValueError` instead.

`Axes.hist` output

Input that consists of multiple empty lists will now return a list of histogram values for each one of the lists. For example, an input of `[[],[]]` will return 2 lists of histogram values. Previously, a single list was returned.

backend_bases.Timer.remove_callback future signature change

Currently, `backend_bases.Timer.remove_callback(func, *args, **kwargs)` removes a callback previously added by `backend_bases.Timer.add_callback(func, *args, **kwargs)`, but if `*args, **kwargs` is not passed in (ex, `Timer.remove_callback(func)`), then the first callback with a matching `func` is removed, regardless of whether it was added with or without `*args, **kwargs`.

In a future version, `Timer.remove_callback` will always use the latter behavior (not consider `*args, **kwargs`); to specifically consider them, add the callback as a `functools.partial` object

```
cb = timer.add_callback(functools.partial(func, *args, **kwargs))
# ...
# later
timer.remove_callback(cb)
```

`backend_bases.Timer.add_callback` was modified to return *func* to simplify the above usage (previously it returned None); this also allows using it as a decorator.

The new API is modelled after `atexit.register` / `atexit.unregister`.

StemContainer performance increase

StemContainer objects can now store a *LineCollection* object instead of a list of *Line2D* objects for stem lines plotted using *stem*. This gives a very large performance boost to displaying and moving *stem* plots.

This will become the default behaviour in Matplotlib 3.3. To use it now, the *use_line_collection* keyword argument to *stem* can be set to `True`

```
ax.stem(..., use_line_collection=True)
```

Individual line segments can be extracted from the *LineCollection* using `get_segements()`. See the *LineCollection* documentation for other methods to retrieve the collection properties.

ColorbarBase inheritance

matplotlib.colorbar.ColorbarBase is no longer a subclass of *cm.ScalarMappable*. This inheritance lead to a confusing situation where the *cm.ScalarMappable* passed to *matplotlib.colorbar.Colorbar* (*colorbar*) had a `set_norm` method, as did the colorbar. The colorbar is now purely a follower to the *ScalarMappable* norm and colormap, and the old inherited methods `set_norm`, `set_cmap`, `set_clim` are deprecated, as are the getter versions of those calls. To set the norm associated with a colorbar do `colorbar.mappable.set_norm()` etc.

FreeType and libpng search paths

The `MPLBASEDIRLIST` environment variables and `basedirlist` entry in `setup.cfg` have no effect anymore. Instead, if building in situations where FreeType or libpng are not in the compiler or linker's default path, set the standard environment variables `CFLAGS`/`LDFLAGS` on Linux or OSX, or `CL`/`LINK` on Windows, to indicate the relevant paths.

See details in *Installation Guide*.

Setting artist properties twice or more in the same call

Setting the same artist property multiple time via aliases is deprecated. Previously, code such as

```
plt.plot([0, 1], c="red", color="blue")
```

would emit a warning indicating that `c` and `color` are aliases of one another, and only keep the `color` kwarg. This behavior has been deprecated; in a future version, this will raise a TypeError, similar to Python's behavior when a keyword argument is passed twice

```
plt.plot([0, 1], c="red", c="blue")
```

This warning is raised by *`normalize_kwargs`*.

Path code types

Path code types like `Path.MOVETO` are now `np.uint8` instead of `int` `Path.STOP`, `Path.MOVETO`, `Path.LINETO`, `Path.CURVE3`, `Path.CURVE4` and `Path.CLOSEPOLY` are now of the type `Path.code_type` (`np.uint8` by default) instead of plain `int`. This makes their type match the array value type of the `Path.codes` array.

LaTeX code in matplotlibrc file

Previously, the rc file keys `pgf.preamble` and `text.latex.preamble` were parsed using commmas as separators. This would break valid LaTeX code, such as:

```
\usepackage[protrusion=true, expansion=false]{microtype}
```

The parsing has been modified to pass the complete line to the LaTeX system, keeping all commas. Passing a list of strings from within a Python script still works as it used to. Passing a list containing non-strings now fails, instead of coercing the results to strings.

Axes.spy

The method *Axes.spy* now raises a TypeError for the keyword arguments *interpolation* and *linestyle* instead of silently ignoring them.

Furthermore, *Axes.spy* spy does now allow for an *extent* argument (was silently ignored so far).

A bug with `Axes.spy(..., origin='lower')` is fixed. Previously this flipped the data but not the y-axis resulting in a mismatch between axes labels and actual data indices. Now, *origin*='*lower*' flips both the data and the y-axis labels.

Boxplot tick methods

The *manage_xticks* parameter of *boxplot* and *bxp* has been renamed (with a deprecation period) to *manage_ticks*, to take into account the fact that it manages either x or y ticks depending on the *vert* parameter.

When `manage_ticks=True` (the default), these methods now attempt to take previously drawn boxplots into account when setting the axis limits, ticks, and tick labels.

MouseEvents

MouseEvents now include the event name in their `str()`. Previously they contained the prefix "MPL MouseEvent".

RGBA buffer return type

FigureCanvasAgg.buffer_rgba and *RendererAgg.buffer_rgba* now return a memoryview The `buffer_rgba` method now allows direct access to the renderer's underlying buffer (as a `(m, n, 4)`-shape memoryview) rather than copying the data to a new bytestring. This is consistent with the behavior on Py2, where a buffer object was returned.

matplotlib.font_manager.win32InstalledFonts return type

matplotlib.font_manager.win32InstalledFonts returns an empty list instead of None if no fonts are found.

`Axes.fmt_xdata` and `Axes.fmt_ydata` error handling

Previously, if the user provided a `Axes.fmt_xdata` or `Axes.fmt_ydata` function that raised a `TypeError` (or set them to a non-callable), the exception would be silently ignored and the default formatter be used instead. This is no longer the case; the exception is now propagated out.

Deprecation of redundant `Tick` attributes

The `gridOn`, `tick1On`, `tick2On`, `label1On`, and `label2On` *Tick* attributes have been deprecated. Directly get and set the visibility on the underlying artists, available as the `gridline`, `tick1line`, `tick2line`, `label1`, and `label2` attributes.

The `label` attribute, which was an alias for `label1`, has been deprecated.

Subclasses that relied on setting the above visibility attributes needs to be updated; see e.g. `examples/api/skewt.py`.

Passing a Line2D's drawstyle together with the linestyle is deprecated

Instead of `plt.plot(..., linestyle="steps--")`, use `plt.plot(..., linestyle="--", drawstyle="steps")`. `ds` is now an alias for `drawstyle`.

`pgi` support dropped

Support for `pgi` in the GTK3 backends has been dropped. `pgi` is an alternative implementation to `PyGObject`. `PyGObject` should be used instead.

rcParam changes

Removed

The following deprecated rcParams have been removed:

- `text.dvipnghack`
- `nbagg.transparent` (use `rcParams["figure.facecolor"]` (default: `'white'`) instead)
- `plugins.directory`
- `axes.hold`
- `backend.qt4` and `backend.qt5` (set the *QT_API* environment variable instead)

Deprecated

The associated validator functions `rcsetup.validate_qt4` and `validate_qt5` are deprecated.

The `verbose.fileo` and `verbose.level` rcParams have been deprecated. These have had no effect since the switch from Matplotlib's old custom Verbose logging to the stdlib's `logging` module. In addition the `rcsetup.validate_verbose` function is deprecated.

The `text.latex.unicode` rcParam now defaults to `True` and is deprecated (i.e., in future versions of Matplotlib, unicode input will always be supported). Moreover, the underlying implementation now uses `\usepackage[utf8]{inputenc}` instead of `\usepackage{ucs}\usepackage[utf8x]{inputenc}`.

Exception changes

- *`mpl_toolkits.axes_grid1.axes_size.GetExtentHelper`* now raises `ValueError` for invalid directions instead of `KeyError`.
- Previously, subprocess failures in the animation framework would raise either in a `RuntimeError` or a `ValueError` depending on when the error occurred. They now raise a `subprocess.CalledProcessError` with attributes set as documented by the exception class.
- In certain cases, Axes methods (and pyplot functions) used to raise a `RuntimeError` if they were called with a `data` kwarg and otherwise mismatched arguments. They now raise a `TypeError` instead.
- *`Axes.streamplot`* does not support irregularly gridded `x` and `y` values. So far, it used to silently plot an incorrect result. This has been changed to raise a `ValueError` instead.
- The `streamplot.Grid` class, which is internally used by streamplot code, also throws a `ValueError` when irregularly gridded values are passed in.

Removals

The following deprecated APIs have been removed:

Classes and methods

- `Verbose` (replaced by python logging library)
- `artist.Artist.hitlist` (no replacement)
- `artist.Artist.is_figure_set` (use `artist.figure is not None` instead)
- `axis.Axis.unit_data` (use `axis.Axis.units` instead)
- `backend_bases.FigureCanvasBase.onRemove` (no replacement) `backend_bases.FigureManagerBase.show_popup` (this never did anything)
- `backend_wx.SubplotToolWx` (no replacement)
- `backend_wx.Toolbar` (use `backend_wx.NavigationToolbar2Wx` instead)
- `cbook.align_iterators` (no replacement)
- `contour.ContourLabeler.get_real_label_width` (no replacement)
- `legend.Legend.draggable` (use `legend.Legend.set_draggable()` instead)
- `texmanager.TexManager.postscriptd`, `texmanager.TexManager.pscnt`, `texmanager.TexManager.make_ps`, `texmanager.TexManager.get_ps_bbox` (no replacements)

Arguments

- The *fig* kwarg to *GridSpec.get_subplot_params* and *GridSpecFromSubplotSpec.get_subplot_params* (use the argument *figure* instead)
- Passing 'box-forced' to *Axes.set_adjustable* (use 'box' instead)
- Support for the strings 'on'/'true'/'off'/'false' to mean `True` / `False` (directly use `True` / `False` instead). The following functions are affected:
  - *axes.Axes.grid*
  - *Axes3D.grid*
  - *Axis.set_tick_params*
  - *pyplot.box*
- Using *pyplot.axes* with an *axes.Axes* type argument (use *pyplot.sca* instead)

Other

The following miscellaneous API elements have been removed

- svgfont support (in `rcParams["svg.fonttype"]` (default: `'path'`))
- Logging is now done with the standard python `logging` library. `matplotlib.verbose` and the command line switches `--verbose-LEVEL` have been removed.

  To control the logging output use:

```
import logging
logger = logging.getLogger('matplotlib')
logger.setLevel(logging.INFO)
# configure log handling: Either include it into your ``logging`` hierarchy,
# e.g. by configuring a root looger using ``logging.basicConfig()``,
# or add a standalone handler to the matplotlib logger:
logger.addHandler(logging.StreamHandler())
```

- `__version__numpy__`
- `collections.CIRCLE_AREA_FACTOR`
- `font_manager.USE_FONTCONFIG`
- `font_manager.cachedir`

`matplotlib.mlab` removals

Lots of code inside the *`matplotlib.mlab`* module which was deprecated in Matplotlib 2.2 has been removed. See below for a list:

- `mlab.exp_safe` (use `numpy.exp` instead)
- `mlab.amap`
- `mlab.logspace` (use `numpy.logspace` instead)
- `mlab.rms_flat`
- `mlab.l1norm` (use `numpy.linalg.norm(a, ord=1)` instead)
- `mlab.l2norm` (use `numpy.linalg.norm(a, ord=2)` instead)
- `mlab.norm_flat` (use `numpy.linalg.norm(a.flat, ord=2)` instead)
- `mlab.frange` (use `numpy.arange` instead)
- `mlab.identity` (use `numpy.identity` instead)
- `mlab.base_repr`
- `mlab.binary_repr`
- `mlab.ispower2`
- `mlab.log2` (use `numpy.log2` instead)

- mlab.isvector
- mlab.movavg
- mlab.safe_isinf (use numpy.isinf instead)
- mlab.safe_isnan (use numpy.isnan instead)
- mlab.cohere_pairs (use scipy.signal.coherence instead)
- mlab.entropy (use scipy.stats.entropy instead)
- mlab.normpdf (use scipy.stats.norm.pdf instead)
- mlab.find (use np.nonzero(np.ravel(condition)) instead)
- mlab.longest_contiguous_ones
- mlab.longest_ones
- mlab.PCA
- mlab.prctile (use numpy.percentile instead)
- mlab.prctile_rank
- mlab.center_matrix
- mlab.rk4 (use scipy.integrate.ode instead)
- mlab.bivariate_normal
- mlab.get_xyz_where
- mlab.get_sparse_matrix
- mlab.dist (use numpy.hypot instead)
- mlab.dist_point_to_segment
- mlab.griddata (use scipy.interpolate.griddata)
- mlab.less_simple_linear_interpolation (use numpy.interp)
- mlab.slopes
- mlab.stineman_interp
- mlab.segments_intersect
- mlab.fftsurr
- mlab.offset_line
- mlab.quad2cubic
- mlab.vector_lengths
- mlab.distances_along_curve
- mlab.path_length
- mlab.cross_from_above

- mlab.cross_from_below
- mlab.contiguous_regions (use *cbook.contiguous_regions* instead)
- mlab.is_closed_polygon
- mlab.poly_between
- mlab.poly_below
- mlab.inside_poly
- mlab.csv2rec
- mlab.rec2csv (use numpy.recarray.tofile instead)
- mlab.rec2text (use numpy.recarray.tofile instead)
- mlab.rec_summarize
- mlab.rec_join
- mlab.recs_join
- mlab.rec_groupby
- mlab.rec_keep_fields
- mlab.rec_drop_fields
- mlab.rec_append_fields
- mlab.csvformat_factory
- mlab.get_formatd
- mlab.FormatDatetime (use datetime.datetime.strftime instead)
- mlab.FormatDate (use datetime.date.strftime instead)
- mlab.FormatMillions, mlab.FormatThousands, mlab.FormatPercent, mlab.FormatBool, mlab.FormatInt, mlab.FormatFloat, mlab.FormatFormatStr, mlab.FormatString, mlab.FormatObj
- mlab.donothing_callback

pylab removals

Lots of code inside the *matplotlib.mlab* module which was deprecated in Matplotlib 2.2 has been removed. This means the following functions are no longer available in the *pylab* module:

- amap
- base_repr
- binary_repr
- bivariate_normal

- `center_matrix`
- `csv2rec` (use `numpy.recarray.tofile` instead)
- `dist` (use `numpy.hypot` instead)
- `dist_point_to_segment`
- `distances_along_curve`
- `entropy` (use `scipy.stats.entropy` instead)
- `exp_safe` (use `numpy.exp` instead)
- `fftsurr`
- `find` (use `np.nonzero(np.ravel(condition))` instead)
- `frange` (use `numpy.arange` instead)
- `get_sparse_matrix`
- `get_xyz_where`
- `griddata` (use `scipy.interpolate.griddata` instead)
- `identity` (use `numpy.identity` instead)
- `inside_poly`
- `is_closed_polygon`
- `ispower2`
- `isvector`
- `l1norm` (use `numpy.linalg.norm(a, ord=1)` instead)
- `l2norm` (use `numpy.linalg.norm(a, ord=2)` instead)
- `log2` (use `numpy.log2` instead)
- `longest_contiguous_ones`
- `longest_ones`
- `movavg`
- `norm_flat` (use `numpy.linalg.norm(a.flat, ord=2)` instead)
- `normpdf` (use `scipy.stats.norm.pdf` instead)
- `path_length`
- `poly_below`
- `poly_between`
- `prctile` (use `numpy.percentile` instead)
- `prctile_rank`
- `rec2csv` (use `numpy.recarray.tofile` instead)

- rec_append_fields
- rec_drop_fields
- rec_join
- rk4 (use scipy.integrate.ode instead)
- rms_flat
- segments_intersect
- slopes
- stineman_interp
- vector_lengths

mplot3d changes

Voxel shading

Axes3D.voxels now shades the resulting voxels; for more details see What's new. The previous behavior can be achieved by passing

```
ax.voxels(..., shade=False)
```

Equal aspect axes disabled

Setting the aspect on 3D axes previously returned non-sensical results (e.g. see #1077). Calling `ax.set_aspect('equal')` or `ax.set_aspect(num)` on a 3D axes now raises a `NotImplementedError`.

Poly3DCollection.set_zsort

Poly3DCollection.set_zsort no longer silently ignores invalid inputs, or `False` (which was always broken). Passing `True` to mean `"average"` is deprecated.

Testing

The `--no-network` flag to `tests.py` has been removed (no test requires internet access anymore). If it is desired to disable internet access both for old and new versions of Matplotlib, use `tests.py -m 'not network'` (which is now a no-op).

The image comparison test decorators now skip (rather than xfail) the test for uncomparable formats. The affected decorators are *image_comparison* and *check_figures_equal*. The deprecated `ImageComparisonTest` class is likewise changed.

Dependency changes

NumPy

Matplotlib 3.1 now requires NumPy>=1.11.

ghostscript

Support for ghostscript 8.60 (released in 2007) has been removed. The oldest supported version of ghostscript is now 9.0 (released in 2010).

Mathtext changes

- In constructs such as `"$1~2$"`, mathtext now interprets the tilde as a space, consistently with TeX (this was previously a parse error).

Deprecations

- The `\stackrel` mathtext command has been deprecated (it behaved differently from LaTeX's `\stackrel`. To stack two mathtext expressions, use `\genfrac{left-delim}{right-delim}{fraction-bar-thickness}{}{top}{bottom}`.
- The `\mathcircled` mathtext command (which is not a real TeX command) is deprecated. Directly use unicode characters (e.g. `"\N{CIRCLED LATIN CAPITAL LETTER A}"` or `"\u24b6"`) instead.
- Support for setting `rcParams["mathtext.default"]` (default: `'it'`) to circled is deprecated.

Signature deprecations

The following signature related behaviours are deprecated:

- The *withdash* keyword argument to *Axes.text()*. Consider using *Axes.annotate()* instead.
- Passing (n, 1)-shaped error arrays to *Axes.errorbar()*, which was not documented and did not work for `n = 2`. Pass a 1D array instead.
- The *frameon* kwarg to *savefig* and the `rcParams["savefig.frameon"]` rcParam. To emulate `frameon = False`, set *facecolor* to fully transparent (`"none"`, or `(0, 0, 0, 0)`).
- Passing a non-1D (typically, (n, 1)-shaped) input to *Axes.pie*. Pass a 1D array instead.

- The *TextPath* constructor used to silently drop ignored arguments; this behavior is deprecated.
- The *usetex* parameter of *TextToPath.get_text_path* is deprecated and folded into the *ismath* parameter, which can now take the values `False`, `True`, and `"TeX"`, consistently with other low-level text processing functions.
- Passing `'normal'` to *axes.Axes.axis()* is deprecated, use `ax.axis('auto')` instead.
- Passing the *block* argument of *pyplot.show* positionally is deprecated; it should be passed by keyword.
- When using the nbagg backend, *pyplot.show* used to silently accept and ignore all combinations of positional and keyword arguments. This behavior is deprecated.
- The unused *shape* and *imlim* parameters to *Axes.imshow* are deprecated. To avoid triggering the deprecation warning, the *filternorm*, *filterrad*, *resample*, and *url* arguments should be passed by keyword.
- The *interp_at_native* parameter to *BboxImage*, which has had no effect since Matplotlib 2.0, is deprecated.
- All arguments to the `cbook.deprecated` decorator and `cbook.warn_deprecated` function, except the first one (the version where the deprecation occurred), are now keyword-only. The goal is to avoid accidentally setting the "message" argument when the "name" (or "alternative") argument was intended, as this has repeatedly occurred in the past.
- The arguments of `matplotlib.testing.compare.calculate_rms` have been renamed from `expectedImage, actualImage`, to `expected_image, actual_image`.
- Passing positional arguments to *Axis.set_ticklabels* beyond *ticklabels* itself has no effect, and support for them is deprecated.
- Passing `shade=None` to *plot_surface* is deprecated. This was an unintended implementation detail with the same semantics as `shade=False`. Please use the latter code instead.
- *matplotlib.ticker.MaxNLocator* and its *set_params* method will issue a warning on unknown keyword arguments instead of silently ignoring them. Future versions will raise an error.

Changes in parameter names

- The *arg* parameter to *matplotlib.use* has been renamed to *backend*.

  This will only affect cases where that parameter has been set as a keyword argument. The common usage pattern as a positional argument `matplotlib.use('Qt5Agg')` is not affected.
- The *normed* parameter to *Axes.hist2d* has been renamed to *density*.
- The *s* parameter to *Annotation* (and indirectly *Axes.annotate*) has been renamed to *text*.

- The *tolerence* parameter to `bezier.find_bezier_t_intersecting_with_closedpath`, `bezier.split_bezier_intersecting_with_closedpath`, `bezier.find_r_to_boundary_of_closedpath`, `bezier.split_path_inout` and `bezier.check_if_parallel` has been renamed to *tolerance*.

In each case, the old parameter name remains supported (it cannot be used simultaneously with the new name), but support for it will be dropped in Matplotlib 3.3.

Class/method/attribute deprecations

Support for custom backends that do not provide a *GraphicsContextBase.set_hatch_color* method is deprecated. We suggest that custom backends let their `GraphicsContext` class inherit from *GraphicsContextBase*, to at least provide stubs for all required methods.

- `spine.Spine.is_frame_like`

This has not been used in the codebase since its addition in 2009.

- `axis3d.Axis.get_tick_positions`

  This has never been used internally, there is no equivalent method exists on the 2D Axis classes, and despite the similar name, it has a completely different behavior from the 2D Axis' `axis.Axis.get_ticks_position` method.

- `.backend_pgf.LatexManagerFactory`
- `mpl_toolkits.axisartist.axislines.SimpleChainedObjects`
- `mpl_toolkits.Axes.AxisDict`

Internal Helper Functions

- `checkdep_dvipng`
- `checkdep_ghostscript`
- `checkdep_pdftops`
- `checkdep_inkscape`
- `ticker.decade_up`
- `ticker.decade_down`
- `cbook.dedent`
- `docstring.Appender`
- `docstring.dedent`
- `docstring.copy_dedent`

Use the standard library's docstring manipulation tools instead, such as `inspect.cleandoc` and `inspect.getdoc`.

- `matplotlib.scale.get_scale_docs()`
- `matplotlib.pyplot.get_scale_docs()`

These are considered internal and will be removed from the public API in a future version.

- `projections.process_projection_requirements`
- `backend_ps.PsBackendHelper`
- `backend_ps.ps_backend_helper`,
- `cbook.iterable`
- `cbook.get_label`
- `cbook.safezip` Manually check the lengths of the inputs instead, or rely on NumPy to do it.
- `cbook.is_hashable` Use `isinstance(..., collections.abc.Hashable)` instead.
- The `.backend_bases.RendererBase.strip_math`. Use *`cbook.strip_math`* instead.

Multiple internal functions that were exposed as part of the public API of *`mpl_toolkits.mplot3d`* are deprecated,

mpl_toolkits.mplot3d.art3d

- `mpl_toolkits.mplot3d.art3d.norm_angle`
- `mpl_toolkits.mplot3d.art3d.norm_text_angle`
- `mpl_toolkits.mplot3d.art3d.path_to_3d_segment`
- `mpl_toolkits.mplot3d.art3d.paths_to_3d_segments`
- `mpl_toolkits.mplot3d.art3d.path_to_3d_segment_with_codes`
- `mpl_toolkits.mplot3d.art3d.paths_to_3d_segments_with_codes`
- `mpl_toolkits.mplot3d.art3d.get_patch_verts`
- `mpl_toolkits.mplot3d.art3d.get_colors`
- `mpl_toolkits.mplot3d.art3d.zalpha`

mpl_toolkits.mplot3d.proj3d

- `mpl_toolkits.mplot3d.proj3d.line2d`
- `mpl_toolkits.mplot3d.proj3d.line2d_dist`
- `mpl_toolkits.mplot3d.proj3d.line2d_seg_dist`
- `mpl_toolkits.mplot3d.proj3d.mod`
- `mpl_toolkits.mplot3d.proj3d.proj_transform_vec`

- `mpl_toolkits.mplot3d.proj3d.proj_transform_vec_clip`
- `mpl_toolkits.mplot3d.proj3d.vec_pad_ones`
- `mpl_toolkits.mplot3d.proj3d.proj_trans_clip_points`

If your project relies on these functions, consider vendoring them.

Font Handling

- `backend_pdf.RendererPdf.afm_font_cache`
- `backend_ps.RendererPS.afmfontd`
- `font_manager.OSXInstalledFonts`
- `.TextToPath.glyph_to_path` (Instead call `font.get_path()` and manually transform the path.)

Date related functions

- `dates.seconds()`
- `dates.minutes()`
- `dates.hours()`
- `dates.weeks()`
- `dates.strpdate2num`
- `dates.bytespdate2num`

These are brittle in the presence of locale changes. Use standard datetime parsers such as `time.strptime` or `dateutil.parser.parse`, and additionally call *`matplotlib.dates.date2num`* if you need to convert to Matplotlib's internal datetime representation; or use `dates.datestr2num`.

Axes3D

- *`axes3d.Axes3D.w_xaxis`*
- *`axes3d.Axes3D.w_yaxis`*
- *`axes3d.Axes3D.w_zaxis`*

Use `axes3d.Axes3D.xaxis`, `axes3d.Axes3D.yaxis` and `axes3d.Axes3D.zaxis` instead.

Testing

- `matplotlib.testing.decorators.switch_backend` decorator

Test functions should use `pytest.mark.backend`, and the mark will be picked up by the `matplotlib.testing.conftest.mpl_test_settings` fixture.

Quiver

- `.color` attribute of *`Quiver`* objects

Instead, use (as for any *`Collection`*) the `get_facecolor` method. Note that setting to the `.color` attribute did not update the quiver artist, whereas calling `set_facecolor` does.

GUI / backend details

- `.get_py2exe_datafiles`
- `.tk_window_focus`
- `backend_gtk3.FileChooserDialog`
- `backend_gtk3.NavigationToolbar2GTK3.get_filechooser`
- `backend_gtk3.SaveFigureGTK3.get_filechooser`
- `NavigationToolbar2QT.adj_window` attribute. This is unused and always `None`.
- `backend_wx.IDLE_DELAY` global variable This is unused and only relevant to the now removed wx "idling" code (note that as it is a module-level global, no deprecation warning is emitted when accessing it).
- `mlab.demean`
- `backend_gtk3cairo.FigureCanvasGTK3Cairo`,
- `backend_wx.debug_on_error`, `backend_wx.fake_stderr`, `backend_wx.raise_msg_to_str`, `backend_wx.MenuButtonWx`, `backend_wx.PrintoutWx`,
- `matplotlib.backends.qt_editor.formlayout` module

This module is a vendored, modified version of the official formlayout module available on PyPI. Install that module separately if you need it.

- `GraphicsContextPS.shouldstroke`

Transforms / scales

- `LogTransformBase`
- `Log10Transform`
- `Log2Transform`,
- `NaturalLogTransformLog`
- `InvertedLogTransformBase`
- `InvertedLog10Transform`
- `InvertedLog2Transform`
- `InvertedNaturalLogTransform`

These classes defined in *matplotlib.scale* are deprecated. As a replacement, use the general *LogTransform* and *InvertedLogTransform* classes, whose constructors take a *base* argument.

Locators / Formatters

- `OldScalarFormatter.pprint_val`
- `ScalarFormatter.pprint_val`
- `LogFormatter.pprint_val`

These are helper methods that do not have a consistent signature across formatter classes.

Path tools

- `path.get_paths_extents`

Use *get_path_collection_extents* instead.

- `.Path.has_nonfinite` attribute

Use `not np.isfinite(path.vertices).all()` instead.

- `bezier.find_r_to_boundary_of_closedpath` function is deprecated

This has always returned None instead of the requested radius.

Text

- `text.TextWithDash`
- `Text.is_math_text`
- `TextPath.is_math_text`
- `TextPath.text_get_vertices_codes` (As an alternative, construct a new `TextPath` object.)

Unused attributes

- `NavigationToolbar2QT.buttons`
- `Line2D.verticalOffset`
- `Quiver.keytext`
- `Quiver.keyvec`
- `SpanSelector.buttonDown`

These are unused and never updated.

Sphinx extensions

- `matplotlib.sphinxext.mathmpl.math_directive`
- `matplotlib.sphinxext.plot_directive.plot_directive`

This is because the `matplotlib.sphinxext.mathmpl` and `matplotlib.sphinxext.plot_directive` interfaces have changed from the (Sphinx-)deprecated function-based interface to a class-based interface; this should not affect end users.

- `mpl_toolkits.axisartist.axis_artist.UnimplementedException`

Environmental Variables

- The `MATPLOTLIBDATA` environment variable

Axis

- `Axis.iter_ticks`

This only served as a helper to the private `Axis._update_ticks`

Undeprecations

The following API elements have bee un-deprecated:

- The *obj_type* kwarg to the `cbook.deprecated` decorator.
- *xmin*, *xmax* kwargs to *Axes.set_xlim* and *ymin*, *ymax* kwargs to *Axes.set_ylim*

New features

`Text` now has a `c` alias for the `color` property

For consistency with *Line2D*, the *Text* class has gained the `c` alias for the `color` property. For example, one can now write

```
ax.text(.5, .5, "foo", c="red")
```

`Cn` colors now support `n>=10`

It is now possible to go beyond the tenth color in the property cycle using `Cn` syntax, e.g.

```
plt.plot([1, 2], color="C11")
```

now uses the 12th color in the cycle.

Note that previously, a construct such as:

```
plt.plot([1, 2], "C11")
```

would be interpreted as a request to use color `C1` and marker `1` (an "inverted Y"). To obtain such a plot, one should now use

```
plt.plot([1, 2], "1C1")
```

(so that the first "1" gets correctly interpreted as a marker specification), or, more explicitly:

```
plt.plot([1, 2], marker="1", color="C1")
```

New `Formatter.format_ticks` method

The *Formatter* class gained a new *format_ticks* method, which takes the list of all tick locations as a single argument and returns the list of all formatted values. It is called by the axis tick handling code and, by default, first calls *set_locs* with all locations, then repeatedly calls `__call__` for each location.

Tick-handling code in the codebase that previously performed this sequence (*set_locs* followed by repeated `__call__`) have been updated to use *format_ticks*.

format_ticks is intended to be overridden by *Formatter* subclasses for which the formatting of a tick value depends on other tick values, such as *ConciseDateFormatter*.

Added support for RGB(A) images in pcolorfast

pcolorfast now accepts 3D images (RGB or RGBA) arrays if the X and Y specifications allow image or pcolorimage rendering; they remain unsupported by the more general quadmesh rendering

Invalid inputs

Passing invalid locations to *legend* and *table* used to fallback on a default location. This behavior is deprecated and will throw an exception in a future version.

offsetbox.AnchoredText is unable to handle the *horizontalalignment* or *verticalalignment* kwargs, and used to ignore them with a warning. This behavior is deprecated and will throw an exception in a future version.

Passing steps less than 1 or greater than 10 to *MaxNLocator* used to result in undefined behavior. It now throws a `ValueError`.

The signature of the (private) `Axis._update_ticks` has been changed to not take the renderer as argument anymore (that argument is unused).

16.1.6 API Changes for 3.0.1

tight_layout.auto_adjust_subplotpars can return `None` now if the new subplotparams will collapse axes to zero width or height. This prevents `tight_layout` from being executed. Similarly *tight_layout.get_tight_layout_figure* will return None.

To improve import (startup) time, private modules are now imported lazily. These modules are no longer available at these locations:

- `matplotlib.backends.backend_agg._png`
- `matplotlib.contour._contour`
- `matplotlib.image._png`

- matplotlib.mathtext._png
- matplotlib.testing.compare._png
- matplotlib.texmanager._png
- matplotlib.tri.triangulation._tri
- matplotlib.tri.triangulation._qhull
- matplotlib.tri.tricontour._tri
- matplotlib.tri.trifinder._tri

16.1.7 API Changes for 3.0.0

Drop support for python 2

Matplotlib 3 only supports python 3.5 and higher.

Changes to backend loading

Failure to load backend modules (`macosx` on non-framework builds and `gtk3` when running headless) now raises `ImportError` (instead of `RuntimeError` and `TypeError`, respectively).

Third-party backends that integrate with an interactive framework are now encouraged to define the `required_interactive_framework` global value to one of the following values: "qt5", "qt4", "gtk3", "wx", "tk", or "macosx". This information will be used to determine whether it is possible to switch from a backend to another (specifically, whether they use the same interactive framework).

`Axes.hist2d` now uses `pcolormesh` instead of `pcolorfast`

Axes.hist2d now uses *pcolormesh* instead of *pcolorfast*, which will improve the handling of log-axes. Note that the returned *image* now is of type *QuadMesh* instead of *AxesImage*.

`matplotlib.axes.Axes.get_tightbbox` now includes all artists

For Matplotlib 3.0, *all* artists are now included in the bounding box returned by *matplotlib.axes.Axes.get_tightbbox*.

matplotlib.axes.Axes.get_tightbbox adds a new kwarg `bbox_extra_artists` to manually specify the list of artists on the axes to include in the tight bounding box calculation.

Layout tools like *Figure.tight_layout*, `constrained_layout`, and `fig.savefig('fname.png', bbox_inches="tight")` use *matplotlib.axes.Axes.get_tightbbox* to determine the bounds of each axes on a figure and adjust spacing between axes.

In Matplotlib 2.2 `get_tightbbox` started to include legends made on the axes, but still excluded some other artists, like text that may overspill an axes. This has been expanded to include *all* artists.

This new default may be overridden in either of three ways:

1. Make the artist to be excluded a child of the figure, not the axes. E.g., call `fig.legend()` instead of `ax.legend()` (perhaps using *get_legend_handles_labels* to gather handles and labels from the parent axes).
2. If the artist is a child of the axes, set the artist property `artist.set_in_layout(False)`.
3. Manually specify a list of artists in the new kwarg `bbox_extra_artists`.

`Text.set_text` with string argument `None` sets string to empty

Text.set_text when passed a string value of `None` would set the string to `"None"`, so subsequent calls to *Text.get_text* would return the ambiguous `"None"` string.

This change sets text objects passed `None` to have empty strings, so that *Text.get_text* returns an empty string.

`Axes3D.get_xlim`, `get_ylim` and `get_zlim` now return a tuple

They previously returned an array. Returning a tuple is consistent with the behavior for 2D axes.

`font_manager.list_fonts` now follows the platform's casefolding semantics

i.e., it behaves case-insensitively on Windows only.

`bar` / `barh` no longer accepts `left` / `bottom` as first named argument

These arguments were renamed in 2.0 to `x` / `y` following the change of the default alignment from `edge` to `center`.

Different exception types for undocumented options

- Passing `style='comma'` to *`ticklabel_format()`* was never supported. It now raises `ValueError` like all other unsupported styles, rather than `NotImplementedError`.
- Passing the undocumented `xmin` or `xmax` arguments to *`set_xlim()`* would silently override the `left` and `right` arguments. *`set_ylim()`* and the 3D equivalents (e.g. *`set_zlim3d`*) had a corresponding problem. A `TypeError` will be raised if they would override the earlier limit arguments. In 3.0 these were kwargs were deprecated, but in 3.1 the deprecation was undone.

Improved call signature for `Axes.margins`

`Axes.margins` and *`Axes3D.margins`* no longer accept arbitrary keywords. `TypeError` will therefore be raised if unknown kwargs are passed; previously they would be silently ignored.

If too many positional arguments are passed, `TypeError` will be raised instead of `ValueError`, for consistency with other call-signature violations.

`Axes3D.margins` now raises `TypeError` instead of emitting a deprecation warning if only two positional arguments are passed. To supply only `x` and `y` margins, use keyword arguments.

Explicit arguments instead of *args, **kwargs

PEP 3102 describes keyword-only arguments, which allow Matplotlib to provide explicit call signatures - where we previously used `*args, **kwargs` and `kwargs.pop`, we can now expose named arguments. In some places, unknown kwargs were previously ignored but now raise `TypeError` because `**kwargs` has been removed.

- *`matplotlib.axes.Axes.stem()`* no longer accepts unknown keywords, and raises `TypeError` instead of emitting a deprecation.
- *`matplotlib.axes.Axes.stem()`* now raises TypeError when passed unhandled positional arguments. If two or more arguments are passed (ie X, Y, [linefmt], ...) and Y cannot be cast to an array, an error will be raised instead of treating X as Y and Y as linefmt.
- *`mpl_toolkits.axes_grid1.axes_divider.SubplotDivider`* raises `TypeError` instead of `Exception` when passed unknown kwargs.

Cleanup decorators and test classes no longer destroy warnings filter on exit

The decorators and classes in matplotlib.testing.decorators no longer destroy the warnings filter on exit. Instead, they restore the warnings filter that existed before the test started using `warnings.catch_warnings`.

Non-interactive FigureManager classes are now aliases of FigureManagerBase

The `FigureManagerPdf`, `FigureManagerPS`, and `FigureManagerSVG` classes, which were previously empty subclasses of *FigureManagerBase* (i.e., not adding or overriding any attribute or method), are now direct aliases for *FigureManagerBase*.

Change to the output of `image.thumbnail`

When called with `preview=False`, *image.thumbnail* previously returned an figure whose canvas class was set according to the output file extension. It now returns a figure whose canvas class is the base *FigureCanvasBase* (and relies on *FigureCanvasBase.print_figure*) to handle the canvas switching properly).

As a side effect of this change, *image.thumbnail* now also supports .ps, .eps, and .svgz output.

`FuncAnimation` now draws artists according to their zorder when blitting

FuncAnimation now draws artists returned by the user- function according to their zorder when using blitting, instead of using the order in which they are being passed. However, note that only zorder of passed artists will be respected, as they are drawn on top of any existing artists (see #11369).

Contour color autoscaling improvements

Selection of contour levels is now the same for contour and contourf; previously, for contour, levels outside the data range were deleted. (Exception: if no contour levels are found within the data range, the `levels` attribute is replaced with a list holding only the minimum of the data range.)

When contour is called with levels specified as a target number rather than a list, and the 'extend' kwarg is used, the levels are now chosen such that some data typically will fall in the extended range.

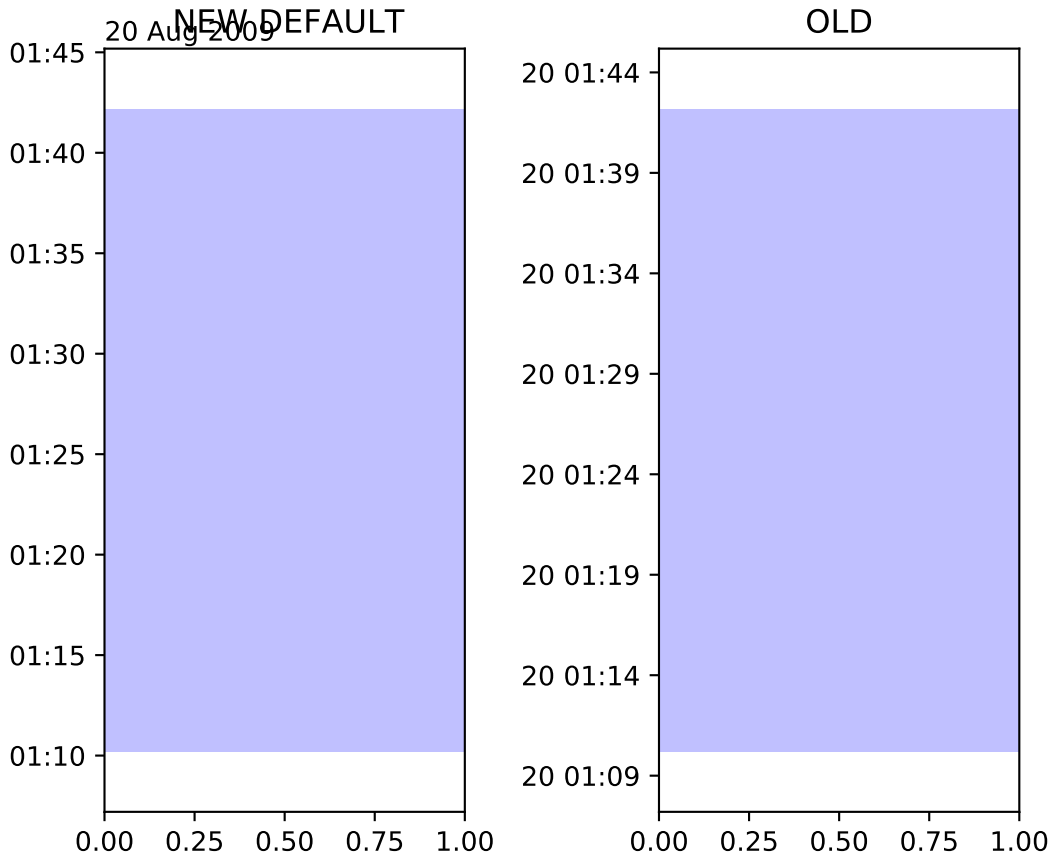
When contour is called with a *LogNorm* or a *LogLocator*, it will now select colors using the geometric mean rather than the arithmetic mean of the contour levels.

Streamplot last row and column fixed

A bug was fixed where the last row and column of data in *streamplot* were being dropped.

Changed default `AutoDateLocator` kwarg *interval_multiples* to `True`

The default value of the tick locator for dates, *dates.AutoDateLocator* kwarg *interval_multiples* was set to `False` which leads to not-nice looking automatic ticks in many instances. The much nicer `interval_multiples=True` is the new default. See below to get the old behavior back:



`Axes.get_position` now returns actual position if aspect changed

Axes.get_position used to return the original position unless a draw had been triggered or *Axes.apply_aspect* had been called, even if the kwarg *original* was set to `False`. Now *Axes.apply_aspect* is called so `ax.get_position()` will return the new modified position. To get the old behavior use `ax.get_position(original=True)`.

The ticks for colorbar now adjust for the size of the colorbar

Colorbar ticks now adjust for the size of the colorbar if the colorbar is made from a mappable that is not a contour or doesn't have a BoundaryNorm, or boundaries are not specified. If boundaries, etc are specified, the colorbar maintains the original behavior.

Colorbar for log-scaled hexbin

When using *hexbin* and plotting with a logarithmic color scale, the colorbar ticks are now correctly log scaled. Previously the tick values were linear scaled log(number of counts).

PGF backend now explicitly makes black text black

Previous behavior with the pgf backend was for text specified as black to actually be the default color of whatever was rendering the pgf file (which was of course usually black). The new behavior is that black text is black, regardless of the default color. However, this means that there is no way to fall back on the default color of the renderer.

Blacklisted rcparams no longer updated by `rcdefaults`, `rc_file_defaults`, `rc_file`

The rc modifier functions *rcdefaults*, *rc_file_defaults* and *rc_file* now ignore rcParams in the `matplotlib.style.core.STYLE_BLACKLIST` set. In particular, this prevents the `backend` and `interactive` rcParams from being incorrectly modified by these functions.

`CallbackRegistry` now stores callbacks using stdlib's `weakref.WeakMethods`

In particular, this implies that `CallbackRegistry.callbacks[signal]` is now a mapping of callback ids to `weakref.WeakMethod`s (i.e., they need to be first called with no arguments to retrieve the method itself).

Changes regarding the text.latex.unicode rcParam

The rcParam now defaults to True and is deprecated (i.e., in future versions of Matplotlib, unicode input will always be supported).

Moreover, the underlying implementation now uses `\usepackage[utf8]{inputenc}` instead of `\usepackage{ucs}\usepackage[utf8x]{inputenc}`.

Return type of ArtistInspector.get_aliases changed

`ArtistInspector.get_aliases` previously returned the set of aliases as `{fullname: {alias1: None, alias2: None, ...}}`. The dict-to-None mapping was used to simulate a set in earlier versions of Python. It has now been replaced by a set, i.e. `{fullname: {alias1, alias2, ...}}`.

This value is also stored in `ArtistInspector.aliasd`, which has likewise changed.

Removed `pytz` as a dependency

Since `dateutil` and `pytz` both provide time zones, and matplotlib already depends on `dateutil`, matplotlib will now use `dateutil` time zones internally and drop the redundant dependency on `pytz`. While `dateutil` time zones are preferred (and currently recommended in the Python documentation), the explicit use of `pytz` zones is still supported.

Deprecations

Modules

The following modules are deprecated:

- `matplotlib.compat.subprocess`. This was a python 2 workaround, but all the functionality can now be found in the python 3 standard library `subprocess`.
- `matplotlib.backends.wx_compat`. Python 3 is only compatible with wxPython 4, so support for wxPython 3 or earlier can be dropped.

Classes, methods, functions, and attributes

The following classes, methods, functions, and attributes are deprecated:

- `RcParams.msg_depr`, `RcParams.msg_depr_ignore`, `RcParams.msg_depr_set`, `RcParams.msg_obsolete`, `RcParams.msg_backend_obsolete`
- `afm.parse_afm`
- `backend_pdf.PdfFile.texFontMap`
- `backend_pgf.get_texcommand`
- `backend_ps.get_bbox`
- `backend_qt5.FigureCanvasQT.keyAutoRepeat` (directly check `event.guiEvent.isAutoRepeat()` in the event handler to decide whether to handle autorepeated key presses).
- `backend_qt5.error_msg_qt`, `backend_qt5.exception_handler`
- `backend_wx.FigureCanvasWx.macros`
- `backends.pylab_setup`
- `cbook.GetRealpathAndStat`, `cbook.Locked`
- `cbook.is_numlike` (use `isinstance(..., numbers.Number)` instead), `cbook.listFiles`, `cbook.unicode_safe`
- `container.Container.set_remove_method`,
- `contour.ContourLabeler.cl`, `.cl_xy`, and `.cl_cvalues`
- `dates.DateFormatter.strftime_pre_1900`, `dates.DateFormatter.strftime`
- `font_manager.TempCache`
- `image._ImageBase.iterpnames`, use the `interpolation_names` property instead. (this affects classes that inherit from `_ImageBase` including *FigureImage*, *BboxImage*, and *AxesImage*)
- `mathtext.unichr_safe` (use `chr` instead)
- `patches.Polygon.xy`
- `table.Table.get_child_artists` (use `get_children` instead)
- `testing.compare.ImageComparisonTest`, `testing.compare.compare_float`
- `testing.decorators.CleanupTest`, `testing.decorators.skip_if_command_unavailable`
- `FigureCanvasQT.keyAutoRepeat` (directly check `event.guiEvent.isAutoRepeat()` in the event handler to decide whether to handle autorepeated key presses)
- `FigureCanvasWx.macros`

- `_ImageBase.iterpnames`, use the `interpolation_names` property instead. (this affects classes that inherit from `_ImageBase` including *FigureImage*, *BboxImage*, and *AxesImage*)
- `patches.Polygon.xy`
- `texmanager.dvipng_hack_alpha`
- `text.Annotation.arrow`
- **`Legend.draggable()`, in favor of** *Legend.set_draggable()*

  (`Legend.draggable` may be reintroduced as a property in future releases)
- `textpath.TextToPath.tex_font_map`
- `matplotlib.cbook.deprecation.mplDeprecation` will be removed in future versions. It is just an alias for *matplotlib.cbook.deprecation.MatplotlibDeprecationWarning*. Please use `matplotlib.cbook.MatplotlibDeprecationWarning` directly if necessary.
- The `matplotlib.cbook.Bunch` class has been deprecated. Instead, use `types.SimpleNamespace` from the standard library which provides the same functionality.
- `Axes.mouseover_set` is now a frozenset, and deprecated. Directly manipulate the artist's `.mouseover` attribute to change their mouseover status.

The following keyword arguments are deprecated:

- passing `verts` to `Axes.scatter` (use `marker` instead)
- passing `obj_type` to `cbook.deprecated`

The following call signatures are deprecated:

- passing a `wx.EvtHandler` as first argument to `backend_wx.TimerWx`

rcParams

The following rcParams are deprecated:

- `examples.directory` (use `datapath` instead)
- `pgf.debug` (the pgf backend relies on logging)
- `text.latex.unicode` (always True now)

marker styles

- Using `(n, 3)` as marker style to specify a circle marker is deprecated. Use `"o"` instead.
- Using `([(x0, y0), (x1, y1), ...], 0)` as marker style to specify a custom marker path is deprecated. Use `[(x0, y0), (x1, y1), ...]` instead.

Deprecation of `LocatableAxes` in toolkits

The `LocatableAxes` classes in toolkits have been deprecated. The base *Axes* classes provide the same functionality to all subclasses, thus these mixins are no longer necessary. Related functions have also been deprecated. Specifically:

- `mpl_toolkits.axes_grid1.axes_divider.LocatableAxesBase`: no specific replacement; use any other `Axes`-derived class directly instead.
- `mpl_toolkits.axes_grid1.axes_divider.locatable_axes_factory`: no specific replacement; use any other `Axes`-derived class directly instead.
- `mpl_toolkits.axes_grid1.axes_divider.Axes`: use *mpl_toolkits.axes_grid1.mpl_axes.Axes* directly.
- `mpl_toolkits.axes_grid1.axes_divider.LocatableAxes`: use *mpl_toolkits.axes_grid1.mpl_axes.Axes* directly.
- `mpl_toolkits.axisartist.axes_divider.Axes`: use *mpl_toolkits.axisartist.axislines.Axes* directly.
- `mpl_toolkits.axisartist.axes_divider.LocatableAxes`: use *mpl_toolkits.axisartist.axislines.Axes* directly.

Removals

Hold machinery

Setting or unsetting `hold` (*deprecated in version 2.0*) has now been completely removed. Matplotlib now always behaves as if `hold=True`. To clear an axes you can manually use *cla()*, or to clear an entire figure use *clf()*.

Removal of deprecated backends

Deprecated backends have been removed:

- GTKAgg
- GTKCairo
- GTK
- GDK

Deprecated APIs

The following deprecated API elements have been removed:

- The deprecated methods `knownfailureif` and `remove_text` have been removed from *matplotlib.testing.decorators*.
- The entire contents of `testing.noseclasses` have also been removed.
- `matplotlib.checkdep_tex`, `matplotlib.checkdep_xmllint`
- `backend_bases.IdleEvent`
- `cbook.converter`, `cbook.tostr`, `cbook.todatetime`, `cbook.todate`, `cbook.tofloat`, `cbook.toint`, `cbook.unique`, `cbook.is_string_like`, `cbook.is_sequence_of_strings`, `cbook.is_scalar`, `cbook.soundex`, `cbook.dict_delall`, `cbook.get_split_ind`, `cbook.wrap`, `cbook.get_recursive_filelist`, `cbook.pieces`, `cbook.exception_to_str`, `cbook.allequal`, `cbook.alltrue`, `cbook.onetrue`, `cbook.allpairs`, `cbook.finddir`, `cbook.reverse_dict`, `cbook.restrict_dict`, `cbook.issubclass_safe`, `cbook.recursive_remove`, `cbook.unmasked_index_ranges`, `cbook.Null`, `cbook.RingBuffer`, `cbook.Sorter`, `cbook.Xlator`,
- `font_manager.weight_as_number`, `font_manager.ttfdict_to_fnames`
- `pyplot.colors`, `pyplot.spectral`
- `rcsetup.validate_negative_linestyle`, `rcsetup.validate_negative_linestyle_legacy`,
- `testing.compare.verifiers`, `testing.compare.verify`
- `testing.decorators.knownfailureif`, `testing.decorators.ImageComparisonTest.remove_text`
- `tests.assert_str_equal`, `tests.test_tinypages.file_same`
- `texmanager.dvipng_hack_alpha`,
- `_AxesBase.axesPatch`, `_AxesBase.set_color_cycle`, `_AxesBase.get_cursor_props`, `_AxesBase.set_cursor_props`
- `_ImageBase.iterpnames`
- `FigureCanvasBase.start_event_loop_default`;

- `FigureCanvasBase.stop_event_loop_default`;
- `Figure.figurePatch`,
- `FigureCanvasBase.dynamic_update`, `FigureCanvasBase.idle_event`, `FigureCanvasBase.get_linestyle`, `FigureCanvasBase.set_linestyle`
- `FigureCanvasQTAggBase`
- `FigureCanvasQTAgg.blitbox`
- `FigureCanvasTk.show` (alternative: `FigureCanvasTk.draw`)
- `FigureManagerTkAgg` (alternative: `FigureManagerTk`)
- `NavigationToolbar2TkAgg` (alternative: `NavigationToolbar2Tk`)
- `backend_wxagg.Toolbar` (alternative: `backend_wxagg.NavigationToolbar2WxAgg`)
- `RendererAgg.debug()`
- passing non-numbers to `EngFormatter.format_eng`
- passing `frac` to `PolarAxes.set_theta_grids`
- any mention of idle events

The following API elements have been removed:

- `backend_cairo.HAS_CAIRO_CFFI`
- `sphinxext.sphinx_version`

Proprietary sphinx directives

The matplotlib documentation used the proprietary sphinx directives `.. htmlonly::`, and `.. latexonly::`. These have been replaced with the standard sphinx directives `.. only:: html` and `.. only:: latex`. This change will not affect any users. Only down-stream package maintainers, who have used the proprietary directives in their docs, will have to switch to the sphinx directives.

lib/mpl_examples symlink

The symlink from lib/mpl_examples to ../examples has been removed. This is not installed as an importable package and should not affect end users, however this may require down-stream packagers to adjust. The content is still available top-level examples directory.

16.1.8 API Changes in 2.2.0

New dependency

kiwisolver is now a required dependency to support the new constrained_layout, see *Constrained Layout Guide* for more details.

Deprecations

Classes, functions, and methods

The unused and untested `Artist.onRemove` and `Artist.hitlist` methods have been deprecated.

The now unused `mlab.less_simple_linear_interpolation` function is deprecated.

The unused `ContourLabeler.get_real_label_width` method is deprecated.

The unused `FigureManagerBase.show_popup` method is deprecated. This introduced in e945059b327d42a99938b939a1be867fa023e7ba in 2005 but never built out into any of the backends.

`backend_tkagg.AxisMenu` is deprecated, as it has become unused since the removal of "classic" toolbars.

Changed function signatures

kwarg `fig` to *GridSpec.get_subplot_params* is deprecated, use `figure` instead.

Using *pyplot.axes* with an *Axes* as argument is deprecated. This sets the current axes, i.e. it has the same effect as *pyplot.sca*. For clarity `plt.sca(ax)` should be preferred over `plt.axes(ax)`.

Using strings instead of booleans to control grid and tick visibility is deprecated. Using `"on"`, `"off"`, `"true"`, or `"false"` to control grid and tick visibility has been deprecated. Instead, use normal booleans (`True`/`False`) or boolean-likes. In the future, all non-empty strings may be interpreted as `True`.

When given 2D inputs with non-matching numbers of columns, *plot* currently cycles through the columns of the narrower input, until all the columns of the wider input have been plotted. This behavior is deprecated; in the future, only broadcasting (1 column to *n* columns) will be performed.

rcparams

The `rcParams["backend.qt4"]` and `rcParams["backend.qt5"]` rcParams were deprecated in version 2.2. In order to force the use of a specific Qt binding, either import that binding first, or set the `QT_API` environment variable.

Deprecation of the `nbagg.transparent` rcParam. To control transparency of figure patches in the nbagg (or any other) backend, directly set `figure.patch.facecolor`, or the `figure.facecolor` rcParam.

Deprecated `Axis.unit_data`

Use `Axis.units` (which has long existed) instead.

Removals

Function Signatures

Contouring no longer supports `legacy` corner masking. The deprecated `ContourSet.vmin` and `ContourSet.vmax` properties have been removed.

Passing `None` instead of `"none"` as format to *`errorbar`* is no longer supported.

The `bgcolor` keyword argument to `Axes` has been removed.

Modules, methods, and functions

The `matplotlib.finance`, `mpl_toolkits.exceltools` and `mpl_toolkits.gtktools` modules have been removed. `matplotlib.finance` remains available at https://github.com/matplotlib/mpl_finance.

The `mpl_toolkits.mplot3d.art3d.iscolor` function has been removed.

The `Axes.get_axis_bgcolor`, `Axes.set_axis_bgcolor`, `Bbox.update_from_data`, `Bbox.update_datalim_numerix`, `MaxNLocator.bin_boundaries` methods have been removed.

`mencoder` can no longer be used to encode animations.

The unused `FONT_SCALE` and `fontd` attributes of the *`RendererSVG`* class have been removed.

color maps

The `spectral` colormap has been removed. The `Vega*` colormaps, which were aliases for the `tab*` colormaps, have been removed.

rcparams

The following deprecated rcParams have been removed:

- `axes.color_cycle` (see `axes.prop_cycle`),
- `legend.isaxes`,
- `svg.embed_char_paths` (see `svg.fonttype`),
- `text.fontstyle`, `text.fontangle`, `text.fontvariant`, `text.fontweight`, `text.fontsize` (renamed to `text.style`, etc.),
- `tick.size` (renamed to `tick.major.size`).

Only accept string-like for Categorical input

Do not accept mixed string / float / int input, only strings are valid categoricals.

Removal of unused imports

Many unused imports were removed from the codebase. As a result, trying to import certain classes or functions from the "wrong" module (e.g. *Figure* from *matplotlib.backends.backend_agg* instead of *matplotlib.figure*) will now raise an `ImportError`.

`Axes3D.get_xlim`, `get_ylim` and `get_zlim` now return a tuple

They previously returned an array. Returning a tuple is consistent with the behavior for 2D axes.

Exception type changes

If *MovieWriterRegistry* can't find the requested *MovieWriter*, a more helpful `RuntimeError` message is now raised instead of the previously raised `KeyError`.

auto_adjust_subplotpars now raises `ValueError` instead of `RuntimeError` when sizes of input lists don't match

`Figure.set_figwidth` and `Figure.set_figheight` default *forward* to True

matplotlib.figure.Figure.set_figwidth and *matplotlib.figure.Figure.set_figheight* had the keyword argument `forward=False` by default, but *figure.Figure.set_size_inches* now defaults to `forward=True`. This makes these functions conistent.

Do not truncate svg sizes to nearest point

There is no reason to size the SVG out put in integer points, change to out putting floats for the *height*, *width*, and *viewBox* attributes of the *svg* element.

Fontsizes less than 1 pt are clipped to be 1 pt.

FreeType doesn't allow fonts to get smaller than 1 pt, so all Agg backends were silently rounding up to 1 pt. PDF (other vector backends?) were letting us write fonts that were less than 1 pt, but they could not be placed properly because position information comes from FreeType. This change makes it so no backends can use fonts smaller than 1 pt, consistent with FreeType and ensuring more consistent results across backends.

Changes to Qt backend class MRO

To support both Agg and cairo rendering for Qt backends all of the non-Agg specific code previously in `backend_qt5agg.FigureCanvasQTAggBase` has been moved to `backend_qt5.FigureCanvasQT` so it can be shared with the cairo implementation. The `FigureCanvasQTAggBase.paintEvent`, `FigureCanvasQTAggBase.blit`, and `FigureCanvasQTAggBase.print_figure` methods have moved to `FigureCanvasQTAgg.paintEvent()`, `FigureCanvasQTAgg.blit()`, and `FigureCanvasQTAgg.print_figure()`. The first two methods assume that the instance is also a `QWidget` so to use `FigureCanvasQTAggBase` it was required to multiple inherit from a `QWidget` sub-class.

Having moved all of its methods either up or down the class hierarchy `FigureCanvasQTAggBase` has been deprecated. To do this without warning and to preserve as much API as possible, `.backend_qt5agg.FigureCanvasQTAggBase` now inherits from `backend_qt5.FigureCanvasQTAgg`.

The MRO for `FigureCanvasQTAgg` and `FigureCanvasQTAggBase` used to be

```
[matplotlib.backends.backend_qt5agg.FigureCanvasQTAgg,
 matplotlib.backends.backend_qt5agg.FigureCanvasQTAggBase,
 matplotlib.backends.backend_agg.FigureCanvasAgg,
 matplotlib.backends.backend_qt5.FigureCanvasQT,
 PyQt5.QtWidgets.QWidget,
 PyQt5.QtCore.QObject,
 sip.wrapper,
 PyQt5.QtGui.QPaintDevice,
```

(continues on next page)

(continued from previous page)

```
 sip.simplewrapper,
 matplotlib.backend_bases.FigureCanvasBase,
 object]
```

and

```
[matplotlib.backends.backend_qt5agg.FigureCanvasQTAggBase,
 matplotlib.backends.backend_agg.FigureCanvasAgg,
 matplotlib.backend_bases.FigureCanvasBase,
 object]
```

respectively. They are now

```
[matplotlib.backends.backend_qt5agg.FigureCanvasQTAgg,
 matplotlib.backends.backend_agg.FigureCanvasAgg,
 matplotlib.backends.backend_qt5.FigureCanvasQT,
 PyQt5.QtWidgets.QWidget,
 PyQt5.QtCore.QObject,
 sip.wrapper,
 PyQt5.QtGui.QPaintDevice,
 sip.simplewrapper,
 matplotlib.backend_bases.FigureCanvasBase,
 object]
```

and

```
[matplotlib.backends.backend_qt5agg.FigureCanvasQTAggBase,
 matplotlib.backends.backend_qt5agg.FigureCanvasQTAgg,
 matplotlib.backends.backend_agg.FigureCanvasAgg,
 matplotlib.backends.backend_qt5.FigureCanvasQT,
 PyQt5.QtWidgets.QWidget,
 PyQt5.QtCore.QObject,
 sip.wrapper,
 PyQt5.QtGui.QPaintDevice,
 sip.simplewrapper,
 matplotlib.backend_bases.FigureCanvasBase,
 object]
```

`axes.Axes.imshow` clips RGB values to the valid range

When *axes.Axes.imshow* is passed an RGB or RGBA value with out-of-range values, it now logs a warning and clips them to the valid range. The old behaviour, wrapping back in to the range, often hid outliers and made interpreting RGB images unreliable.

GTKAgg and GTKCairo backends deprecated

The GTKAgg and GTKCairo backends have been deprecated. These obsolete backends allow figures to be rendered via the GTK+ 2 toolkit. They are untested, known to be broken, will not work with Python 3, and their use has been discouraged for some time. Instead, use the `GTK3Agg` and `GTK3Cairo` backends for rendering to GTK+ 3 windows.

16.1.9 API Changes in 2.1.2

`Figure.legend` no longer checks for repeated lines to ignore

matplotlib.figure.Figure.legend used to check if a line had the same label as an existing legend entry. If it also had the same line color or marker color legend didn't add a new entry for that line. However, the list of conditions was incomplete, didn't handle RGB tuples, didn't handle linewidths or linestyles etc.

This logic did not exist in *axes.Axes.legend*. It was included (erroneously) in Matplotlib 2.1.1 when the legend argument parsing was unified [#9324](https://github.com/matplotlib/matplotlib/pull/9324). This change removes that check in *axes.Axes.legend* again to restore the old behavior.

This logic has also been dropped from *Figure.legend*, where it was previously undocumented. Repeated lines with the same label will now each have an entry in the legend. If you do not want the duplicate entries, don't add a label to the line, or prepend the label with an underscore.

16.1.10 API Changes in 2.1.1

Default behavior of log scales reverted to clip <= 0 values

The change it 2.1.0 to mask in logscale by default had more disruptive changes than anticipated and has been reverted, however the clipping is now done in a way that fixes the issues that motivated changing the default behavior to `'mask'`.

As a side effect of this change, error bars which go negative now work as expected on log scales.

16.1.11 API Changes in 2.1.0

Default behavior of log scales changed to mask <= 0 values

Calling *matplotlib.axes.Axes.set_xscale* or *matplotlib.axes.Axes.set_yscale* now uses 'mask' as the default method to handle invalid values (as opposed to 'clip'). This means that any values <= 0 on a log scale will not be shown.

Previously they were clipped to a very small number and shown.

`matplotlib.cbook.CallbackRegistry.process()` suppresses exceptions by default

Matplotlib uses instances of *CallbackRegistry* as a bridge between user input event from the GUI and user callbacks. Previously, any exceptions raised in a user call back would bubble out of of the `process` method, which is typically in the GUI event loop. Most GUI frameworks simple print the traceback to the screen and continue as there is not always a clear method of getting the exception back to the user. However PyQt5 now exits the process when it receives an un-handled python exception in the event loop. Thus, *process()* now suppresses and prints tracebacks to stderr by default.

What *process()* does with exceptions is now user configurable via the `exception_handler` attribute and kwarg. To restore the previous behavior pass `None`

```
cb = CallbackRegistry(exception_handler=None)
```

A function which take and `Exception` as its only argument may also be passed

```
def maybe_reraise(exc):
    if isinstance(exc, RuntimeError):
        pass
    else:
        raise exc

cb = CallbackRegistry(exception_handler=maybe_reraise)
```

Improved toggling of the axes grids

The `g` key binding now switches the states of the `x` and `y` grids independently (by cycling through all four on/off combinations).

The new `G` key binding switches the states of the minor grids.

Both bindings are disabled if only a subset of the grid lines (in either direction) is visible, to avoid making irreversible changes to the figure.

Ticklabels are turned off instead of being invisible

Internally, the *Tick*'s `~matplotlib.axis.Tick.label1On` attribute is now used to hide tick labels instead of setting the visibility on the tick label objects. This improves overall performance and fixes some issues. As a consequence, in case those labels ought to be shown, *tick_params()* needs to be used, e.g.

```
ax.tick_params(labelbottom=True)
```

Removal of warning on empty legends

pyplot.legend used to issue a warning when no labeled artist could be found. This warning has been removed.

More accurate legend autopositioning

Automatic positioning of legends now prefers using the area surrounded by a *Line2D* rather than placing the legend over the line itself.

Cleanup of stock sample data

The sample data of stocks has been cleaned up to remove redundancies and increase portability. The `AAPL.dat.gz`, `INTC.dat.gz` and `aapl.csv` files have been removed entirely and will also no longer be available from *matplotlib.cbook.get_sample_data*. If a CSV file is required, we suggest using the `msft.csv` that continues to be shipped in the sample data. If a NumPy binary file is acceptable, we suggest using one of the following two new files. The `aapl.npy.gz` and `goog.npy` files have been replaced by `aapl.npz` and `goog.npz`, wherein the first column's type has changed from `datetime.date` to `numpy.datetime64` for better portability across Python versions. Note that Matplotlib does not fully support `numpy.datetime64` as yet.

Updated qhull to 2015.2

The version of qhull shipped with Matplotlib, which is used for Delaunay triangulation, has been updated from version 2012.1 to 2015.2.

Improved Delaunay triangulations with large offsets

Delaunay triangulations now deal with large x,y offsets in a better way. This can cause minor changes to any triangulations calculated using Matplotlib, i.e. any use of *matplotlib.tri.Triangulation* that requests that a Delaunay triangulation is calculated, which includes *matplotlib.pyplot.tricontour*, *matplotlib.pyplot.tricontourf*, *matplotlib.pyplot.tripcolor*, *matplotlib.pyplot.triplot*, `matplotlib.mlab.griddata` and *mpl_toolkits.mplot3d.axes3d.Axes3D.plot_trisurf*.

Use `backports.functools_lru_cache` instead of `functools32`

It's better maintained and more widely used (by pylint, jaraco, etc).

`cbook.is_numlike` only performs an instance check

`matplotlib.cbook.is_numlike` now only checks that its argument is an instance of `(numbers.Number, np.Number)`. In particular, this means that arrays are now not numlike.

Elliptical arcs now drawn between correct angles

The *matplotlib.patches.Arc* patch is now correctly drawn between the given angles.

Previously a circular arc was drawn and then stretched into an ellipse, so the resulting arc did not lie between *theta1* and *theta2*.

`-d$backend` no longer sets the backend

It is no longer possible to set the backend by passing `-d$backend` at the command line. Use the `MPLBACKEND` environment variable instead.

Path.intersects_bbox always treats the bounding box as filled

Previously, when `Path.intersects_bbox` was called with `filled` set to `False`, it would treat both the path and the bounding box as unfilled. This behavior was not well documented and it is usually not the desired behavior, since bounding boxes are used to represent more complex shapes located inside the bounding box. This behavior has now been changed: when `filled` is `False`, the path will be treated as unfilled, but the bounding box is still treated as filled. The old behavior was arguably an implementation bug.

When `Path.intersects_bbox` is called with `filled` set to `True` (the default value), there is no change in behavior. For those rare cases where `Path.intersects_bbox` was called

with `filled` set to `False` and where the old behavior is actually desired, the suggested workaround is to call `Path.intersects_path` with a rectangle as the path:

```
from matplotlib.path import Path
from matplotlib.transforms import Bbox, BboxTransformTo
rect = Path.unit_rectangle().transformed(BboxTransformTo(bbox))
result = path.intersects_path(rect, filled=False)
```

WX no longer calls generates `IdleEvent` events or calls `idle_event`

Removed unused private method `_onIdle` from `FigureCanvasWx`.

The `IdleEvent` class and `FigureCanvasBase.idle_event` method will be removed in 2.2

Correct scaling of `magnitude_spectrum()`

The functions *matplotlib.mlab.magnitude_spectrum()* and *matplotlib.pyplot.magnitude_spectrum()* implicitly assumed the sum of windowing function values to be one. In Matplotlib and Numpy the standard windowing functions are scaled to have maximum value of one, which usually results in a sum of the order of n/2 for a n-point signal. Thus the amplitude scaling `magnitude_spectrum()` was off by that amount when using standard windowing functions (Bug 8417). Now the behavior is consistent with *matplotlib.pyplot.psd()* and `scipy.signal.welch()`. The following example demonstrates the new and old scaling:

```
import matplotlib.pyplot as plt
import numpy as np

tau, n = 10, 1024  # 10 second signal with 1024 points
T = tau/n  # sampling interval
t = np.arange(n)*T

a = 4  # amplitude
x = a*np.sin(40*np.pi*t)  # 20 Hz sine with amplitude a

# New correct behavior: Amplitude at 20 Hz is a/2
plt.magnitude_spectrum(x, Fs=1/T, sides='onesided', scale='linear')

# Original behavior: Amplitude at 20 Hz is (a/2)*(n/2) for a Hanning window
w = np.hanning(n)  # default window is a Hanning window
plt.magnitude_spectrum(x*np.sum(w), Fs=1/T, sides='onesided', scale='linear')
```

Change to signatures of `bar()` & `barh()`

For 2.0 the *default value of *align** changed to `'center'`. However this caused the signature of *bar()* and *barh()* to be misleading as the first parameters were still *left* and *bottom* respectively:

```
bar(left, height, *, align='center', **kwargs)
barh(bottom, width, *, align='center', **kwargs)
```

despite behaving as the center in both cases. The methods now take `*args, **kwargs` as input and are documented to have the primary signatures of:

```
bar(x, height, *, align='center', **kwargs)
barh(y, width, *, align='center', **kwargs)
```

Passing *left* and *bottom* as keyword arguments to *bar()* and *barh()* respectively will warn. Support will be removed in Matplotlib 3.0.

Font cache as json

The font cache is now saved as json, rather than a pickle.

Invalid (Non-finite) Axis Limit Error

When using *set_xlim()* and *set_ylim()*, passing non-finite values now results in a `ValueError`. The previous behavior resulted in the limits being erroneously reset to `(-0.001, 0.001)`.

`scatter` and `Collection` offsets are no longer implicitly flattened

Collection (and thus both 2D *scatter* and 3D *scatter*) no longer implicitly flattens its offsets. As a consequence, `scatter`'s `x` and `y` arguments can no longer be 2+-dimensional arrays.

Deprecations

`GraphicsContextBase`'s `linestyle` property.

The `GraphicsContextBase.get_linestyle` and `GraphicsContextBase.set_linestyle` methods, which had no effect, have been deprecated. All of the backends Matplotlib ships use `GraphicsContextBase.get_dashes` and `GraphicsContextBase.set_dashes` which are more general. Third-party backends should also migrate to the `*_dashes` methods.

NavigationToolbar2.dynamic_update

Use *draw_idle()* method on the Canvas instance instead.

Testing

matplotlib.testing.noseclasses is deprecated and will be removed in 2.3

EngFormatter *num* arg as string

Passing a string as *num* argument when calling an instance of *matplotlib.ticker.EngFormatter* is deprecated and will be removed in 2.3.

mpl_toolkits.axes_grid module

All functionally from mpl_toolkits.axes_grid can be found in either *mpl_toolkits.axes_grid1* or *mpl_toolkits.axisartist*. Axes classes from mpl_toolkits.axes_grid based on Axis from *mpl_toolkits.axisartist* can be found in *mpl_toolkits.axisartist*.

Axes collision in Figure.add_axes

Adding an axes instance to a figure by using the same arguments as for a previous axes instance currently reuses the earlier instance. This behavior has been deprecated in Matplotlib 2.1. In a future version, a *new* instance will always be created and returned. Meanwhile, in such a situation, a deprecation warning is raised by *AxesStack*.

This warning can be suppressed, and the future behavior ensured, by passing a *unique* label to each axes instance. See the docstring of *add_axes()* for more information.

Additional details on the rationale behind this deprecation can be found in #7377 and #9024.

Former validators for contour.negative_linestyle

The former public validation functions validate_negative_linestyle and validate_negative_linestyle_legacy will be deprecated in 2.1 and may be removed in 2.3. There are no public functions to replace them.

cbook

Many unused or near-unused *matplotlib.cbook* functions and classes have been deprecated: `converter`, `tostr`, `todatetime`, `todate`, `tofloat`, `toint`, `unique`, `is_string_like`, `is_sequence_of_strings`, `is_scalar`, `Sorter`, `Xlator`, `soundex`, `Null`, `dict_delall`, `RingBuffer`, `get_split_ind`, `wrap`, `get_recursive_filelist`, `pieces`, `exception_to_str`, `allequal`, `alltrue`, `onetrue`, `allpairs`, `finddir`, `reverse_dict`, `restrict_dict`, `issubclass_safe`, `recursive_remove`, `unmasked_index_ranges`.

Code Removal

qt4_compat.py

Moved to `qt_compat.py`. Renamed because it now handles Qt5 as well.

Previously Deprecated methods

The `GraphicsContextBase.set_graylevel`, `FigureCanvasBase.onHilite` and `mpl_toolkits.axes_grid1.mpl_axes.Axes.toggle_axisline` methods have been removed.

The `ArtistInspector.findobj` method, which was never working due to the lack of a `get_children` method, has been removed.

The deprecated `point_in_path`, `get_path_extents`, `point_in_path_collection`, `path_intersects_path`, `convert_path_to_polygons`, `cleanup_path` and `clip_path_to_rect` functions in the `matplotlib.path` module have been removed. Their functionality remains exposed as methods on the `Path` class.

The deprecated `Artist.get_axes` and `Artist.set_axes` methods have been removed

The `matplotlib.backends.backend_ps.seq_allequal` function has been removed. Use `np.array_equal` instead.

The deprecated `matplotlib.rcsetup.validate_maskedarray`, `matplotlib.rcsetup.deprecate_savefig_extension` and `matplotlib.rcsetup.validate_tkpythoninspect` functions, and associated `savefig.extension` and `tk.pythoninspect` rcparams entries have been removed.

The keyword argument *resolution* of *matplotlib.projections.polar.PolarAxes* has been removed. It has deprecation with no effect from version *0.98.x*.

`Axes.set_aspect("normal")`

Support for setting an `Axes`'s aspect to `"normal"` has been removed, in favor of the synonym `"auto"`.

`shading` kwarg to `pcolor`

The `shading` kwarg to *`pcolor`* has been removed. Set `edgecolors` appropriately instead.

Functions removed from the `lines` module

The *`matplotlib.lines`* module no longer imports the `pts_to_prestep`, `pts_to_midstep` and `pts_to_poststep` functions from *`matplotlib.cbook`*.

PDF backend functions

The methods `embedTeXFont` and `tex_font_mapping` of *`matplotlib.backends.backend_pdf.PdfFile`* have been removed. It is unlikely that external users would have called these methods, which are related to the font system internal to the PDF backend.

matplotlib.delaunay

Remove the delaunay triangulation code which is now handled by Qhull via *`matplotlib.tri`*.

16.1.12 API Changes in 2.0.1

Extensions to `matplotlib.backend_bases.GraphicsContextBase`

To better support controlling the color of hatches, the method *`matplotlib.backend_bases.GraphicsContextBase.set_hatch_color`* was added to the expected API of `GraphicsContext` classes. Calls to this method are currently wrapped with a `try:...except Attribute:` block to preserve back-compatibility with any third-party backends which do not extend *`GraphicsContextBase`*.

This value can be accessed in the backends via *`matplotlib.backend_bases.GraphicsContextBase.get_hatch_color`* (which was added in 2.0 see *Extension to matplotlib.backend_bases.GraphicsContextBase*) and should be used to color the hatches.

In the future there may also be `hatch_linewidth` and `hatch_density` related methods added. It is encouraged, but not required that third-party backends extend *`GraphicsContextBase`* to make adapting to these changes easier.

`afm.get_fontconfig_fonts` returns a list of paths and does not check for existence

`afm.get_fontconfig_fonts` used to return a set of paths encoded as a `{key: 1, ...}` dict, and checked for the existence of the paths. It now returns a list and dropped the existence check, as the same check is performed by the caller (`afm.findSystemFonts`) as well.

`bar` now returns rectangles of negative height or width if the corresponding input is negative

`pyplot.bar` used to normalize the coordinates of the rectangles that it created, to keep their height and width positives, even if the corresponding input was negative. This normalization has been removed to permit a simpler computation of the correct *`Artist.sticky_edges`* to use.

Do not clip line width when scaling dashes

The algorithm to scale dashes was changed to no longer clip the scaling factor: the dash patterns now continue to shrink at thin line widths. If the line width is smaller than the effective pixel size, this may result in dashed lines turning into solid gray-ish lines. This also required slightly tweaking the default patterns for '--', ':', and '.-' so that with the default line width the final patterns would not change.

There is no way to restore the old behavior.

Deprecate 'Vega' color maps

The "Vega" colormaps are deprecated in Matplotlib 2.0.1 and will be removed in Matplotlib 2.2. Use the "tab" colormaps instead: "tab10", "tab20", "tab20b", "tab20c".

16.1.13 API Changes in 2.0.0

Deprecation and removal

Color of Axes

The `axisbg` and `axis_bgcolor` properties on *Axes* have been deprecated in favor of `facecolor`.

GTK and GDK backends deprecated

The GDK and GTK backends have been deprecated. These obsolete backends allow figures to be rendered via the GDK API to files and GTK2 figures. They are untested and known to be broken, and their use has been discouraged for some time. Instead, use the `GTKAgg` and `GTKCairo` backends for rendering to GTK2 windows.

WX backend deprecated

The WX backend has been deprecated. It is untested, and its use has been discouraged for some time. Instead, use the `WXAgg` backend for rendering figures to WX windows.

CocoaAgg backend removed

The deprecated and not fully functional CocoaAgg backend has been removed.

`round` removed from TkAgg Backend

The TkAgg backend had its own implementation of the `round` function. This was unused internally and has been removed. Instead, use either the `round` builtin function or `numpy.around`.

'hold' functionality deprecated

The 'hold' keyword argument and all functions and methods related to it are deprecated, along with the `axes.hold` rcParams entry. The behavior will remain consistent with the default `hold=True` state that has long been in place. Instead of using a function or keyword argument (`hold=False`) to change that behavior, explicitly clear the axes or figure as needed prior to subsequent plotting commands.

`Artist.update` has return value

The methods *matplotlib.artist.Artist.set*, *matplotlib.artist.Artist.update*, and the function *matplotlib.artist.setp* now use a common codepath to look up how to update the given artist properties (either using the setter methods or an attribute/property).

The behavior of *matplotlib.artist.Artist.update* is slightly changed to return a list of the values returned from the setter methods to avoid changing the API of *matplotlib.artist.Artist.set* and *matplotlib.artist.setp*.

The keys passed into *matplotlib.artist.Artist.update* are now converted to lower case before being processed, to match the behavior of *matplotlib.artist.Artist.set*

and *matplotlib.artist.setp*. This should not break any user code because there are no set methods with capitals in their names, but this puts a constraint on naming properties in the future.

Legend initializers gain *edgecolor* and *facecolor* keyword arguments

The *Legend* background patch (or 'frame') can have its `edgecolor` and `facecolor` determined by the corresponding keyword arguments to the *matplotlib.legend.Legend* initializer, or to any of the methods or functions that call that initializer. If left to their default values of `None`, their values will be taken from `matplotlib.rcParams`. The previously-existing `framealpha` kwarg still controls the alpha transparency of the patch.

Qualitative colormaps

Colorbrewer's qualitative/discrete colormaps ("Accent", "Dark2", "Paired", "Pastel1", "Pastel2", "Set1", "Set2", "Set3") are now implemented as *ListedColormap* instead of *LinearSegmentedColormap*.

To use these for images where categories are specified as integers, for instance, use:

```
plt.imshow(x, cmap='Dark2', norm=colors.NoNorm())
```

Change in the `draw_image` backend API

The `draw_image` method implemented by backends has changed its interface.

This change is only relevant if the backend declares that it is able to transform images by returning `True` from `option_scale_image`. See the `draw_image` docstring for more information.

`matplotlib.ticker.LinearLocator` algorithm update

The *matplotlib.ticker.LinearLocator* is used to define the range and location of axis ticks when the user wants an exact number of ticks. `LinearLocator` thus differs from the default locator `MaxNLocator`, for which the user specifies a maximum number of intervals rather than a precise number of ticks.

The view range algorithm in `matplotlib.ticker.LinearLocator` has been changed so that more convenient tick locations are chosen. The new algorithm returns a plot view range that is a multiple of the user-requested number of ticks. This ensures tick marks will be located at whole integers more consistently. For example, when both y-axes of a``twinx`` plot use `matplotlib.ticker.LinearLocator` with the same number of ticks, their y-tick locations and grid lines will coincide.

`matplotlib.ticker.LogLocator` gains numticks kwarg

The maximum number of ticks generated by the *`LogLocator`* can now be controlled explicitly via setting the new 'numticks' kwarg to an integer. By default the kwarg is None which internally sets it to the 'auto' string, triggering a new algorithm for adjusting the maximum according to the axis length relative to the ticklabel font size.

`matplotlib.ticker.LogFormatter`: two new kwargs

Previously, minor ticks on log-scaled axes were not labeled by default. An algorithm has been added to the *`LogFormatter`* to control the labeling of ticks between integer powers of the base. The algorithm uses two parameters supplied in a kwarg tuple named 'minor_thresholds'. See the docstring for further explanation.

To improve support for axes using *`SymmetricalLogLocator`*, a *linthresh* keyword argument was added.

New defaults for 3D quiver function in mpl_toolkits.mplot3d.axes3d.py

Matplotlib has both a 2D and a 3D `quiver` function. These changes affect only the 3D function and make the default behavior of the 3D function match the 2D version. There are two changes:

1) The 3D quiver function previously normalized the arrows to be the same length, which makes it unusable for situations where the arrows should be different lengths and does not match the behavior of the 2D function. This normalization behavior is now controlled with the `normalize` keyword, which defaults to False.
2) The `pivot` keyword now defaults to `tail` instead of `tip`. This was done in order to match the default behavior of the 2D quiver function.

To obtain the previous behavior with the 3D quiver function, one can call the function with

```
ax.quiver(x, y, z, u, v, w, normalize=True, pivot='tip')
```

where "ax" is an `Axes3d` object created with something like

```
import mpl_toolkits.mplot3d.axes3d
ax = plt.subplot(111, projection='3d')
```

Stale figure behavior

Attempting to draw the figure will now mark it as not stale (independent if the draw succeeds). This change is to prevent repeatedly trying to re-draw a figure which is raising an error on draw. The previous behavior would only mark a figure as not stale after a full re-draw succeeded.

The spectral colormap is now nipy_spectral

The colormaps formerly known as `spectral` and `spectral_r` have been replaced by `nipy_spectral` and `nipy_spectral_r` since Matplotlib 1.3.0. Even though the colormap was deprecated in Matplotlib 1.3.0, it never raised a warning. As of Matplotlib 2.0.0, using the old names raises a deprecation warning. In the future, using the old names will raise an error.

Default install no longer includes test images

To reduce the size of wheels and source installs, the tests and baseline images are no longer included by default.

To restore installing the tests and images, use a `setup.cfg` with

```
[packages]
tests = True
toolkits_tests = True
```

in the source directory at build/install time.

16.1.14 Changes in 1.5.3

`ax.plot(..., marker=None)` gives default marker

Prior to 1.5.3 keyword arguments passed to *`plot`* were handled in two parts -- default keyword arguments generated internal to *`plot`* (such as the cycled styles) and user supplied keyword arguments. The internally generated keyword arguments were passed to the *`matplotlib.lines.Line2D`* and the user keyword arguments were passed to `ln.set(**kwargs)` to update the artist after it was created. Now both sets of keyword arguments are merged and passed to *`Line2D`*. This change was made to allow *None* to be passed in via the user keyword arguments to mean 'do the default thing' as is the convention through out Matplotlib rather than raising an exception.

Unlike most *`Line2D`* setter methods *`set_marker`* did accept `None` as a valid input which was mapped to 'no marker'. Thus, by routing this `marker=None` through `__init__` rather than `set(...)` the meaning of `ax.plot(..., marker=None)` changed from 'no markers' to 'default markers from rcparams'.

This is change is only evident if `mpl.rcParams['lines.marker']` has a value other than `'None'` (which is string `'None'` which means 'no marker').

16.1.15 Changes in 1.5.2

Default Behavior Changes

Changed default `autorange` behavior in boxplots

Prior to v1.5.2, the whiskers of boxplots would extend to the minimum and maximum values if the quartiles were all equal (i.e., Q1 = median = Q3). This behavior has been disabled by default to restore consistency with other plotting packages.

To restore the old behavior, simply set `autorange=True` when calling `plt.boxplot`.

16.1.16 Changes in 1.5.0

Code Changes

Reversed `matplotlib.cbook.ls_mapper`, added `ls_mapper_r`

Formerly, `matplotlib.cbook.ls_mapper` was a dictionary with the long-form line-style names (`"solid"`) as keys and the short forms (`"-"`) as values. This long-to-short mapping is now done by `ls_mapper_r`, and the short-to-long mapping is done by the `ls_mapper`.

Prevent moving artists between Axes, Property-ify Artist.axes, deprecate Artist.{get,set}_axes

This was done to prevent an Artist that is already associated with an Axes from being moved/added to a different Axes. This was never supported as it causes havoc with the transform stack. The apparent support for this (as it did not raise an exception) was the source of multiple bug reports and questions on SO.

For almost all use-cases, the assignment of the axes to an artist should be taken care of by the axes as part of the `Axes.add_*` method, hence the deprecation of {get,set}_axes.

Removing the `set_axes` method will also remove the 'axes' line from the ACCEPTS kwarg tables (assuming that the removal date gets here before that gets overhauled).

Tightened input validation on 'pivot' kwarg to quiver

Tightened validation so that only {'tip', 'tail', 'mid', and 'middle'} (but any capitalization) are valid values for the *pivot* keyword argument in the *Quiver* class (and hence *axes.Axes.quiver* and *pyplot.quiver* which both fully delegate to *Quiver*). Previously any input matching 'mid.*' would be interpreted as 'middle', 'tip.*' as 'tip' and any string not matching one of those patterns as 'tail'.

The value of `Quiver.pivot` is normalized to be in the set {'tip', 'tail', 'middle'} in *Quiver*.

Reordered `Axes.get_children`

The artist order returned by *axes.Axes.get_children* did not match the one used by *axes.Axes.draw*. They now use the same order, as *axes.Axes.draw* now calls *axes.Axes.get_children*.

Changed behaviour of contour plots

The default behaviour of *contour()* and *contourf()* when using a masked array is now determined by the new keyword argument *corner_mask*, or if this is not specified then the new `rcParams["contour.corner_mask"]` (default: `True`) instead. The new default behaviour is equivalent to using `corner_mask=True`; the previous behaviour can be obtained using `corner_mask=False` or by changing the rcParam. The example http://matplotlib.org/examples/pylab_examples/contour_corner_mask.html demonstrates the difference. Use of the old contouring algorithm, which is obtained with `corner_mask='legacy'`, is now deprecated.

Contour labels may now appear in different places than in earlier versions of Matplotlib.

In addition, the keyword argument *nchunk* now applies to *contour()* as well as *contourf()*, and it subdivides the domain into subdomains of exactly *nchunk* by *nchunk* quads, whereas previously it was only roughly *nchunk* by *nchunk* quads.

The C/C++ object that performs contour calculations used to be stored in the public attribute `QuadContourSet.Cntr`, but is now stored in a private attribute and should not be accessed by end users.

Added set_params function to all Locator types

This was a bug fix targeted at making the api for Locators more consistent.

In the old behavior, only locators of type MaxNLocator have set_params() defined, causing its use on any other Locator to raise an AttributeError *(aside: set_params(args) is a function that sets the parameters of a Locator instance to be as specified within args)*. The fix involves moving set_params() to the Locator class such that all subtypes will have this function defined.

Since each of the Locator subtypes have their own modifiable parameters, a universal set_params() in Locator isn't ideal. Instead, a default no-operation function that raises a warning is implemented in Locator. Subtypes extending Locator will then override with their own implementations. Subtypes that do not have a need for set_params() will fall back onto their parent's implementation, which raises a warning as intended.

In the new behavior, Locator instances will not raise an AttributeError when set_params() is called. For Locators that do not implement set_params(), the default implementation in Locator is used.

Disallow `None` as x or y value in ax.plot

Do not allow `None` as a valid input for the `x` or `y` args in *`axes.Axes.plot`*. This may break some user code, but this was never officially supported (ex documented) and allowing `None` objects through can lead to confusing exceptions downstream.

To create an empty line use

```
ln1, = ax.plot([], [], ...)
ln2, = ax.plot([], ...)
```

In either case to update the data in the *`Line2D`* object you must update both the `x` and `y` data.

Removed *args* and *kwargs* from `MicrosecondLocator.__call__`

The call signature of `__call__()` has changed from `__call__(self, *args, **kwargs)` to `__call__(self)`. This is consistent with the superclass *`Locator`* and also all the other Locators derived from this superclass.

No `ValueError` for the MicrosecondLocator and YearLocator

The *MicrosecondLocator* and *YearLocator* objects when called will return an empty list if the axes have no data or the view has no interval. Previously, they raised a `ValueError`. This is consistent with all the Date Locators.

'OffsetBox.DrawingArea' respects the 'clip' keyword argument

The call signature was `OffsetBox.DrawingArea(..., clip=True)` but nothing was done with the *clip* argument. The object did not do any clipping regardless of that parameter. Now the object can and does clip the child *Artist*s if they are set to be clipped.

You can turn off the clipping on a per-child basis using `child.set_clip_on(False)`.

Add salt to clipPath id

Add salt to the hash used to determine the id of the `clipPath` nodes. This is to avoid conflicts when two svg documents with the same clip path are included in the same document (see https://github.com/ipython/ipython/issues/8133 and https://github.com/matplotlib/matplotlib/issues/4349), however this means that the svg output is no longer deterministic if the same figure is saved twice. It is not expected that this will affect any users as the current ids are generated from an md5 hash of properties of the clip path and any user would have a very difficult time anticipating the value of the id.

Changed snap threshold for circle markers to inf

When drawing circle markers above some marker size (previously 6.0) the path used to generate the marker was snapped to pixel centers. However, this ends up distorting the marker away from a circle. By setting the snap threshold to inf snapping is never done on circles.

This change broke several tests, but is an improvement.

Preserve units with Text position

Previously the 'get_position' method on Text would strip away unit information even though the units were still present. There was no inherent need to do this, so it has been changed so that unit data (if present) will be preserved. Essentially a call to 'get_position' will return the exact value from a call to 'set_position'.

If you wish to get the old behaviour, then you can use the new method called 'get_unitless_position'.

New API for custom Axes view changes

Interactive pan and zoom were previously implemented using a Cartesian-specific algorithm that was not necessarily applicable to custom Axes. Three new private methods, `matplotlib.axes._base._AxesBase._get_view`, `matplotlib.axes._base._AxesBase._set_view`, and `matplotlib.axes._base._AxesBase._set_view_from_bbox`, allow for custom *Axes* classes to override the pan and zoom algorithms. Implementors of custom *Axes* who override these methods may provide suitable behaviour for both pan and zoom as well as the view navigation buttons on the interactive toolbars.

MathTex visual changes

The spacing commands in mathtext have been changed to more closely match vanilla TeX.

Improved spacing in mathtext

The extra space that appeared after subscripts and superscripts has been removed.

No annotation coordinates wrap

In #2351 for 1.4.0 the behavior of ['axes points', 'axes pixel', 'figure points', 'figure pixel'] as coordinates was change to no longer wrap for negative values. In 1.4.3 this change was reverted for 'axes points' and 'axes pixel' and in addition caused 'axes fraction' to wrap. For 1.5 the behavior has been reverted to as it was in 1.4.0-1.4.2, no wrapping for any type of coordinate.

Deprecation

Deprecated `GraphicsContextBase.set_graylevel`

The `GraphicsContextBase.set_graylevel` function has been deprecated in 1.5 and will be removed in 1.6. It has been unused. The *`GraphicsContextBase.set_foreground`* could be used instead.

deprecated idle_event

The `idle_event` was broken or missing in most backends and causes spurious warnings in some cases, and its use in creating animations is now obsolete due to the animations module. Therefore code involving it has been removed from all but the wx backend (where it partially works), and its use is deprecated. The *animation* module may be used instead to create animations.

`color_cycle` deprecated

In light of the new property cycling feature, the Axes method `set_color_cycle` is now deprecated. Calling this method will replace the current property cycle with one that cycles just the given colors.

Similarly, the rc parameter *axes.color_cycle* is also deprecated in lieu of the new `rcParams["axes.prop_cycle"]` (default: `cycler('color', ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728', '#9467bd', '#8c564b', '#e377c2', '#7f7f7f', '#bcbd22', '#17becf'])`) parameter. Having both parameters in the same rc file is not recommended as the result cannot be predicted. For compatibility, setting *axes.color_cycle* will replace the cycler in `rcParams["axes.prop_cycle"]` (default: `cycler('color', ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728', '#9467bd', '#8c564b', '#e377c2', '#7f7f7f', '#bcbd22', '#17becf'])`) with a color cycle. Accessing *axes.color_cycle* will return just the color portion of the property cycle, if it exists.

Timeline for removal has not been set.

Bundled jquery

The version of jquery bundled with the webagg backend has been upgraded from 1.7.1 to 1.11.3. If you are using the version of jquery bundled with webagg you will need to update your html files as such

```
-    <script src="_static/jquery/js/jquery-1.7.1.min.js"></script>
+    <script src="_static/jquery/js/jquery-1.11.3.min.js"></script>
```

Code Removed

Removed `Image` from main namespace

`Image` was imported from PIL/pillow to test if PIL is available, but there is no reason to keep `Image` in the namespace once the availability has been determined.

Removed `lod` from Artist

Removed the method `set_lod` and all references to the attribute `_lod` as they are not used anywhere else in the code base. It appears to be a feature stub that was never built out.

Removed threading related classes from cbook

The classes `Scheduler`, `Timeout`, and `Idle` were in cbook, but are not used internally. They appear to be a prototype for the idle event system which was not working and has recently been pulled out.

Removed *Lena* images from sample_data

The `lena.png` and `lena.jpg` images have been removed from Matplotlib's sample_data directory. The images are also no longer available from *matplotlib.cbook.get_sample_data*. We suggest using `matplotlib.cbook.get_sample_data('grace_hopper.png')` or `matplotlib.cbook.get_sample_data('grace_hopper.jpg')` instead.

Legend

Removed handling of *loc* as a positional argument to *Legend*

Legend handlers

Remove code to allow legend handlers to be callable. They must now implement a method `legend_artist`.

Axis

Removed method `set_scale`. This is now handled via a private method which should not be used directly by users. It is called via `Axes.set_{x,y}scale` which takes care of ensuring the related changes are also made to the Axes object.

finance.py

Removed functions with ambiguous argument order from finance.py

Annotation

Removed `textcoords` and `xytext` proprieties from Annotation objects.

sphinxext.ipython_*.py

Both `ipython_console_highlighting` and `ipython_directive` have been moved to IPython.

Change your import from `matplotlib.sphinxext.ipython_directive` to `IPython.sphinxext.ipython_directive` and from `matplotlib.sphinxext.ipython_directive` to `IPython.sphinxext.ipython_directive`

LineCollection.color

Deprecated in 2005, use `set_color`

remove `'faceted'` as a valid value for *shading* in `tri.tripcolor`

Use *edgecolor* instead. Added validation on *shading* to only be valid values.

Remove `faceted` kwarg from scatter

Remove support for the `faceted` kwarg. This was deprecated in d48b34288e9651ff95c3b8a071ef5ac5cf50bae7 (2008-04-18!) and replaced by `edgecolor`.

Remove `set_colorbar` method from `ScalarMappable`

Remove `set_colorbar` method, use *`colorbar`* attribute directly.

patheffects.svg

- remove `get_proxy_renderer` method from `AbstarctPathEffect` class
- remove `patch_alpha` and `offset_xy` from `SimplePatchShadow`

Remove `testing.image_util.py`

Contained only a no-longer used port of functionality from PIL

Remove `mlab.FIFOBuffer`

Not used internally and not part of core mission of mpl.

Remove `mlab.prepca`

Deprecated in 2009.

Remove `NavigationToolbar2QTAgg`

Added no functionality over the base `NavigationToolbar2Qt`

mpl.py

Remove the module `matplotlib.mpl`. Deprecated in 1.3 by PR #1670 and commit 78ce67d161625833cacff23cfe5d74920248c5b2

16.1.17 Changes in 1.4.x

Code changes

- A major refactoring of the axes module was made. The axes module has been split into smaller modules:
  - the `_base` module, which contains a new private `_AxesBase` class. This class contains all methods except plotting and labelling methods.
  - the *`axes`* module, which contains the *`axes.Axes`* class. This class inherits from `_AxesBase`, and contains all plotting and labelling methods.
  - the `_subplot` module, with all the classes concerning subplotting.

There are a couple of things that do not exists in the *axes* module's namespace anymore. If you use them, you need to import them from their original location:

- `math -> import math`
- `ma -> from numpy import ma`
- `cbook -> from matplotlib import cbook`
- `docstring -> from matplotlib import docstring`
- `is_sequence_of_strings -> from matplotlib.cbook import is_sequence_of_strings`
- `is_string_like -> from matplotlib.cbook import is_string_like`
- `iterable -> from matplotlib.cbook import iterable`
- `itertools -> import itertools`
- `martist -> from matplotlib import artist as martist`
- `matplotlib -> import matplotlib`
- `mcoll -> from matplotlib import collections as mcoll`
- `mcolors -> from matplotlib import colors as mcolors`
- `mcontour -> from matplotlib import contour as mcontour`
- `mpatches -> from matplotlib import patches as mpatches`
- `mpath -> from matplotlib import path as mpath`
- `mquiver -> from matplotlib import quiver as mquiver`
- `mstack -> from matplotlib import stack as mstack`
- `mstream -> from matplotlib import stream as mstream`
- `mtable -> from matplotlib import table as mtable`
- As part of the refactoring to enable Qt5 support, the module `matplotlib.backends.qt4_compat` was renamed to `matplotlib.backends.qt_compat`. `qt4_compat` is deprecated in 1.4 and will be removed in 1.5.
- The *errorbar()* method has been changed such that the upper and lower limits (*lolims*, *uplims*, *xlolims*, *xuplims*) now point in the correct direction.
- The *fmt* kwarg for *errorbar()* now supports the string 'none' to suppress drawing of a line and markers; use of the *None* object for this is deprecated. The default *fmt* value is changed to the empty string (''), so the line and markers are governed by the *plot()* defaults.
- A bug has been fixed in the path effects rendering of fonts, which now means that the font size is consistent with non-path effect fonts. See https://github.com/matplotlib/matplotlib/issues/2889 for more detail.
- The Sphinx extensions `ipython_directive` and `ipython_console_highlighting` have been moved to the IPython project itself. While they remain in Matplotlib for this release, they have been deprecated. Update your extensions in `conf.py`

to point to `IPython.sphinxext.ipython_directive` instead of `matplotlib.sphinxext.ipython_directive`.

- In `matplotlib.finance`, almost all functions have been deprecated and replaced with a pair of functions name `*_ochl` and `*_ohlc`. The former is the 'open-close-high-low' order of quotes used previously in this module, and the latter is the 'open-high-low-close' order that is standard in finance.
- For consistency the `face_alpha` keyword to *matplotlib.patheffects.SimplePatchShadow* has been deprecated in favour of the `alpha` keyword. Similarly, the keyword `offset_xy` is now named `offset` across all *AbstractPathEffect*s. `matplotlib.patheffects._Base` has been renamed to *matplotlib.patheffects.AbstractPathEffect*. `matplotlib.patheffect.ProxyRenderer` has been renamed to *matplotlib.patheffects.PathEffectRenderer* and is now a full RendererBase subclass.
- The artist used to draw the outline of a *Figure.colorbar* has been changed from a *matplotlib.lines.Line2D* to *matplotlib.patches.Polygon*, thus `colorbar.ColorbarBase.outline` is now a *matplotlib.patches.Polygon* object.
- The legend handler interface has changed from a callable, to any object which implements the `legend_artists` method (a deprecation phase will see this interface be maintained for v1.4). See *Legend guide* for further details. Further legend changes include:
  - `matplotlib.axes.Axes._get_legend_handles` now returns a generator of handles, rather than a list.
  - The *legend()* function's *loc* positional argument has been deprecated. Use the *loc* keyword argument instead.
- The `rcParams["savefig.transparent"]` (default: `False`) has been added to control default transparency when saving figures.
- Slightly refactored the *Annotation* family. The text location in *Annotation* is now entirely handled by the underlying *Text* object so `.set_position` works as expected. The attributes *xytext* and *textcoords* have been deprecated in favor of *xyann* and *anncoords* so that *Annotation* and *AnnotationBbox* can share a common sensibly named api for getting/setting the location of the text or box.
  - *xyann* -> set the location of the annotation
  - *xy* -> set where the arrow points to
  - *anncoords* -> set the units of the annotation location
  - *xycoords* -> set the units of the point location
  - `set_position()` -> *Annotation* only set location of annotation
- *matplotlib.mlab.specgram*, *matplotlib.mlab.psd*, *matplotlib.mlab.csd*, *matplotlib.mlab.cohere*, `matplotlib.mlab.cohere_pairs`, *matplotlib.pyplot.specgram*, *matplotlib.pyplot.psd*, *matplotlib.pyplot.csd*, and *matplotlib.pyplot.cohere* now raise ValueError where they previously raised AssertionError.

- For *matplotlib.mlab.psd*, *matplotlib.mlab.csd*, *matplotlib.mlab.cohere*, matplotlib.mlab.cohere_pairs, *matplotlib.pyplot.specgram*, *matplotlib.pyplot.psd*, *matplotlib.pyplot.csd*, and *matplotlib.pyplot.cohere*, in cases where a shape (n, 1) array is returned, this is now converted to a (n,) array. Previously, (n, m) arrays were averaged to an (n,) array, but (n, 1) arrays were returned unchanged. This change makes the dimensions consistent in both cases.
- Added the rcParams["axes.formatter.useoffset"] (default: True) to control the default value of *useOffset* in *ticker.ScalarFormatter*
- Added *Formatter* sub-class *StrMethodFormatter* which does the exact same thing as *FormatStrFormatter*, but for new-style formatting strings.
- Deprecated matplotlib.testing.image_util and the only function within, matplotlib.testing.image_util.autocontrast. These will be removed completely in v1.5.0.
- The fmt argument of *plot_date()* has been changed from bo to just o, so color cycling can happen by default.
- Removed the class FigureManagerQTAgg and deprecated NavigationToolbar2QTAgg which will be removed in 1.5.
- Removed formerly public (non-prefixed) attributes rect and drawRect from FigureCanvasQTAgg; they were always an implementation detail of the (preserved) drawRectangle() function.
- The function signatures of tight_bbox.adjust_bbox and tight_bbox.process_figure_for_rasterizing have been changed. A new *fixed_dpi* parameter allows for overriding the figure.dpi setting instead of trying to deduce the intended behaviour from the file format.
- Added support for horizontal/vertical axes padding to *mpl_toolkits.axes_grid1.axes_grid.ImageGrid* --- argument *axes_pad* can now be tuple-like if separate axis padding is required. The original behavior is preserved.
- Added support for skewed transforms to *matplotlib.transforms.Affine2D*, which can be created using the *skew* and *skew_deg* methods.
- Added clockwise parameter to control sectors direction in *axes.Axes.pie*
- In *matplotlib.lines.Line2D* the *markevery* functionality has been extended. Previously an integer start-index and stride-length could be specified using either a two-element-list or a two-element-tuple. Now this can only be done using a two-element-tuple. If a two-element-list is used then it will be treated as NumPy fancy indexing and only the two markers corresponding to the given indexes will be shown.
- Removed *prop* keyword argument from *mpl_toolkits.axes_grid1.anchored_artists.AnchoredSizeBar* call. It was passed through to the base-class __init__ and is only used for setting padding. Now *fontproperties* (which is what is really used to set the font properties of *AnchoredSizeBar*) is passed through in place of *prop*. If *fontproperties* is not passed in, but *prop* is, then *prop* is used in place of *fontproperties*. If both are passed in, *prop* is silently ignored.

- The use of the index 0 in *pyplot.subplot* and related commands is deprecated. Due to a lack of validation, calling `plt.subplots(2, 2, 0)` does not raise an exception, but puts an axes in the _last_ position. This is due to the indexing in subplot being 1-based (to mirror MATLAB) so before indexing into the *GridSpec* object used to determine where the axes should go, 1 is subtracted off. Passing in 0 results in passing -1 to *GridSpec* which results in getting the last position back. Even though this behavior is clearly wrong and not intended, we are going through a deprecation cycle in an abundance of caution that any users are exploiting this 'feature'. The use of 0 as an index will raise a warning in 1.4 and an exception in 1.5.
- Clipping is now off by default on offset boxes.
- Matplotlib now uses a less-aggressive call to `gc.collect(1)` when closing figures to avoid major delays with large numbers of user objects in memory.
- The default clip value of *all* pie artists now defaults to `False`.

Code removal

- Removed `mlab.levypdf`. The code raised a NumPy error (and has for a long time) and was not the standard form of the Levy distribution. `scipy.stats.levy` should be used instead

16.1.18 Changes in 1.3.x

Changes in 1.3.1

It is rare that we make an API change in a bugfix release, however, for 1.3.1 since 1.3.0 the following change was made:

- `text.Text.cached` (used to cache font objects) has been made into a private variable. Among the obvious encapsulation benefit, this removes this confusing-looking member from the documentation.
- The method *hist()* now always returns bin occupancies as an array of type `float`. Previously, it was sometimes an array of type `int`, depending on the call.

Code removal

- The following items that were deprecated in version 1.2 or earlier have now been removed completely.
  - The Qt 3.x backends (`qt` and `qtagg`) have been removed in favor of the Qt 4.x backends (`qt4` and `qt4agg`).
  - The FltkAgg and Emf backends have been removed.

- The `matplotlib.nxutils` module has been removed. Use the functionality on *matplotlib.path.Path.contains_point* and friends instead.
- Instead of `axes.Axes.get_frame`, use `axes.Axes.patch`.
- The following keyword arguments to the *legend* function have been renamed:
  * *pad* -> *borderpad*
  * *labelsep* -> *labelspacing*
  * *handlelen* -> *handlelength*
  * *handletextsep* -> *handletextpad*
  * *axespad* -> *borderaxespad*

  Related to this, the following rcParams have been removed:
  * `legend.pad`,
  * `legend.labelsep`,
  * `legend.handlelen`,
  * `legend.handletextsep` and
  * `legend.axespad`
- For the *hist* function, instead of *width*, use *rwidth* (relative width).
- On *patches.Circle*, the *resolution* keyword argument has been removed. For a circle made up of line segments, use *patches.CirclePolygon*.
- The printing functions in the Wx backend have been removed due to the burden of keeping them up-to-date.
- `mlab.liaupunov` has been removed.
- `mlab.save`, `mlab.load`, `pylab.save` and `pylab.load` have been removed. We recommend using `numpy.savetxt` and `numpy.loadtxt` instead.
- `widgets.HorizontalSpanSelector` has been removed. Use *widgets.SpanSelector* instead.

Code deprecation

- The CocoaAgg backend has been deprecated, with the possibility for deletion or resurrection in a future release.
- The top-level functions in *matplotlib.path* that are implemented in C++ were never meant to be public. Instead, users should use the Pythonic wrappers for them in the *path.Path* and *collections.Collection* classes. Use the following mapping to update your code:
  - `point_in_path` -> *path.Path.contains_point*
  - `get_path_extents` -> *path.Path.get_extents*

  - point_in_path_collection -> *collections.Collection.contains*
  - path_in_path -> *path.Path.contains_path*
  - path_intersects_path -> *path.Path.intersects_path*
  - convert_path_to_polygons -> *path.Path.to_polygons*
  - cleanup_path -> *path.Path.cleaned*
  - points_in_path -> *path.Path.contains_points*
  - clip_path_to_rect -> *path.Path.clip_to_bbox*

- `matplotlib.colors.normalize` and `matplotlib.colors.no_norm` have been deprecated in favour of *matplotlib.colors.Normalize* and *matplotlib.colors.NoNorm* respectively.
- The *ScalarMappable* class' `set_colorbar` method is now deprecated. Instead, the *matplotlib.cm.ScalarMappable.colorbar* attribute should be used. In previous Matplotlib versions this attribute was an undocumented tuple of `(colorbar_instance, colorbar_axes)` but is now just `colorbar_instance`. To get the colorbar axes it is possible to just use the `ax` attribute on a colorbar instance.
- The `matplotlib.mpl` module is now deprecated. Those who relied on this module should transition to simply using `import matplotlib as mpl`.

Code changes

- *Patch* now fully supports using RGBA values for its `facecolor` and `edgecolor` attributes, which enables faces and edges to have different alpha values. If the *Patch* object's `alpha` attribute is set to anything other than `None`, that value will override any alpha-channel value in both the face and edge colors. Previously, if *Patch* had `alpha=None`, the alpha component of `edgecolor` would be applied to both the edge and face.
- The optional `isRGB` argument to *set_foreground()* (and the other GraphicsContext classes that descend from it) has been renamed to `isRGBA`, and should now only be set to `True` if the `fg` color argument is known to be an RGBA tuple.
- For *Patch*, the `capstyle` used is now `butt`, to be consistent with the default for most other objects, and to avoid problems with non-solid `linestyle` appearing solid when using a large `linewidth`. Previously, *Patch* used `capstyle='projecting'`.
- *Path* objects can now be marked as *readonly* by passing `readonly=True` to its constructor. The built-in path singletons, obtained through `Path.unit*` class methods return readonly paths. If you have code that modified these, you will need to make a deepcopy first, using either:

```
import copy
path = copy.deepcopy(Path.unit_circle())
```

(continues on next page)

(continued from previous page)

```
# or

path = Path.unit_circle().deepcopy()
```

Deep copying a *Path* always creates an editable (i.e. non-readonly) *Path*.

- The list at `Path.NUM_VERTICES` was replaced by a dictionary mapping Path codes to the number of expected vertices at *NUM_VERTICES_FOR_CODE*.
- To support XKCD style plots, the `matplotlib.path.cleanup_path` method's signature was updated to require a sketch argument. Users of `matplotlib.path.cleanup_path` are encouraged to use the new *cleaned()* Path method.
- Data limits on a plot now start from a state of having "null" limits, rather than limits in the range (0, 1). This has an effect on artists that only control limits in one direction, such as *axes.Axes.axvline* and *axes.Axes.axhline*, since their limits will no longer also include the range (0, 1). This fixes some problems where the computed limits would be dependent on the order in which artists were added to the axes.
- Fixed a bug in setting the position for the right/top spine with data position type. Previously, it would draw the right or top spine at +1 data offset.
- In *FancyArrow*, the default arrow head width, `head_width`, has been made larger to produce a visible arrow head. The new value of this kwarg is `head_width = 20 * width`.
- It is now possible to provide `number of levels + 1` colors in the case of `extend='both'` for contourf (or just `number of levels` colors for an extend value `min` or `max`) such that the resulting colormap's `set_under` and `set_over` are defined appropriately. Any other number of colors will continue to behave as before (if more colors are provided than levels, the colors will be unused). A similar change has been applied to contour, where `extend='both'` would expect `number of levels + 2` colors.
- A new keyword *extendrect* in *colorbar()* and *ColorbarBase* allows one to control the shape of colorbar extensions.
- The extension of *MultiCursor* to both vertical (default) and/or horizontal cursor implied that `self.line` is replaced by `self.vline` for vertical cursors lines and `self.hline` is added for the horizontal cursors lines.
- On POSIX platforms, the *report_memory()* function raises `NotImplementedError` instead of `OSError` if the `ps` command cannot be run.
- The `matplotlib.cbook.check_output` function has been moved to `matplotlib.compat.subprocess`.

Configuration and rcParams

- On Linux, the user-specific `matplotlibrc` configuration file is now located in `~/.config/matplotlib/matplotlibrc` to conform to the XDG Base Directory Specification.
- The `font.*` rcParams now affect only text objects created after the rcParam has been set, and will not retroactively affect already existing text objects. This brings their behavior in line with most other rcParams.
- Removed call of *grid()* in `matplotlib.pyplot.plotfile`. To draw the axes grid, set the `axes.grid` rcParam to *True*, or explicitly call *grid()*.

16.1.19 Changes in 1.2.x

- The `classic` option of the rc parameter `toolbar` is deprecated and will be removed in the next release.
- The `matplotlib.cbook.isvector` method has been removed since it is no longer functional.
- The `rasterization_zorder` property on *Axes* sets a zorder below which artists are rasterized. This has defaulted to -30000.0, but it now defaults to *None*, meaning no artists will be rasterized. In order to rasterize artists below a given zorder value, *set_rasterization_zorder* must be explicitly called.
- In *scatter()*, and *scatter*, when specifying a marker using a tuple, the angle is now specified in degrees, not radians.
- Using *twinx()* or *twiny()* no longer overrides the current locaters and formatters on the axes.
- In *contourf()*, the handling of the *extend* kwarg has changed. Formerly, the extended ranges were mapped after to 0, 1 after being normed, so that they always corresponded to the extreme values of the colormap. Now they are mapped outside this range so that they correspond to the special colormap values determined by the *set_under()* and *set_over()* methods, which default to the colormap end points.
- The new rc parameter `savefig.format` replaces `cairo.format` and `savefig.extension`, and sets the default file format used by *matplotlib.figure.Figure.savefig()*.
- In *pyplot.pie()* and *axes.Axes.pie()*, one can now set the radius of the pie; setting the *radius* to 'None' (the default value), will result in a pie with a radius of 1 as before.
- Use of `matplotlib.projections.projection_factory` is now deprecated in favour of axes class identification using `matplotlib.projections.process_projection_requirements` followed by direct axes class invocation

(at the time of writing, functions which do this are: *add_axes()*, *add_subplot()* and *gca()*). Therefore:

```
key = figure._make_key(*args, **kwargs)
ispolar = kwargs.pop('polar', False)
projection = kwargs.pop('projection', None)
if ispolar:
    if projection is not None and projection != 'polar':
        raise ValueError('polar and projection args are inconsistent')
    projection = 'polar'
ax = projection_factory(projection, self, rect, **kwargs)
key = self._make_key(*args, **kwargs)

# is now

projection_class, kwargs, key = \
                  process_projection_requirements(self, *args, **kwargs)
ax = projection_class(self, rect, **kwargs)
```

This change means that third party objects can expose themselves as Matplotlib axes by providing a `_as_mpl_axes` method. See *Developer's guide for creating scales and transformations* for more detail.

- A new keyword *extendfrac* in *colorbar()* and *ColorbarBase* allows one to control the size of the triangular minimum and maximum extensions on colorbars.
- A new keyword *capthick* in *errorbar()* has been added as an intuitive alias to the *markeredgewidth* and *mew* keyword arguments, which indirectly controlled the thickness of the caps on the errorbars. For backwards compatibility, specifying either of the original keyword arguments will override any value provided by *capthick*.
- Transform subclassing behaviour is now subtly changed. If your transform implements a non-affine transformation, then it should override the `transform_non_affine` method, rather than the generic `transform` method. Previously transforms would define `transform` and then copy the method into `transform_non_affine`:

```
class MyTransform(mtrans.Transform):
    def transform(self, xy):
        ...
    transform_non_affine = transform
```

This approach will no longer function correctly and should be changed to:

```
class MyTransform(mtrans.Transform):
    def transform_non_affine(self, xy):
        ...
```

- Artists no longer have `x_isdata` or `y_isdata` attributes; instead any artist's transform can be interrogated with `artist_instance.get_transform().contains_branch(ax.transData)`

- Lines added to an axes now take into account their transform when updating the data and view limits. This means transforms can now be used as a pre-transform. For instance:

```
>>> import matplotlib.pyplot as plt
>>> import matplotlib.transforms as mtrans
>>> ax = plt.axes()
>>> ax.plot(range(10), transform=mtrans.Affine2D().scale(10) + ax.transData)
>>> print(ax.viewLim)
Bbox('array([[  0.,   0.],\n       [ 90.,  90.]])')
```

- One can now easily get a transform which goes from one transform's coordinate system to another, in an optimized way, using the new subtract method on a transform. For instance, to go from data coordinates to axes coordinates:

```
>>> import matplotlib.pyplot as plt
>>> ax = plt.axes()
>>> data2ax = ax.transData - ax.transAxes
>>> print(ax.transData.depth, ax.transAxes.depth)
3, 1
>>> print(data2ax.depth)
2
```

  for versions before 1.2 this could only be achieved in a sub-optimal way, using `ax.transData + ax.transAxes.inverted()` (depth is a new concept, but had it existed it would return 4 for this example).

- `twinx` and `twiny` now returns an instance of SubplotBase if parent axes is an instance of SubplotBase.
- All Qt3-based backends are now deprecated due to the lack of py3k bindings. Qt and QtAgg backends will continue to work in v1.2.x for py2.6 and py2.7. It is anticipated that the Qt3 support will be completely removed for the next release.
- `matplotlib.colors.ColorConverter`, *Colormap* and *Normalize* now subclasses `object`
- ContourSet instances no longer have a `transform` attribute. Instead, access the transform with the `get_transform` method.

16.1.20 Changes in 1.1.x

- Added new *matplotlib.sankey.Sankey* for generating Sankey diagrams.
- In *imshow()*, setting *interpolation* to 'nearest' will now always mean that the nearest-neighbor interpolation is performed. If you want the no-op interpolation to be performed, choose 'none'.
- There were errors in how the tri-functions were handling input parameters that had to be fixed. If your tri-plots are not working correctly anymore, or you were working around apparent mistakes, please see issue #203 in the github tracker. When in doubt, use kwargs.

- The 'symlog' scale had some bad behavior in previous versions. This has now been fixed and users should now be able to use it without frustrations. The fixes did result in some minor changes in appearance for some users who may have been depending on the bad behavior.
- There is now a common set of markers for all plotting functions. Previously, some markers existed only for *scatter()* or just for *plot()*. This is now no longer the case. This merge did result in a conflict. The string 'd' now means "thin diamond" while 'D' will mean "regular diamond".

16.1.21 Changes beyond 0.99.x

- The default behavior of *matplotlib.axes.Axes.set_xlim()*, *matplotlib.axes.Axes.set_ylim()*, and *matplotlib.axes.Axes.axis()*, and their corresponding pyplot functions, has been changed: when view limits are set explicitly with one of these methods, autoscaling is turned off for the matching axis. A new *auto* kwarg is available to control this behavior. The limit kwargs have been renamed to *left* and *right* instead of *xmin* and *xmax*, and *bottom* and *top* instead of *ymin* and *ymax*. The old names may still be used, however.
- There are five new Axes methods with corresponding pyplot functions to facilitate autoscaling, tick location, and tick label formatting, and the general appearance of ticks and tick labels:
  - *matplotlib.axes.Axes.autoscale()* turns autoscaling on or off, and applies it.
  - *matplotlib.axes.Axes.margins()* sets margins used to autoscale the `matplotlib.axes.Axes.viewLim` based on the `matplotlib.axes.Axes.dataLim`.
  - *matplotlib.axes.Axes.locator_params()* allows one to adjust axes locator parameters such as *nbins*.
  - *matplotlib.axes.Axes.ticklabel_format()* is a convenience method for controlling the *matplotlib.ticker.ScalarFormatter* that is used by default with linear axes.
  - *matplotlib.axes.Axes.tick_params()* controls direction, size, visibility, and color of ticks and their labels.
- The *matplotlib.axes.Axes.bar()* method accepts a *error_kw* kwarg; it is a dictionary of kwargs to be passed to the errorbar function.
- The *matplotlib.axes.Axes.hist()* *color* kwarg now accepts a sequence of color specs to match a sequence of datasets.
- The *EllipseCollection* has been changed in two ways:
  - There is a new *units* option, 'xy', that scales the ellipse with the data units. This matches the :class:'~matplotlib.patches.Ellipse` scaling.
  - The *height* and *width* kwargs have been changed to specify the height and width, again for consistency with *Ellipse*, and to better match their names; previously they specified the half-height and half-width.

- There is a new rc parameter `axes.color_cycle`, and the color cycle is now independent of the rc parameter `lines.color`. `matplotlib.Axes.set_default_color_cycle` is deprecated.
- You can now print several figures to one pdf file and modify the document information dictionary of a pdf file. See the docstrings of the class *matplotlib.backends.backend_pdf.PdfPages* for more information.
- Removed configobj and enthought.traits packages, which are only required by the experimental traited config and are somewhat out of date. If needed, install them independently.
- The new rc parameter `savefig.extension` sets the filename extension that is used by *matplotlib.figure.Figure.savefig()* if its *fname* argument lacks an extension.
- In an effort to simplify the backend API, all clipping rectangles and paths are now passed in using GraphicsContext objects, even on collections and images. Therefore:

```
draw_path_collection(self, master_transform, cliprect, clippath,
                     clippath_trans, paths, all_transforms, offsets,
                     offsetTrans, facecolors, edgecolors, linewidths,
                     linestyles, antialiaseds, urls)

# is now

draw_path_collection(self, gc, master_transform, paths, all_transforms,
                     offsets, offsetTrans, facecolors, edgecolors,
                     linewidths, linestyles, antialiaseds, urls)


draw_quad_mesh(self, master_transform, cliprect, clippath,
               clippath_trans, meshWidth, meshHeight, coordinates,
               offsets, offsetTrans, facecolors, antialiased,
               showedges)

# is now

draw_quad_mesh(self, gc, master_transform, meshWidth, meshHeight,
               coordinates, offsets, offsetTrans, facecolors,
               antialiased, showedges)


draw_image(self, x, y, im, bbox, clippath=None, clippath_trans=None)

# is now

draw_image(self, gc, x, y, im)
```

- There are four new Axes methods with corresponding pyplot functions that deal with unstructured triangular grids:
  - *matplotlib.axes.Axes.tricontour()* draws contour lines on a triangular grid.

- *matplotlib.axes.Axes.tricontourf()* draws filled contours on a triangular grid.
- *matplotlib.axes.Axes.tripcolor()* draws a pseudocolor plot on a triangular grid.
- *matplotlib.axes.Axes.triplot()* draws a triangular grid as lines and/or markers.

16.1.22 Changes in 0.99

- pylab no longer provides a load and save function. These are available in matplotlib.mlab, or you can use numpy.loadtxt and numpy.savetxt for text files, or np.save and np.load for binary NumPy arrays.
- User-generated colormaps can now be added to the set recognized by *matplotlib.cm.get_cmap()*. Colormaps can be made the default and applied to the current image using *matplotlib.pyplot.set_cmap()*.
- changed use_mrecords default to False in mlab.csv2rec since this is partially broken
- Axes instances no longer have a "frame" attribute. Instead, use the new "spines" attribute. Spines is a dictionary where the keys are the names of the spines (e.g., 'left','right' and so on) and the values are the artists that draw the spines. For normal (rectilinear) axes, these artists are Line2D instances. For other axes (such as polar axes), these artists may be Patch instances.
- Polar plots no longer accept a resolution kwarg. Instead, each Path must specify its own number of interpolation steps. This is unlikely to be a user-visible change -- if interpolation of data is required, that should be done before passing it to Matplotlib.

16.1.23 Changes for 0.98.x

- `psd()`, `csd()`, and `cohere()` will now automatically wrap negative frequency components to the beginning of the returned arrays. This is much more sensible behavior and makes them consistent with `specgram()`. The previous behavior was more of an oversight than a design decision.
- Added new keyword parameters *nonposx*, *nonposy* to *matplotlib.axes.Axes* methods that set log scale parameters. The default is still to mask out non-positive values, but the kwargs accept 'clip', which causes non-positive values to be replaced with a very small positive value.
- Added new *matplotlib.pyplot.fignum_exists()* and *matplotlib.pyplot.get_fignums()*; they merely expose information that had been hidden in `matplotlib._pylab_helpers`.
- Deprecated numerix package.

- Added new *matplotlib.image.imsave()* and exposed it to the *matplotlib.pyplot* interface.
- Remove support for pyExcelerator in exceltools -- use xlwt instead
- Changed the defaults of acorr and xcorr to use usevlines=True, maxlags=10 and normed=True since these are the best defaults
- Following keyword parameters for *matplotlib.legend.Legend* are now deprecated and new set of parameters are introduced. The new parameters are given as a fraction of the font-size. Also, *scatteryoffsets*, *fancybox* and *columnspacing* are added as keyword parameters.

Deprecated	New
pad	borderpad
labelsep	labelspacing
handlelen	handlelength
handlestextsep	handletextpad
axespad	borderaxespad

- Removed the configobj and experimental traits rc support
- Modified *matplotlib.mlab.psd()*, *matplotlib.mlab.csd()*, *matplotlib.mlab.cohere()*, and *matplotlib.mlab.specgram()* to scale one-sided densities by a factor of 2. Also, optionally scale the densities by the sampling frequency, which gives true values of densities that can be integrated by the returned frequency values. This also gives better MATLAB compatibility. The corresponding *matplotlib.axes.Axes* methods and *matplotlib.pyplot* functions were updated as well.
- Font lookup now uses a nearest-neighbor approach rather than an exact match. Some fonts may be different in plots, but should be closer to what was requested.
- *matplotlib.axes.Axes.set_xlim()*, *matplotlib.axes.Axes.set_ylim()* now return a copy of the `viewlim` array to avoid modify-in-place surprises.
- *matplotlib.afm.AFM.get_fullname()* and *matplotlib.afm.AFM.get_familyname()* no longer raise an exception if the AFM file does not specify these optional attributes, but returns a guess based on the required FontName attribute.
- Changed precision kwarg in *matplotlib.pyplot.spy()*; default is 0, and the string value 'present' is used for sparse arrays only to show filled locations.
- *matplotlib.collections.EllipseCollection* added.
- Added `angles` kwarg to *matplotlib.pyplot.quiver()* for more flexible specification of the arrow angles.
- Deprecated (raise NotImplementedError) all the mlab2 functions from *matplotlib.mlab* out of concern that some of them were not clean room implementations.

- Methods *matplotlib.collections.Collection.get_offsets()* and *matplotlib.collections.Collection.set_offsets()* added to *Collection* base class.
- matplotlib.figure.Figure.figurePatch renamed matplotlib.figure.Figure.patch; matplotlib.axes.Axes.axesPatch renamed matplotlib.axes.Axes.patch; matplotlib.axes.Axes.axesFrame renamed matplotlib.axes.Axes.frame. matplotlib.axes.Axes.get_frame, which returns matplotlib.axes.Axes.patch, is deprecated.
- Changes in the *matplotlib.contour.ContourLabeler* attributes (*matplotlib.pyplot.clabel()* function) so that they all have a form like .labelAttribute. The three attributes that are most likely to be used by end users, .cl, .cl_xy and .cl_cvalues have been maintained for the moment (in addition to their renamed versions), but they are deprecated and will eventually be removed.
- Moved several functions in *matplotlib.mlab* and *matplotlib.cbook* into a separate module matplotlib.numerical_methods because they were unrelated to the initial purpose of mlab or cbook and appeared more coherent elsewhere.

16.1.24 Changes for 0.98.1

- Removed broken matplotlib.axes3d support and replaced it with a non-implemented error pointing to 0.91.x

16.1.25 Changes for 0.98.0

- *matplotlib.image.imread()* now no longer always returns RGBA data---if the image is luminance or RGB, it will return a MxN or MxNx3 array if possible. Also uint8 is no longer always forced to float.
- Rewrote the *matplotlib.cm.ScalarMappable* callback infrastructure to use *matplotlib.cbook.CallbackRegistry* rather than custom callback handling. Any users of matplotlib.cm.ScalarMappable.add_observer of the *ScalarMappable* should use the matplotlib.cm.ScalarMappable.callbacksSM *CallbackRegistry* instead.
- New axes function and Axes method provide control over the plot color cycle: matplotlib.axes.set_default_color_cycle and matplotlib.axes.Axes.set_color_cycle.
- Matplotlib now requires Python 2.4, so *matplotlib.cbook* will no longer provide set, enumerate(), reversed() or izip compatibility functions.
- In Numpy 1.0, bins are specified by the left edges only. The axes method *matplotlib.axes.Axes.hist()* now uses future Numpy 1.3 semantics for histograms. Providing binedges, the last value gives the upper-right edge now, which was implicitly set to +infinity in Numpy 1.0. This also means that the last bin doesn't contain upper outliers any more by default.

- New axes method and pyplot function, *hexbin()*, is an alternative to *scatter()* for large datasets. It makes something like a *pcolor()* of a 2-D histogram, but uses hexagonal bins.
- New kwarg, `symmetric`, in *matplotlib.ticker.MaxNLocator* allows one require an axis to be centered around zero.
- Toolkits must now be imported from `mpl_toolkits` (not `matplotlib.toolkits`)

Notes about the transforms refactoring

A major new feature of the 0.98 series is a more flexible and extensible transformation infrastructure, written in Python/Numpy rather than a custom C extension.

The primary goal of this refactoring was to make it easier to extend matplotlib to support new kinds of projections. This is mostly an internal improvement, and the possible user-visible changes it allows are yet to come.

See *matplotlib.transforms* for a description of the design of the new transformation framework.

For efficiency, many of these functions return views into Numpy arrays. This means that if you hold on to a reference to them, their contents may change. If you want to store a snapshot of their current values, use the Numpy array method copy().

The view intervals are now stored only in one place -- in the *matplotlib.axes.Axes* instance, not in the locator instances as well. This means locators must get their limits from their *matplotlib.axis.Axis*, which in turn looks up its limits from the *Axes*. If a locator is used temporarily and not assigned to an Axis or Axes, (e.g., in *matplotlib.contour*), a dummy axis must be created to store its bounds. Call *matplotlib.ticker.TickHelper.create_dummy_axis()* to do so.

The functionality of `Pbox` has been merged with *Bbox*. Its methods now all return copies rather than modifying in place.

The following lists many of the simple changes necessary to update code from the old transformation framework to the new one. In particular, methods that return a copy are named with a verb in the past tense, whereas methods that alter an object in place are named with a verb in the present tense.

`matplotlib.transforms`

Old method	New method
`Bbox.get_bounds`	*transforms.Bbox.bounds*
`Bbox.width`	*transforms.Bbox.width*
`Bbox.height`	*transforms.Bbox.height*
`Bbox.intervalx().get_bounds()` `Bbox.intervalx().set_bounds()`	*transforms.Bbox.intervalx* [It is now a property.]
`Bbox.intervaly().get_bounds()` `Bbox.intervaly().set_bounds()`	*transforms.Bbox.intervaly* [It is now a property.]
`Bbox.xmin`	*transforms.Bbox.x0* or *transforms.Bbox.xmin*[1]
`Bbox.ymin`	*transforms.Bbox.y0* or *transforms.Bbox.ymin*[1]
`Bbox.xmax`	*transforms.Bbox.x1* or *transforms.Bbox.xmax*[1]
`Bbox.ymax`	*transforms.Bbox.y1* or *transforms.Bbox.ymax*[1]
`Bbox.overlaps(bboxes)`	*Bbox.count_overlaps(bboxes)*
`bbox_all(bboxes)`	*Bbox.union(bboxes)* [It is a staticmethod.]
`lbwh_to_bbox(l, b, w, h)`	*Bbox.from_bounds(x0, y0, w, h)* [It is a staticmethod.]
`inverse_transform_bbox(trans, bbox)`	*Bbox.inverse_transformed(trans)*
`Interval.contains_open(v)`	*interval_contains_open(tuple, v)*
`Interval.contains(v)`	*interval_contains(tuple, v)*
`identity_transform()`	*transforms.IdentityTransform*
`blend_xy_sep_transform(xtrans, ytrans)`	*blended_transform_factory(xtrans, ytrans)*
`scale_transform(xs, ys)`	*Affine2D().scale(xs[, ys])*
`get_bbox_transform(boxin, boxout)`	*BboxTransform(boxin, boxout)* or *BboxTransformFrom(boxin)* or *BboxTransformTo(boxout)*
`Transform.seq_xy_tup(points)`	*Transform.transform(points)*
`Transform.inverse_xy_tup(points)`	*Transform.inverted()*.transform(points)

[1] The *Bbox* is bound by the points (x0, y0) to (x1, y1) and there is no defined order to these points, that is, x0 is not necessarily the left edge of the box. To get the left edge of the *Bbox*, use the read-only property *xmin*.

matplotlib.axes

Old method	New method
`Axes.get_position()`	*matplotlib.axes.Axes.get_position()*[2]
`Axes.set_position()`	*matplotlib.axes.Axes.set_position()*[3]
`Axes.toggle_log_lineary()`	*matplotlib.axes.Axes.set_yscale()*[4]
`Subplot` class	removed

The `Polar` class has moved to *matplotlib.projections.polar*.

matplotlib.artist

Old method	New method
`Artist.set_clip_path(path)`	`Artist.set_clip_path(path, transform)`[5]

matplotlib.collections

Old method	New method
linestyle	*linestyles*[6]

matplotlib.colors

Old method	New method
`ColorConvertor.to_rgba_list(c)`	`colors.to_rgba_array(c)` [*matplotlib.colors.to_rgba_array()* returns an Nx4 NumPy array of RGBA color quadruples.]

[2] *matplotlib.axes.Axes.get_position()* used to return a list of points, now it returns a *matplotlib.transforms.Bbox* instance.

[3] *matplotlib.axes.Axes.set_position()* now accepts either four scalars or a *matplotlib.transforms.Bbox* instance.

[4] Since the recfactoring allows for more than two scale types ('log' or 'linear'), it no longer makes sense to have a toggle. `Axes.toggle_log_lineary()` has been removed.

[5] *matplotlib.artist.Artist.set_clip_path()* now accepts a *matplotlib.path.Path* instance and a *matplotlib.transforms.Transform* that will be applied to the path immediately before clipping.

[6] Linestyles are now treated like all other collection attributes, i.e. a single value or multiple values may be provided.

matplotlib.contour

Old method	New method
`Contour._segments`	`matplotlib.contour.Contour.get_paths` [Returns a list of *matplotlib.path.Path* instances.]

matplotlib.figure

Old method	New method
`Figure.dpi.get() Figure.dpi.set()`	*matplotlib.figure.Figure.dpi (a property)*

matplotlib.patches

Old method	New method
`Patch.get_verts()`	*matplotlib.patches.Patch.get_path()* [Returns a *matplotlib.path.Path* instance]

matplotlib.backend_bases

Old method	New method
`GraphicsContext.set_clip_rectangle(tuple)`	*GraphicsContext.set_clip_rectangle(bbox)*
`GraphicsContext.get_clip_path()`	*GraphicsContext.get_clip_path()*[7]
`GraphicsContext.set_clip_path()`	*GraphicsContext.set_clip_path()*[8]

RendererBase

New methods:

- *draw_path(self, gc, path, transform, rgbFace)*
- *draw_markers(self, gc, marker_path, marker_trans, path, trans, rgbFace)*

[7] *matplotlib.backend_bases.GraphicsContextBase.get_clip_path()* returns a tuple of the form (*path*, *affine_transform*), where *path* is a *matplotlib.path.Path* instance and *affine_transform* is a *matplotlib.transforms.Affine2D* instance.

[8] *matplotlib.backend_bases.GraphicsContextBase.set_clip_path()* now only accepts a *matplotlib.transforms.TransformedPath* instance.

- *draw_path_collection(self, master_transform, cliprect, clippath, clippath_trans, paths, all_transforms, offsets, offsetTrans, facecolors, edgecolors, linewidths, linestyles, antialiaseds)* *[optional]*

Changed methods:

- `draw_image(self, x, y, im, bbox)` is now *draw_image(self, x, y, im, bbox, clippath, clippath_trans)*

Removed methods:

- `draw_arc`
- `draw_line_collection`
- `draw_line`
- `draw_lines`
- `draw_point`
- `draw_quad_mesh`
- `draw_poly_collection`
- `draw_polygon`
- `draw_rectangle`
- `draw_regpoly_collection`

16.1.26 Changes for 0.91.2

- For `csv2rec`, checkrows=0 is the new default indicating all rows will be checked for type inference
- A warning is issued when an image is drawn on log-scaled axes, since it will not log-scale the image data.
- Moved `rec2gtk` to `matplotlib.toolkits.gtktools`
- Moved `rec2excel` to `matplotlib.toolkits.exceltools`
- Removed, dead/experimental ExampleInfo, Namespace and Importer code from `matplotlib`

16.1.27 Changes for 0.91.0

- Changed `cbook.is_file_like` to `cbook.is_writable_file_like` and corrected behavior.
- Added *ax* keyword argument to *pyplot.colorbar()* and *Figure.colorbar()* so that one can specify the axes object from which space for the colorbar is to be taken, if one does not want to make the colorbar axes manually.
- Changed `cbook.reversed` so it yields a tuple rather than a (index, tuple). This agrees with the Python reversed builtin, and cbook only defines reversed if Python doesn't provide the builtin.
- Made skiprows=1 the default on `csv2rec`
- The gd and paint backends have been deleted.
- The errorbar method and function now accept additional kwargs so that upper and lower limits can be indicated by capping the bar with a caret instead of a straight line segment.
- The *matplotlib.dviread* file now has a parser for files like psfonts.map and pdftex.map, to map TeX font names to external files.
- The file *matplotlib.type1font* contains a new class for Type 1 fonts. Currently it simply reads pfa and pfb format files and stores the data in a way that is suitable for embedding in pdf files. In the future the class might actually parse the font to allow e.g., subsetting.
- `matplotlib.ft2font` now supports `FT_Attach_File`. In practice this can be used to read an afm file in addition to a pfa/pfb file, to get metrics and kerning information for a Type 1 font.
- The *AFM* class now supports querying CapHeight and stem widths. The get_name_char method now has an isord kwarg like get_width_char.
- Changed *pcolor()* default to `shading='flat'`; but as noted now in the docstring, it is preferable to simply use the *edgecolor* keyword argument.
- The mathtext font commands (`\cal`, `\rm`, `\it`, `\tt`) now behave as TeX does: they are in effect until the next font change command or the end of the grouping. Therefore uses of `$\cal{R}$` should be changed to `${\cal R}$`. Alternatively, you may use the new LaTeX-style font commands (`\mathcal`, `\mathrm`, `\mathit`, `\mathtt`) which do affect the following group, e.g., `$\mathcal{R}$`.
- Text creation commands have a new default linespacing and a new `linespacing` kwarg, which is a multiple of the maximum vertical extent of a line of ordinary text. The default is 1.2; `linespacing=2` would be like ordinary double spacing, for example.
- Changed default kwarg in *matplotlib.colors.Normalize* to `clip=False`; clipping silently defeats the purpose of the special over, under, and bad values in the colormap, thereby leading to unexpected behavior. The new default should reduce such surprises.

- Made the emit property of *set_xlim()* and *set_ylim()* `True` by default; removed the Axes custom callback handling into a 'callbacks' attribute which is a *CallbackRegistry* instance. This now supports the 'xlim_changed' and 'ylim_changed' Axes events.

16.1.28 Changes for 0.90.1

```
The file dviread.py has a (very limited and fragile) dvi reader
for usetex support. The API might change in the future so don't
depend on it yet.

Removed deprecated support for a float value as a gray-scale;
now it must be a string, like '0.5'.  Added alpha kwarg to
ColorConverter.to_rgba_list.

New method set_bounds(vmin, vmax) for formatters, locators sets
the viewInterval and dataInterval from floats.

Removed deprecated colorbar_classic.

Line2D.get_xdata and get_ydata valid_only=False kwarg is replaced
by orig=True.  When True, it returns the original data, otherwise
the processed data (masked, converted)

Some modifications to the units interface.
units.ConversionInterface.tickers renamed to
units.ConversionInterface.axisinfo and it now returns a
units.AxisInfo object rather than a tuple.  This will make it
easier to add axis info functionality (e.g., I added a default label
on this iteration) w/o having to change the tuple length and hence
the API of the client code every time new functionality is added.
Also, units.ConversionInterface.convert_to_value is now simply
named units.ConversionInterface.convert.

Axes.errorbar uses Axes.vlines and Axes.hlines to draw its error
limits int he vertical and horizontal direction.  As you'll see
in the changes below, these functions now return a LineCollection
rather than a list of lines.  The new return signature for
errorbar is  ylins, caplines, errorcollections where
errorcollections is a xerrcollection, yerrcollection

Axes.vlines and Axes.hlines now create and returns a LineCollection, not a list
of lines.  This is much faster.  The kwarg signature has changed,
so consult the docs

MaxNLocator accepts a new Boolean kwarg ('integer') to force
ticks to integer locations.

Commands that pass an argument to the Text constructor or to
Text.set_text() now accept any object that can be converted
with '%s'.  This affects xlabel(), title(), etc.
```

(continues on next page)

(continued from previous page)

```
Barh now takes a **kwargs dict instead of most of the old
arguments. This helps ensure that bar and barh are kept in sync,
but as a side effect you can no longer pass e.g., color as a
positional argument.

ft2font.get_charmap() now returns a dict that maps character codes
to glyph indices (until now it was reversed)

Moved data files into lib/matplotlib so that setuptools' develop
mode works. Re-organized the mpl-data layout so that this source
structure is maintained in the installation. (i.e., the 'fonts' and
'images' sub-directories are maintained in site-packages.).
Suggest removing site-packages/matplotlib/mpl-data and
~/.matplotlib/ttffont.cache before installing
```

16.1.29 Changes for 0.90.0

```
All artists now implement a "pick" method which users should not
call.  Rather, set the "picker" property of any artist you want to
pick on (the epsilon distance in points for a hit test) and
register with the "pick_event" callback.  See
examples/pick_event_demo.py for details

Bar, barh, and hist have "log" binary kwarg: log=True
sets the ordinate to a log scale.

Boxplot can handle a list of vectors instead of just
an array, so vectors can have different lengths.

Plot can handle 2-D x and/or y; it plots the columns.

Added linewidth kwarg to bar and barh.

Made the default Artist._transform None (rather than invoking
identity_transform for each artist only to have it overridden
later).  Use artist.get_transform() rather than artist._transform,
even in derived classes, so that the default transform will be
created lazily as needed

New LogNorm subclass of Normalize added to colors.py.
All Normalize subclasses have new inverse() method, and
the __call__() method has a new clip kwarg.

Changed class names in colors.py to match convention:
normalize -> Normalize, no_norm -> NoNorm.  Old names
are still available for now.

Removed obsolete pcolor_classic command and method.
```

(continues on next page)

(continued from previous page)

```
Removed lineprops and markerprops from the Annotation code and
replaced them with an arrow configurable with kwarg arrowprops.
See examples/annotation_demo.py - JDH
```

16.1.30 Changes for 0.87.7

```
Completely reworked the annotations API because I found the old
API cumbersome.  The new design is much more legible and easy to
read.  See matplotlib.text.Annotation and
examples/annotation_demo.py

markeredgecolor and markerfacecolor cannot be configured in
matplotlibrc any more. Instead, markers are generally colored
automatically based on the color of the line, unless marker colors
are explicitly set as kwargs - NN

Changed default comment character for load to '#' - JDH

math_parse_s_ft2font_svg from mathtext.py & mathtext2.py now returns
width, height, svg_elements. svg_elements is an instance of Bunch (
cmbook.py) and has the attributes svg_glyphs and svg_lines, which are both
lists.

Renderer.draw_arc now takes an additional parameter, rotation.
It specifies to draw the artist rotated in degrees anti-
clockwise.  It was added for rotated ellipses.

Renamed Figure.set_figsize_inches to Figure.set_size_inches to
better match the get method, Figure.get_size_inches.

Removed the copy_bbox_transform from transforms.py; added
shallowcopy methods to all transforms.  All transforms already
had deepcopy methods.

FigureManager.resize(width, height): resize the window
specified in pixels

barh: x and y args have been renamed to width and bottom
respectively, and their order has been swapped to maintain
a (position, value) order.

bar and barh: now accept kwarg 'edgecolor'.

bar and barh: The left, height, width and bottom args can
now all be scalars or sequences; see docstring.

barh: now defaults to edge aligned instead of center
aligned bars
```

(continues on next page)

(continued from previous page)

```
bar, barh and hist: Added a keyword arg 'align' that
controls between edge or center bar alignment.

Collections: PolyCollection and LineCollection now accept
vertices or segments either in the original form [(x,y),
(x,y), ...] or as a 2D numerix array, with X as the first column
and Y as the second. Contour and quiver output the numerix
form.  The transforms methods Bbox.update() and
Transformation.seq_xy_tups() now accept either form.

Collections: LineCollection is now a ScalarMappable like
PolyCollection, etc.

Specifying a grayscale color as a float is deprecated; use
a string instead, e.g., 0.75 -> '0.75'.

Collections: initializers now accept any mpl color arg, or
sequence of such args; previously only a sequence of rgba
tuples was accepted.

Colorbar: completely new version and api; see docstring.  The
original version is still accessible as colorbar_classic, but
is deprecated.

Contourf: "extend" kwarg replaces "clip_ends"; see docstring.
Masked array support added to pcolormesh.

Modified aspect-ratio handling:
    Removed aspect kwarg from imshow
    Axes methods:
        set_aspect(self, aspect, adjustable=None, anchor=None)
        set_adjustable(self, adjustable)
        set_anchor(self, anchor)
    Pylab interface:
        axis('image')

 Backend developers: ft2font's load_char now takes a flags
 argument, which you can OR together from the LOAD_XXX
 constants.
```

16.1.31 Changes for 0.86

```
Matplotlib data is installed into the matplotlib module.
This is similar to package_data.  This should get rid of
having to check for many possibilities in _get_data_path().
The MATPLOTLIBDATA env key is still checked first to allow
for flexibility.
```

(continues on next page)

(continued from previous page)

```
1) Separated the color table data from cm.py out into
a new file, _cm.py, to make it easier to find the actual
code in cm.py and to add new colormaps. Everything
from _cm.py is imported by cm.py, so the split should be
transparent.
2) Enabled automatic generation of a colormap from
a list of colors in contour; see modified
examples/contour_demo.py.
3) Support for imshow of a masked array, with the
ability to specify colors (or no color at all) for
masked regions, and for regions that are above or
below the normally mapped region.  See
examples/image_masked.py.
4) In support of the above, added two new classes,
ListedColormap, and no_norm, to colors.py, and modified
the Colormap class to include common functionality. Added
a clip kwarg to the normalize class.
```

16.1.32 Changes for 0.85

```
Made xtick and ytick separate props in rc

made pos=None the default for tick formatters rather than 0 to
indicate "not supplied"

Removed "feature" of minor ticks which prevents them from
overlapping major ticks.  Often you want major and minor ticks at
the same place, and can offset the major ticks with the pad.  This
could be made configurable

Changed the internal structure of contour.py to a more OO style.
Calls to contour or contourf in axes.py or pylab.py now return
a ContourSet object which contains references to the
LineCollections or PolyCollections created by the call,
as well as the configuration variables that were used.
The ContourSet object is a "mappable" if a colormap was used.

Added a clip_ends kwarg to contourf. From the docstring:
         * clip_ends = True
           If False, the limits for color scaling are set to the
           minimum and maximum contour levels.
           True (default) clips the scaling limits.  Example:
           if the contour boundaries are V = [-100, 2, 1, 0, 1, 2, 100],
           then the scaling limits will be [-100, 100] if clip_ends
           is False, and [-3, 3] if clip_ends is True.
Added kwargs linewidths, antialiased, and nchunk to contourf.  These
are experimental; see the docstring.

Changed Figure.colorbar():
```

(continues on next page)

(continued from previous page)

```
    kw argument order changed;
    if mappable arg is a non-filled ContourSet, colorbar() shows
            lines instead hof polygons.
    if mappable arg is a filled ContourSet with clip_ends=True,
            the endpoints are not labelled, so as to give the
            correct impression of open-endedness.

Changed LineCollection.get_linewidths to get_linewidth, for
consistency.
```

16.1.33 Changes for 0.84

```
Unified argument handling between hlines and vlines.  Both now
take optionally a fmt argument (as in plot) and a keyword args
that can be passed onto Line2D.

Removed all references to "data clipping" in rc and lines.py since
these were not used and not optimized.  I'm sure they'll be
resurrected later with a better implementation when needed.

'set' removed - no more deprecation warnings.  Use 'setp' instead.

Backend developers: Added flipud method to image and removed it
from to_str.  Removed origin kwarg from backend.draw_image.
origin is handled entirely by the frontend now.
```

16.1.34 Changes for 0.83

```
- Made HOME/.matplotlib the new config dir where the matplotlibrc
  file, the ttf.cache, and the tex.cache live.  The new default
  filenames in .matplotlib have no leading dot and are not hidden.
  e.g., the new names are matplotlibrc, tex.cache, and ttffont.cache.
  This is how ipython does it so it must be right.

  If old files are found, a warning is issued and they are moved to
  the new location.

- backends/__init__.py no longer imports new_figure_manager,
  draw_if_interactive and show from the default backend, but puts
  these imports into a call to pylab_setup.  Also, the Toolbar is no
  longer imported from WX/WXAgg.  New usage:

    from backends import pylab_setup
    new_figure_manager, draw_if_interactive, show = pylab_setup()

- Moved Figure.get_width_height() to FigureCanvasBase. It now
  returns int instead of float.
```

16.1.35 Changes for 0.82

```
- toolbar import change in GTKAgg, GTKCairo and WXAgg

- Added subplot config tool to GTK* backends -- note you must now
  import the NavigationToolbar2 from your backend of choice rather
  than from backend_gtk because it needs to know about the backend
  specific canvas -- see examples/embedding_in_gtk2.py.  Ditto for
  wx backend -- see examples/embedding_in_wxagg.py


- hist bin change

    Sean Richards notes there was a problem in the way we created
    the binning for histogram, which made the last bin
    underrepresented.  From his post:

      I see that hist uses the linspace function to create the bins
      and then uses searchsorted to put the values in their correct
      bin. That's all good but I am confused over the use of linspace
      for the bin creation. I wouldn't have thought that it does
      what is needed, to quote the docstring it creates a "Linear
      spaced array from min to max". For it to work correctly
      shouldn't the values in the bins array be the same bound for
      each bin? (i.e. each value should be the lower bound of a
      bin). To provide the correct bins for hist would it not be
      something like

      def bins(xmin, xmax, N):
        if N==1: return xmax
        dx = (xmax-xmin)/N # instead of N-1
        return xmin + dx*arange(N)


     This suggestion is implemented in 0.81.  My test script with these
     changes does not reveal any bias in the binning

      from matplotlib.numerix.mlab import randn, rand, zeros, Float
      from matplotlib.mlab import hist, mean

      Nbins = 50
      Ntests = 200
      results = zeros((Ntests,Nbins), typecode=Float)
      for i in range(Ntests):
          print 'computing', i
          x = rand(10000)
          n, bins = hist(x, Nbins)
          results[i] = n
      print mean(results)
```

16.1.36 Changes for 0.81

```
- pylab and artist "set" functions renamed to setp to avoid clash
  with python2.4 built-in set.  Current version will issue a
  deprecation warning which will be removed in future versions

- imshow interpolation arguments changes for advanced interpolation
  schemes.  See help imshow, particularly the interpolation,
  filternorm and filterrad kwargs

- Support for masked arrays has been added to the plot command and
  to the Line2D object.  Only the valid points are plotted.  A
  "valid_only" kwarg was added to the get_xdata() and get_ydata()
  methods of Line2D; by default it is False, so that the original
  data arrays are returned. Setting it to True returns the plottable
  points.

- contour changes:

  Masked arrays: contour and contourf now accept masked arrays as
    the variable to be contoured.  Masking works correctly for
    contour, but a bug remains to be fixed before it will work for
    contourf.  The "badmask" kwarg has been removed from both
    functions.

   Level argument changes:

     Old version: a list of levels as one of the positional
     arguments specified the lower bound of each filled region; the
     upper bound of the last region was taken as a very large
     number.  Hence, it was not possible to specify that z values
     between 0 and 1, for example, be filled, and that values
     outside that range remain unfilled.

     New version: a list of N levels is taken as specifying the
     boundaries of N-1 z ranges.  Now the user has more control over
     what is colored and what is not.  Repeated calls to contourf
     (with different colormaps or color specifications, for example)
     can be used to color different ranges of z.  Values of z
     outside an expected range are left uncolored.

     Example:
       Old: contourf(z, [0, 1, 2]) would yield 3 regions: 0-1, 1-2, and >2.
       New: it would yield 2 regions: 0-1, 1-2.  If the same 3 regions were
       desired, the equivalent list of levels would be [0, 1, 2,
       1e38].
```

16.1.37 Changes for 0.80

```
- xlim/ylim/axis always return the new limits regardless of
  arguments.  They now take kwargs which allow you to selectively
  change the upper or lower limits while leaving unnamed limits
  unchanged.  See help(xlim) for example
```

16.1.38 Changes for 0.73

```
- Removed deprecated ColormapJet and friends

- Removed all error handling from the verbose object

- figure num of zero is now allowed
```

16.1.39 Changes for 0.72

```
- Line2D, Text, and Patch copy_properties renamed update_from and
  moved into artist base class

- LineCollecitons.color renamed to LineCollections.set_color for
  consistency with set/get introspection mechanism,

- pylab figure now defaults to num=None, which creates a new figure
  with a guaranteed unique number

- contour method syntax changed - now it is MATLAB compatible

    unchanged: contour(Z)
    old: contour(Z, x=Y, y=Y)
    new: contour(X, Y, Z)

  see http://matplotlib.sf.net/matplotlib.pylab.html#-contour


 - Increased the default resolution for save command.

 - Renamed the base attribute of the ticker classes to _base to avoid conflict
   with the base method.  Sitt for subs

 - subs=none now does autosubbing in the tick locator.

 - New subplots that overlap old will delete the old axes.  If you
   do not want this behavior, use fig.add_subplot or the axes
   command
```

16.1.40 Changes for 0.71

```
Significant numerix namespace changes, introduced to resolve
namespace clashes between python built-ins and mlab names.
Refactored numerix to maintain separate modules, rather than
folding all these names into a single namespace.  See the following
mailing list threads for more information and background

  http://sourceforge.net/mailarchive/forum.php?thread_id=6398890&forum_id=36187
  http://sourceforge.net/mailarchive/forum.php?thread_id=6323208&forum_id=36187


OLD usage

  from matplotlib.numerix import array, mean, fft

NEW usage

  from matplotlib.numerix import array
  from matplotlib.numerix.mlab import mean
  from matplotlib.numerix.fft import fft

numerix dir structure mirrors numarray (though it is an incomplete
implementation)

  numerix
  numerix/mlab
  numerix/linear_algebra
  numerix/fft
  numerix/random_array

but of course you can use 'numerix : Numeric' and still get the
symbols.

pylab still imports most of the symbols from Numerix, MLab, fft,
etc, but is more cautious.  For names that clash with python names
(min, max, sum), pylab keeps the builtins and provides the numeric
versions with an a* prefix, e.g., (amin, amax, asum)
```

16.1.41 Changes for 0.70

```
MplEvent factored into a base class Event and derived classes
MouseEvent and KeyEvent

Removed definct set_measurement in wx toolbar
```

16.1.42 Changes for 0.65.1

```
removed add_axes and add_subplot from backend_bases.  Use
figure.add_axes and add_subplot instead.  The figure now manages the
current axes with gca and sca for get and set current axes.  If you
have code you are porting which called, e.g., figmanager.add_axes, you
can now simply do figmanager.canvas.figure.add_axes.
```

16.1.43 Changes for 0.65

```
mpl_connect and mpl_disconnect in the MATLAB interface renamed to
connect and disconnect

Did away with the text methods for angle since they were ambiguous.
fontangle could mean fontstyle (obligue, etc) or the rotation of the
text.  Use style and rotation instead.
```

16.1.44 Changes for 0.63

```
Dates are now represented internally as float days since 0001-01-01,
UTC.

All date tickers and formatters are now in matplotlib.dates, rather
than matplotlib.tickers

converters have been abolished from all functions and classes.
num2date and date2num are now the converter functions for all date
plots

Most of the date tick locators have a different meaning in their
constructors.  In the prior implementation, the first argument was a
base and multiples of the base were ticked.  e.g.,

  HourLocator(5)  # old: tick every 5 minutes

In the new implementation, the explicit points you want to tick are
provided as a number or sequence

```

(continues on next page)

(continued from previous page)

```
   HourLocator(range(0,5,61))  # new: tick every 5 minutes

This gives much greater flexibility.  I have tried to make the
default constructors (no args) behave similarly, where possible.

Note that YearLocator still works under the base/multiple scheme.
The difference between the YearLocator and the other locators is
that years are not recurrent.


Financial functions:

  matplotlib.finance.quotes_historical_yahoo(ticker, date1, date2)

   date1, date2 are now datetime instances.  Return value is a list
   of quotes where the quote time is a float - days since gregorian
   start, as returned by date2num

   See examples/finance_demo.py for example usage of new API
```

16.1.45 Changes for 0.61

```
canvas.connect is now deprecated for event handling.  use
mpl_connect and mpl_disconnect instead.  The callback signature is
func(event) rather than func(widget, event)
```

16.1.46 Changes for 0.60

```
ColormapJet and Grayscale are deprecated.  For backwards
compatibility, they can be obtained either by doing

  from matplotlib.cm import ColormapJet

or

  from matplotlib.matlab import *

They are replaced by cm.jet and cm.grey
```

16.1.47 Changes for 0.54.3

```
removed the set_default_font / get_default_font scheme from the
font_manager to unify customization of font defaults with the rest of
the rc scheme.  See examples/font_properties_demo.py and help(rc) in
matplotlib.matlab.
```

16.1.48 Changes for 0.54

MATLAB interface

dpi

Several of the backends used a PIXELS_PER_INCH hack that I added to try and make images render consistently across backends. This just complicated matters. So you may find that some font sizes and line widths appear different than before. Apologies for the inconvenience. You should set the dpi to an accurate value for your screen to get true sizes.

pcolor and scatter

There are two changes to the MATLAB interface API, both involving the patch drawing commands. For efficiency, pcolor and scatter have been rewritten to use polygon collections, which are a new set of objects from matplotlib.collections designed to enable efficient handling of large collections of objects. These new collections make it possible to build large scatter plots or pcolor plots with no loops at the python level, and are significantly faster than their predecessors. The original pcolor and scatter functions are retained as pcolor_classic and scatter_classic.

The return value from pcolor is a PolyCollection. Most of the properties that are available on rectangles or other patches are also available on PolyCollections, e.g., you can say:

```
c = scatter(blah, blah)
c.set_linewidth(1.0)
c.set_facecolor('r')
c.set_alpha(0.5)
```

or:

```
c = scatter(blah, blah)
set(c, 'linewidth', 1.0, 'facecolor', 'r', 'alpha', 0.5)
```

Because the collection is a single object, you no longer need to loop over the return value of scatter or pcolor to set properties for the entire list.

If you want the different elements of a collection to vary on a property, e.g., to have different line widths, see matplotlib.collections for a discussion on how to set the properties as a sequence.

For scatter, the size argument is now in points^2 (the area of the symbol in points) as in MATLAB and is not in data coords as before. Using sizes in data coords caused several problems. So you will need to adjust your size arguments accordingly or use scatter_classic.

mathtext spacing

For reasons not clear to me (and which I'll eventually fix) spacing no longer works in font groups. However, I added three new spacing commands which compensate for this '' (regular space), '/' (small space) and 'hspace{frac}' where frac is a fraction of fontsize in points. You will need to quote spaces in font strings, is:

```
title(r'$\rm{Histogram\ of\ IQ:}\ \mu=100,\ \sigma=15$')
```

Object interface - Application programmers

Autoscaling

The x and y axis instances no longer have autoscale view. These are handled by axes.autoscale_view

Axes creation

You should not instantiate your own Axes any more using the OO API. Rather, create a Figure as before and in place of:

```
f = Figure(figsize=(5,4), dpi=100)
a = Subplot(f, 111)
f.add_axis(a)
```

use:

```
f = Figure(figsize=(5,4), dpi=100)
a = f.add_subplot(111)
```

That is, add_axis no longer exists and is replaced by:

```
add_axes(rect, axisbg=defaultcolor, frameon=True)
add_subplot(num, axisbg=defaultcolor, frameon=True)
```

Artist methods

If you define your own Artists, you need to rename the _draw method to draw

Bounding boxes

matplotlib.transforms.Bound2D is replaced by matplotlib.transforms.Bbox. If you want to construct a bbox from left, bottom, width, height (the signature for Bound2D), use matplotlib.transforms.lbwh_to_bbox, as in

bbox = clickBBox = lbwh_to_bbox(left, bottom, width, height)

The Bbox has a different API than the Bound2D. e.g., if you want to get the width and height of the bbox

OLD::

width = fig.bbox.x.interval() height = fig.bbox.y.interval()

New::

width = fig.bbox.width() height = fig.bbox.height()

Object constructors

You no longer pass the bbox, dpi, or transforms to the various Artist constructors. The old way or creating lines and rectangles was cumbersome because you had to pass so many attributes to the Line2D and Rectangle classes not related directly to the geometry and properties of the object. Now default values are added to the object when you call axes.add_line or axes.add_patch, so they are hidden from the user.

If you want to define a custom transformation on these objects, call o.set_transform(trans) where trans is a Transformation instance.

In prior versions of you wanted to add a custom line in data coords, you would have to do

l = Line2D(dpi, bbox, x, y,

color = color, transx = transx, transy = transy,)

now all you need is

l = Line2D(x, y, color=color)

and the axes will set the transformation for you (unless you have set your own already, in which case it will eave it unchanged)

Transformations

The entire transformation architecture has been rewritten. Previously the x and y transformations where stored in the xaxis and yaxis instances. The problem with this approach is it only allows for separable transforms (where the x and y transformations don't depend on one another). But for cases like polar, they do. Now transformations operate on x,y together. There is a new base class matplotlib.transforms.Transformation and two concrete implementations, matplotlib.transforms.SeparableTransformation and matplotlib.transforms.Affine. The SeparableTransformation is constructed with the bounding box of the input (this determines the rectangular coordinate system of the input, i.e., the x and y view limits), the bounding box of the display, and possibly nonlinear transformations of x and y. The 2 most frequently used transformations, data coordinates -> display and axes coordinates -> display are available as ax.transData and ax.transAxes. See alignment_demo.py which uses axes coords.

Also, the transformations should be much faster now, for two reasons

- they are written entirely in extension code
- because they operate on x and y together, they can do the entire transformation in one loop. Earlier I did something along the lines of:

```
xt = sx*func(x) + tx
yt = sy*func(y) + ty
```

  Although this was done in numerix, it still involves 6 length(x) for-loops (the multiply, add, and function evaluation each for x and y). Now all of that is done in a single pass.

If you are using transformations and bounding boxes to get the cursor position in data coordinates, the method calls are a little different now. See the updated examples/coords_demo.py which shows you how to do this.

Likewise, if you are using the artist bounding boxes to pick items on the canvas with the GUI, the bbox methods are somewhat different. You will need to see the updated examples/object_picker.py.

See unit/transforms_unit.py for many examples using the new transformations.

16.1.49 Changes for 0.50

```
* refactored Figure class so it is no longer backend dependent.
  FigureCanvasBackend takes over the backend specific duties of the
  Figure.  matplotlib.backend_bases.FigureBase moved to
  matplotlib.figure.Figure.

* backends must implement FigureCanvasBackend (the thing that
  controls the figure and handles the events if any) and
  FigureManagerBackend (wraps the canvas and the window for MATLAB
  interface).  FigureCanvasBase implements a backend switching
  mechanism

* Figure is now an Artist (like everything else in the figure) and
  is totally backend independent

* GDFONTPATH renamed to TTFPATH

* backend faceColor argument changed to rgbFace

* colormap stuff moved to colors.py

* arg_to_rgb in backend_bases moved to class ColorConverter in
  colors.py

* GD users must upgrade to gd-2.0.22 and gdmodule-0.52 since new gd
  features (clipping, antialiased lines) are now used.

* Renderer must implement points_to_pixels

Migrating code:

MATLAB interface:

  The only API change for those using the MATLAB interface is in how
  you call figure redraws for dynamically updating figures.  In the
  old API, you did

    fig.draw()

  In the new API, you do

    manager = get_current_fig_manager()
    manager.canvas.draw()

  See the examples system_monitor.py, dynamic_demo.py, and anim.py

API

  There is one important API change for application developers.
  Figure instances used subclass GUI widgets that enabled them to be
  placed directly into figures.  e.g., FigureGTK subclassed
```

(continues on next page)

(continued from previous page)

```
  gtk.DrawingArea.  Now the Figure class is independent of the
  backend, and FigureCanvas takes over the functionality formerly
  handled by Figure.  In order to include figures into your apps,
  you now need to do, for example

    # gtk example
    fig = Figure(figsize=(5,4), dpi=100)
    canvas = FigureCanvasGTK(fig)  # a gtk.DrawingArea
    canvas.show()
    vbox.pack_start(canvas)

  If you use the NavigationToolbar, this in now initialized with a
  FigureCanvas, not a Figure.  The examples embedding_in_gtk.py,
  embedding_in_gtk2.py, and mpl_with_glade.py all reflect the new
  API so use these as a guide.

  All prior calls to

   figure.draw()  and
   figure.print_figure(args)

  should now be

   canvas.draw()  and
   canvas.print_figure(args)

  Apologies for the inconvenience.  This refactorization brings
  significant more freedom in developing matplotlib and should bring
  better plotting capabilities, so I hope the inconvenience is worth
  it.
```

16.1.50 Changes for 0.42

```
* Refactoring AxisText to be backend independent.  Text drawing and
  get_window_extent functionality will be moved to the Renderer.

* backend_bases.AxisTextBase is now text.Text module

* All the erase and reset functionality removed from AxisText - not
  needed with double buffered drawing.  Ditto with state change.
  Text instances have a get_prop_tup method that returns a hashable
  tuple of text properties which you can use to see if text props
  have changed, e.g., by caching a font or layout instance in a dict
  with the prop tup as a key -- see RendererGTK.get_pango_layout in
  backend_gtk for an example.

* Text._get_xy_display renamed Text.get_xy_display

* Artist set_renderer and wash_brushes methods removed
```

(continues on next page)

(continued from previous page)

```
* Moved Legend class from matplotlib.axes into matplotlib.legend

* Moved Tick, XTick, YTick, Axis, XAxis, YAxis from matplotlib.axes
  to matplotlib.axis

* moved process_text_args to matplotlib.text

* After getting Text handled in a backend independent fashion, the
  import process is much cleaner since there are no longer cyclic
  dependencies

* matplotlib.matlab._get_current_fig_manager renamed to
  matplotlib.matlab.get_current_fig_manager to allow user access to
  the GUI window attribute, e.g., figManager.window for GTK and
  figManager.frame for wx
```

16.1.51 Changes for 0.40

```
- Artist
    * __init__ takes a DPI instance and a Bound2D instance which is
      the bounding box of the artist in display coords
    * get_window_extent returns a Bound2D instance
    * set_size is removed; replaced by bbox and dpi
    * the clip_gc method is removed.  Artists now clip themselves with
      their box
    * added _clipOn boolean attribute.  If True, gc clip to bbox.

- AxisTextBase
    * Initialized with a transx, transy which are Transform instances
    * set_drawing_area removed
    * get_left_right and get_top_bottom are replaced by get_window_extent

- Line2D Patches now take transx, transy
    * Initialized with a transx, transy which are Transform instances

- Patches
   * Initialized with a transx, transy which are Transform instances

- FigureBase attributes dpi is a DPI instance rather than scalar and
  new attribute bbox is a Bound2D in display coords, and I got rid
  of the left, width, height, etc... attributes.  These are now
  accessible as, for example, bbox.x.min is left, bbox.x.interval()
  is width, bbox.y.max is top, etc...

- GcfBase attribute pagesize renamed to figsize

- Axes
    * removed figbg attribute
```

(continues on next page)

(continued from previous page)

```
    * added fig instance to __init__
    * resizing is handled by figure call to resize.

- Subplot
    * added fig instance to __init__

- Renderer methods for patches now take gcEdge and gcFace instances.
  gcFace=None takes the place of filled=False

- True and False symbols provided by cbook in a python2.3 compatible
  way

- new module transforms supplies Bound1D, Bound2D and Transform
  instances and more

- Changes to the MATLAB helpers API

  * _matlab_helpers.GcfBase is renamed by Gcf.  Backends no longer
    need to derive from this class.  Instead, they provide a factory
    function new_figure_manager(num, figsize, dpi).  The destroy
    method of the GcfDerived from the backends is moved to the derived
    FigureManager.

  * FigureManagerBase moved to backend_bases

  * Gcf.get_all_figwins renamed to Gcf.get_all_fig_managers

Jeremy:

  Make sure to self._reset = False in AxisTextWX._set_font.  This was
  something missing in my backend code.
```

Changes for the latest version are listed below. For new features that were added to Matplotlib, see *What's new?*

16.2 API Changes for 3.3.1

16.2.1 Deprecations

Reverted deprecation of `num2epoch` and `epoch2num`

These two functions were deprecated in 3.3.0, and did not return an accurate Matplotlib datenum relative to the new Matplotlib epoch handling (*`get_epoch`* and `rcParams["date.epoch"]` (default: `'1970-01-01T00:00:00'`)). This version reverts the deprecation.

Functions `epoch2num` and `dates.julian2num` use `date.epoch` rcParam

Now *`epoch2num`* and (undocumented) `julian2num` return floating point days since *`get_epoch`* as set by `rcParams["date.epoch"]` (default: `'1970-01-01T00:00:00'`), instead of floating point days since the old epoch of "0000-12-31T00:00:00". If needed, you can translate from the new to old values as `old = new + mdates.date2num(np.datetime64('0000-12-31'))`

16.3 API Changes for 3.3.0

16.3.1 Behaviour changes

`Formatter.fix_minus`

`Formatter.fix_minus` now performs hyphen-to-unicode-minus replacement whenever `rcParams["axes.unicode_minus"]` (default: `True`) is True; i.e. its behavior matches the one of `ScalarFormatter.fix_minus` (*`ScalarFormatter`* now just inherits that implementation).

This replacement is now used by the `format_data_short` method of the various builtin formatter classes, which affects the cursor value in the GUI toolbars.

`FigureCanvasBase` now always has a `manager` attribute, which may be None

Previously, it did not necessarily have such an attribute. A check for `hasattr(figure.canvas, "manager")` should now be replaced by `figure.canvas.manager is not None` (or `getattr(figure.canvas, "manager", None) is not None` for back-compatibility).

`cbook.CallbackRegistry` now propagates exceptions when no GUI event loop is running

cbook.CallbackRegistry now defaults to propagating exceptions thrown by callbacks when no interactive GUI event loop is running. If a GUI event loop *is* running, *cbook.CallbackRegistry* still defaults to just printing a traceback, as unhandled exceptions can make the program completely `abort()` in that case.

`Axes.locator_params()` validates `axis` parameter

axes.Axes.locator_params used to accept any value for `axis` and silently did nothing, when passed an unsupported value. It now raises a `ValueError`.

`Axis.set_tick_params()` validates `which` parameter

Axis.set_tick_params (and the higher level *axes.Axes.tick_params* and *pyplot.tick_params*) used to accept any value for `which` and silently did nothing, when passed an unsupported value. It now raises a `ValueError`.

`Axis.set_ticklabels()` must match `FixedLocator.locs`

If an axis is using a *ticker.FixedLocator*, typically set by a call to *Axis.set_ticks*, then the number of ticklabels supplied must match the number of locations available (`FixedFormattor.locs`). If not, a `ValueError` is raised.

`backend_pgf.LatexManager.latex`

`backend_pgf.LatexManager.latex` is now created with `encoding="utf-8"`, so its `stdin`, `stdout`, and `stderr` attributes are utf8-encoded.

`pyplot.xticks()` and `pyplot.yticks()`

Previously, passing labels without passing the ticks to either *pyplot.xticks* and *pyplot.yticks* would result in

> TypeError: object of type 'NoneType' has no len()

It now raises a `TypeError` with a proper description of the error.

Setting the same property under multiple aliases now raises a TypeError

Previously, calling e.g. `plot(..., color=somecolor, c=othercolor)` would emit a warning because `color` and `c` actually map to the same Artist property. This now raises a TypeError.

`FileMovieWriter` temporary frames directory

`FileMovieWriter` now defaults to writing temporary frames in a temporary directory, which is always cleared at exit. In order to keep the individual frames saved on the filesystem, pass an explicit *frame_prefix*.

`Axes.plot` no longer accepts *x* and *y* being both 2D and with different numbers of columns

Previously, calling *`Axes.plot`* e.g. with *x* of shape `(n, 3)` and *y* of shape `(n, 2)` would plot the first column of *x* against the first column of *y*, the second column of *x* against the second column of *y*, **and** the first column of *x* against the third column of *y*. This now raises an error instead.

`Text.update_from` now copies usetex state from the source Text

`stem` now defaults to `use_line_collection=True`

This creates the stem plot as a *`LineCollection`* rather than individual *`Line2D`* objects, greatly improving performance.

rcParams color validator is now stricter

Previously, rcParams entries whose values were color-like accepted "spurious" extra letters or characters in the "middle" of the string, e.g. `"(0, 1a, '0.5')"` would be interpreted as `(0, 1, 0.5)`. These extra characters (including the internal quotes) now cause a ValueError to be raised.

`SymLogNorm` now has a *base* parameter

Previously, *`SymLogNorm`* had no *base* keyword argument, and defaulted to `base=np.e` whereas the documentation said it was `base=10`. In preparation to make the default 10, calling *`SymLogNorm`* without the new *base* keyword argument emits a deprecation warning.

`errorbar` now color cycles when only errorbar color is set

Previously setting the *ecolor* would turn off automatic color cycling for the plot, leading to the the lines and markers defaulting to whatever the first color in the color cycle was in the case of multiple plot calls.

`rcsetup.validate_color_for_prop_cycle` now always raises TypeError for bytes input

It previously raised `TypeError`, **except** when the input was of the form `b"C[number]"` in which case it raised a ValueError.

`FigureCanvasPS.print_ps` and `FigureCanvasPS.print_eps` no longer apply edgecolor and facecolor

These methods now assume that the figure edge and facecolor have been correctly applied by *`FigureCanvasBase.print_figure`*, as they are normally called through it.

This behavior is consistent with other figure saving methods (*`FigureCanvasAgg.print_png`*, *`FigureCanvasPdf.print_pdf`*, *`FigureCanvasSVG.print_svg`*).

`pyplot.subplot()` now raises TypeError when given an incorrect number of arguments

This is consistent with other signature mismatch errors. Previously a ValueError was raised.

Shortcut for closing all figures

Shortcuts for closing all figures now also work for the classic toolbar. There is no default shortcut any more because unintentionally closing all figures by a key press might happen too easily. You can configure the shortcut yourself using `rcParams["keymap.quit_all"]` (default: `[]`).

Autoscale for arrow

Calling ax.arrow() will now autoscale the axes.

`set_tick_params(label1On=False)` now also makes the offset text (if any) invisible

... because the offset text can rarely be interpreted without tick labels anyways.

`Axes.annotate` and `pyplot.annotate` parameter name changed

The parameter `s` to *Axes.annotate* and *pyplot.annotate* is renamed to `text`, matching *Annotation*.

The old parameter name remains supported, but support for it will be dropped in a future Matplotlib release.

`font_manager.json_dump` now locks the font manager dump file

... to prevent multiple processes from writing to it at the same time.

`pyplot.rgrids` and `pyplot.thetagrids` now act as setters also when called with only kwargs

Previously, keyword arguments were silently ignored when no positional arguments were given.

`Axis.get_minorticklabels` and `Axis.get_majorticklabels` now returns plain list

Previously, *Axis.get_minorticklabels* and *Axis.get_majorticklabels* returns silent_list. Their return type is now changed to normal list. *get_xminorticklabels*, *get_yminorticklabels*, *get_zminorticklabels*, *Axis.get_ticklabels*, *get_xmajorticklabels*, *get_ymajorticklabels* and *get_zmajorticklabels* methods will be affected by this change.

Default slider formatter

The default method used to format *Slider* values has been changed to use a *ScalarFormatter* adapted the slider values limits. This should ensure that values are displayed with an appropriate number of significant digits even if they are much smaller or much bigger than 1. To restore the old behavior, explicitly pass a "%1.2f" as the *valfmt* parameter to *Slider*.

Add *normalize* keyword argument to `Axes.pie`

`pie()` used to draw a partial pie if the sum of the values was < 1. This behavior is deprecated and will change to always normalizing the values to a full pie by default. If you want to draw a partial pie, please pass `normalize=False` explicitly.

`table.CustomCell` is now an alias for `table.Cell`

All the functionality of `CustomCell` has been moved to its base class *Cell*.

wx Timer interval

Setting the timer interval on a not-yet-started `TimerWx` won't start it anymore.

"step"-type histograms default to the zorder of `Line2D`

This ensures that they go above gridlines by default. The old `zorder` can be kept by passing it as a keyword argument to *Axes.hist*.

`Legend` and `OffsetBox` visibility

Legend and *OffsetBox* subclasses (*PaddedBox*, *AnchoredOffsetbox*, and *AnnotationBbox*) no longer directly keep track of the visibility of their underlying *Patch* artist, but instead pass that flag down to the *Patch*.

`Legend` and `Table` no longer allow invalid locations

This affects legends produced on an Axes (*Axes.legend* and *pyplot.legend*) and on a Figure (*Figure.legend* and *pyplot.figlegend*). Figure legends also no longer accept the unsupported `'best'` location. Previously, invalid Axes locations would use `'best'` and invalid Figure locations would used `'upper right'`.

Passing Line2D's *drawstyle* together with *linestyle* is removed

Instead of `plt.plot(..., linestyle="steps--")`, use `plt.plot(..., linestyle="--", drawstyle="steps")`. `ds` is also an alias for `drawstyle`.

Upper case color strings

Support for passing single-letter colors (one of "rgbcmykw") as UPPERCASE characters is removed; these colors are now case-sensitive (lowercase).

tight/constrained_layout no longer worry about titles that are too wide

tight_layout and *constrained_layout* shrink axes to accommodate "decorations" on the axes. However, if an xlabel or title is too long in the x direction, making the axes smaller in the x-direction doesn't help. The behavior of both has been changed to ignore the width of the title and xlabel and the height of the ylabel in the layout logic.

This also means there is a new keyword argument for *axes.Axes.get_tightbbox* and *axis.Axis.get_tightbbox*: `for_layout_only`, which defaults to *False*, but if *True* returns a bounding box using the rules above.

`rcParams["savefig.facecolor"]` (default: `'auto'`) and `rcParams["savefig.edgecolor"]` (default: `'auto'`) now default to "auto"

This newly allowed value for `rcParams["savefig.facecolor"]` (default: `'auto'`) and `rcParams["savefig.edgecolor"]` (default: `'auto'`), as well as the *facecolor* and *edgecolor* parameters to *Figure.savefig*, means "use whatever facecolor and edgecolor the figure current has".

When using a single dataset, `Axes.hist` no longer wraps the added artist in a `silent_list`

When *Axes.hist* is called with a single dataset, it adds to the axes either a *BarContainer* object (when `histtype="bar"` or `"barstacked"`), or a *Polygon* object (when `histype="step"` or `"stepfilled"`) -- the latter being wrapped in a list-of-one-element. Previously, either artist would be wrapped in a *silent_list*. This is no longer the case: the *BarContainer* is now returned as is (this is an API breaking change if you were directly relying on the concrete `list` API; however, *BarContainer* inherits from `tuple` so most common operations remain available), and the list-of-one *Polygon* is returned as is. This makes the `repr` of the returned artist more accurate: it is now

```
<BarContainer object of of 10 artists>  # "bar", "barstacked"
[<matplotlib.patches.Polygon object at 0xdeadbeef>]  # "step", "stepfilled"
```

instead of

```
<a list of 10 Patch objects>  # "bar", "barstacked"
<a list of 1 Patch objects>  # "step", "stepfilled"
```

When *Axes.hist* is called with multiple artists, it still wraps its return value in a *silent_list*, but uses more accurate type information

```
<a list of 3 BarContainer objects>  # "bar", "barstacked"
<a list of 3 List[Polygon] objects>  # "step", "stepfilled"
```

instead of

```
<a list of 3 Lists of Patches objects>  # "bar", "barstacked"
<a list of 3 Lists of Patches objects>  # "step", "stepfilled"
```

Qt and wx backends no longer create a status bar by default

The coordinates information is now displayed in the toolbar, consistently with the other backends. This is intended to simplify embedding of Matplotlib in larger GUIs, where Matplotlib may control the toolbar but not the status bar.

`rcParams["text.hinting"]` (default: `'force_autohint'`) now supports names mapping to FreeType flags

`rcParams["text.hinting"]` (default: `'force_autohint'`) now supports the values "default", "no_autohint", "force_autohint", and "no_hinting", which directly map to the FreeType flags FT_LOAD_DEFAULT, etc. The old synonyms (respectively "either", "native", "auto", and "none") are still supported, but their use is discouraged. To get normalized values, use *backend_agg.get_hinting_flag*, which returns integer flag values.

`cbook.get_sample_data` auto-loads numpy arrays

When *cbook.get_sample_data* is used to load a npy or npz file and the keyword-only parameter `np_load` is True, the file is automatically loaded using `numpy.load`. `np_load` defaults to False for backwards compatibility, but will become True in a later release.

`get_text_width_height_descent` now checks `ismath` rather than `rcParams["text.usetex"]` (default: `False`)

... to determine whether a string should be passed to the usetex machinery or not. This allows single strings to be marked as not-usetex even when the rcParam is True.

`Axes.vlines`, `Axes.hlines`, `pyplot.vlines` and `pyplot.hlines` *colors* parameter default change

The *colors* parameter will now default to `rcParams["lines.color"]` (default: `'C0'`), while previously it defaulted to 'k'.

Aggressively autoscale clim in `ScalerMappable` classes

Previously some plotting methods would defer autoscaling until the first draw if only one of the *vmin* or *vmax* keyword arguments were passed (*`Axes.scatter`*, *`Axes.hexbin`*, *`Axes.imshow`*, *`Axes.pcolorfast`*) but would scale based on the passed data if neither was passed (independent of the *norm* keyword arguments). Other methods (*`Axes.pcolor`*, *`Axes.pcolormesh`*) always autoscaled base on the initial data.

All of the plotting methods now resolve the unset *vmin* or *vmax* at the initial call time using the data passed in.

If you were relying on exactly one of the *vmin* or *vmax* remaining unset between the time when the method is called and the first time the figure is rendered you get back the old behavior by manually setting the relevant limit back to `None`

```
cm_obj.norm.vmin = None
# or
cm_obj.norm.vmax = None
```

which will be resolved during the draw process.

16.3.2 Deprecations

`figure.add_axes()` without arguments

Calling `fig.add_axes()` with no arguments currently does nothing. This call will raise an error in the future. Adding a free-floating axes needs a position rectangle. If you want a figure-filling single axes, use `add_subplot()` instead.

`backend_wx.DEBUG_MSG`

`backend_wx.DEBUG_MSG` is deprecated. The wx backends now use regular logging.

`Colorbar.config_axis()`

`Colorbar.config_axis()` is considered internal. Its use is deprecated.

`NonUniformImage.is_grayscale` and `PcolorImage.is_grayscale`

These attributes are deprecated, for consistency with `AxesImage.is_grayscale`, which was removed back in Matplotlib 2.0.0. (Note that previously, these attributes were only available *after rendering the image*).

`den` parameter and attribute to `mpl_toolkits.axisartist.angle_helper`

For all locator classes defined in *mpl_toolkits.axisartist.angle_helper*, the `den` parameter has been renamed to `nbins`, and the `den` attribute deprecated in favor of its (pre-existing) synonym `nbins`, for consistency with locator classes defined in *matplotlib.ticker*.

`backend_pgf.LatexManager.latex_stdin_utf8`

`backend_pgf.LatexManager.latex` is now created with `encoding="utf-8"`, so its `stdin` attribute is already utf8-encoded; the `latex_stdin_utf8` attribute is thus deprecated.

Flags containing "U" passed to `cbook.to_filehandle` and `cbook.open_file_cm`

Please remove "U" from flags passed to *cbook.to_filehandle* and *cbook.open_file_cm*. This is consistent with their removal from `open` in Python 3.9.

PDF and PS character tracking internals

The `used_characters` attribute and `track_characters` and `merge_used_characters` methods of *RendererPdf*, *PdfFile*, and *RendererPS* are deprecated.

Case-insensitive capstyles and joinstyles

Please pass capstyles ("miter", "round", "bevel") and joinstyles ("butt", "round", "projecting") as lowercase.

Passing raw data to `register_cmap()`

Passing raw data via parameters *data* and *lut* to *`register_cmap()`* is deprecated. Instead, explicitly create a *`LinearSegmentedColormap`* and pass it via the *cmap* parameter: `register_cmap(cmap=LinearSegmentedColormap(name, data, lut))`.

`DateFormatter.illegal_s`

This attribute is unused and deprecated.

`widgets.TextBox.params_to_disable`

This attribute is deprecated.

Revert deprecation *min, *max keyword arguments to `set_x/y/zlim_3d()`

These keyword arguments were deprecated in 3.0, alongside with the respective parameters in `set_xlim()` / `set_ylim()`. The deprecations of the 2D versions were already reverted in in 3.1.

`cbook.local_over_kwdict`

This function is deprecated. Use *`cbook.normalize_kwargs`* instead.

Passing both singular and plural *colors*, *linewidths*, *linestyles* to `Axes.eventplot`

Passing e.g. both *linewidth* and *linewidths* will raise a TypeError in the future.

Setting `rcParams["text.latex.preamble"]` (default: `''`) or `rcParams["pdf.preamble"]` to non-strings

These rcParams should be set to string values. Support for None (meaning the empty string) and lists of strings (implicitly joined with newlines) is deprecated.

Parameters *norm* and *vmin*/*vmax* should not be used simultaneously

Passing parameters *norm* and *vmin*/*vmax* simultaneously to functions using colormapping such as `scatter()` and `imshow()` is deprecated. Inestead of `norm=LogNorm(), vmin=min_val, vmax=max_val` pass `norm=LogNorm(min_val, max_val)`. *vmin* and *vmax* should only be used without setting *norm*.

Effectless parameters of `Figure.colorbar` and `matplotlib.colorbar.Colorbar`

The *cmap* and *norm* parameters of *Figure.colorbar* and *matplotlib.colorbar.Colorbar* have no effect because they are always overridden by the mappable's colormap and norm; they are thus deprecated. Likewise, passing the *alpha*, *boundaries*, *values*, *extend*, or *filled* parameters with a *ContourSet* mappable, or the *alpha* parameter with an *Artist* mappable, is deprecated, as the mappable would likewise override them.

`args_key` and `exec_key` attributes of builtin `MovieWriters`

These attributes are deprecated.

Unused parameters

The following parameters do not have any effect and are deprecated:

- arbitrary keyword arguments to `StreamplotSet`
- parameter *quantize* of *Path.cleaned()*
- parameter *s* of *AnnotationBbox.get_fontsize()*
- parameter *label* of *Tick*

Passing *props* to `Shadow`

The parameter *props* of *Shadow* is deprecated. Use keyword arguments instead.

`Axes.update_datalim_bounds`

This method is deprecated. Use `ax.dataLim.set(Bbox.union([ax.dataLim, bounds]))` instead.

{,Symmetrical}LogScale.{,Inverted}LogTransform

LogScale.LogTransform, LogScale.InvertedLogTransform, SymmetricalScale.SymmetricalTransform and SymmetricalScale.InvertedSymmetricalTransform are deprecated. Directly access the transform classes from the *scale* module.

TexManager.cachedir, TexManager.rgba_arrayd

Use *matplotlib.get_cachedir()* instead for the former; there is no replacement for the latter.

Setting Line2D's pickradius via Line2D.set_picker

Setting a *Line2D*'s pickradius (i.e. the tolerance for pick events and containment checks) via *Line2D.set_picker* is deprecated. Use *Line2D.set_pickradius* instead.

Line2D.set_picker no longer sets the artist's custom-contain() check.

Artist.set_contains, Artist.get_contains

Setting a custom method overriding *Artist.contains* is deprecated. There is no replacement, but you may still customize pick events using *Artist.set_picker*.

Colorbar methods

The on_mappable_changed and update_bruteforce methods of *Colorbar* are deprecated; both can be replaced by calls to *update_normal*.

OldScalarFormatter, IndexFormatter and DateIndexFormatter

These formatters are deprecated. Their functionality can be implemented using e.g. *FuncFormatter*.

OldAutoLocator

This ticker is deprecated.

required, *forbidden* and *allowed* parameters of `cbook.normalize_kwargs`

These parameters are deprecated.

The `TTFPATH` and `AFMPATH` environment variables

Support for the (undocumented) `TTFPATH` and `AFMPATH` environment variables is deprecated. Additional fonts may be registered using `matplotlib.font_manager.fontManager.addfont()`.

`matplotlib.compat`

This module is deprecated.

`matplotlib.backends.qt_editor.formsubplottool`

This module is deprecated. Use `matplotlib.backends.backend_qt5.SubplotToolQt` instead.

AVConv animation writer deprecated

The `AVConvBase`, `AVConvWriter` and `AVConvFileWriter` classes, and the associated `animation.avconv_path` and `animation.avconv_args` rcParams are deprecated.

Debian 8 (2015, EOL 06/2020) and Ubuntu 14.04 (EOL 04/2019) were the last versions of Debian and Ubuntu to ship avconv. It remains possible to force the use of avconv by using the ffmpeg-based writers with `rcParams["animation.ffmpeg_path"]` (default: `'ffmpeg'`) set to "avconv".

log/symlog scale base, ticks, and nonpos specification

semilogx, *semilogy*, *loglog*, *LogScale*, and *SymmetricalLogScale* used to take keyword arguments that depends on the axis orientation ("basex" vs "basey", "subsx" vs "subsy", "nonposx" vs "nonposy"); these parameter names are now deprecated in favor of "base", "subs", "nonpositive". This deprecation also affects e.g. `ax.set_yscale("log", basey=...)` which must now be spelled `ax.set_yscale("log", base=...)`.

The change from "nonpos" to "nonpositive" also affects *LogTransform*, *InvertedLogTransform*, *SymmetricalLogTransform*, etc.

To use *different* bases for the x-axis and y-axis of a *loglog* plot, use e.g. `ax.set_xscale("log", base=10); ax.set_yscale("log", base=2)`.

DraggableBase.artist_picker

This method is deprecated. If you previously reimplemented it in a subclass, set the artist's picker instead with *Artist.set_picker*.

clear_temp parameter and attribute of FileMovieWriter

The *clear_temp* parameter and attribute of *FileMovieWriter* is deprecated. In the future, files placed in a temporary directory (using `frame_prefix=None`, the default) will be cleared; files placed elsewhere will not.

Deprecated rcParams validators

The following validators, defined in *rcsetup*, are deprecated: `validate_fontset`, `validate_mathtext_default`, `validate_alignment`, `validate_svg_fontset`, `validate_pgf_texsystem`, `validate_movie_frame_fmt`, `validate_axis_locator`, `validate_movie_html_fmt`, `validate_grid_axis`, `validate_axes_titlelocation`, `validate_toolbar`, `validate_ps_papersize`, `validate_legend_loc`, `validate_bool_maybe_none`, `validate_hinting`, `validate_movie_writers`, `validate_webagg_address`, `validate_nseq_float`, `validate_nseq_int`. To test whether an rcParam value would be acceptable, one can test e.g. `rc = RcParams(); rc[k] = v` raises an exception.

Stricter rcParam validation

`rcParams["axes.axisbelow"]` (default: `'line'`) currently normalizes all strings starting with "line" (case-insensitive) to the option "line". This is deprecated; in a future version only the exact string "line" (case-sensitive) will be supported.

add_subplot() validates its inputs

In particular, for `add_subplot(rows, cols, index)`, all parameters must be integral. Previously strings and floats were accepted and converted to int. This will now emit a deprecation warning.

Toggling axes navigation from the keyboard using "a" and digit keys

Axes navigation can still be toggled programmatically using *Axes.set_navigate*.

The following related APIs are also deprecated: `backend_tools.ToolEnableAllNavigation`, `backend_tools.ToolEnableNavigation`, and `rcParams["keymap.all_axes"]`.

`matplotlib.test(recursionlimit=...)`

The *recursionlimit* parameter of `matplotlib.test` is deprecated.

mathtext glues

The *copy* parameter of `mathtext.Glue` is deprecated (the underlying glue spec is now immutable). `mathtext.GlueSpec` is deprecated.

Signatures of `Artist.draw` and `Axes.draw`

The *inframe* parameter to *Axes.draw* is deprecated. Use *Axes.redraw_in_frame* instead.

Not passing the *renderer* parameter to *Axes.draw* is deprecated. Use `axes.draw_artist(axes)` instead.

These changes make the signature of the `draw` (`artist.draw(renderer)`) method consistent across all artists; thus, additional parameters to *Artist.draw* are deprecated.

`DraggableBase.on_motion_blit`

This method is deprecated. *DraggableBase.on_motion* now handles both the blitting and the non-blitting cases.

Passing the dash offset as None

Fine control of dash patterns can be achieved by passing an `(offset, (on-length, off-length, on-length, off-length, ...))` pair as the linestyle property of *Line2D* and *LineCollection*. Previously, certain APIs would accept `offset = None` as a synonym for `offset = 0`, but this was never universally implemented, e.g. for vector output. Support for `offset = None` is deprecated, set the offset to 0 instead.

RendererCairo.fontweights, RendererCairo.fontangles

... are deprecated.

autofmt_xdate(which=None)

This is deprecated, use its more explicit synonym, `which="major"`, instead.

JPEG options

The *quality*, *optimize*, and *progressive* keyword arguments to *savefig*, which were only used when saving to JPEG, are deprecated. `rcParams["savefig.jpeg_quality"]` (default: `95`) is likewise deprecated.

Such options should now be directly passed to Pillow using `savefig(..., pil_kwargs={"quality": ..., "optimize": ..., "progressive": ...})`.

dviread.Encoding

This class was (mostly) broken and is deprecated.

Axis and Locator pan and zoom

The unused `pan` and `zoom` methods of *Axis* and *Locator* are deprecated. Panning and zooming are now implemented using the `start_pan`, `drag_pan`, and `end_pan` methods of *Axes*.

Passing None to various Axes subclass factories

Support for passing `None` as base class to *axes.subplot_class_factory*, `axes_grid1.parasite_axes.host_axes_class_factory`, `axes_grid1.parasite_axes.host_subplot_class_factory`, `axes_grid1.parasite_axes.parasite_axes_class_factory`, and `axes_grid1.parasite_axes.parasite_axes_auxtrans_class_factory` is deprecated. Explicitly pass the correct base `Axes` class instead.

axes_rgb

In *mpl_toolkits.axes_grid1.axes_rgb*, `imshow_rgb` is deprecated (use `ax.imshow(np.dstack([r, g, b]))` instead); `RGBAxesBase` is deprecated (use `RGBAxes` instead); `RGBAxes.add_RGB_to_figure` is deprecated (it was an internal helper).

Substitution.from_params

This method is deprecated. If needed, directly assign to the `params` attribute of the Substitution object.

PGF backend cleanups

The *dummy* parameter of *RendererPgf* is deprecated.

GraphicsContextPgf is deprecated (use *GraphicsContextBase* instead).

set_factor method of mpl_toolkits.axisartist locators

The `set_factor` method of *mpl_toolkits.axisartist* locators (which are different from "standard" Matplotlib tick locators) is deprecated.

widgets.SubplotTool callbacks and axes

The `funcleft`, `funcright`, `funcbottom`, `functop`, `funcwspace`, and `funchspace` methods of *widgets.SubplotTool* are deprecated.

The `axleft`, `axright`, `axbottom`, `axtop`, `axwspace`, and `axhspace` attributes of *widgets.SubplotTool* are deprecated. Access the `ax` attribute of the corresponding slider, if needed.

mathtext Glue helper classes

The `Fil`, `Fill`, `Filll`, `NegFil`, `NegFill`, `NegFilll`, and `SsGlue` classes in the *matplotlib.mathtext* module are deprecated. As an alternative, directly construct glue instances with `Glue("fil")`, etc.

NavigationToolbar2._init_toolbar

Overriding this method to initialize third-party toolbars is deprecated. Instead, the toolbar should be initialized in the `__init__` method of the subclass (which should call the base-class' `__init__` as appropriate). To keep back-compatibility with earlier versions of Matplotlib (which *required* `_init_toolbar` to be overridden), a fully empty implementation (`def _init_toolbar(self): pass`) may be kept and will not trigger the deprecation warning.

NavigationToolbar2QT.parent and .basedir

These attributes are deprecated. In order to access the parent window, use `toolbar.canvas.parent()`. Once the deprecation period is elapsed, it will also be accessible as `toolbar.parent()`. The base directory to the icons is `os.path.join(mpl.get_data_path(), "images")`.

NavigationToolbar2QT.ctx

This attribute is deprecated.

NavigationToolbar2Wx attributes

The `prevZoomRect`, `retinaFix`, `savedRetinaImage`, `wxoverlay`, `zoomAxes`, `zoomStartX`, and `zoomStartY` attributes are deprecated.

NavigationToolbar2.press and .release

These methods were called when pressing or releasing a mouse button, but *only* when an interactive pan or zoom was occurring (contrary to what the docs stated). They are deprecated; if you write a backend which needs to customize such events, please directly override `press_pan`/`press_zoom`/`release_pan`/`release_zoom` instead.

FigureCanvasGTK3._renderer_init

Overriding this method to initialize renderers for GTK3 canvases is deprecated. Instead, the renderer should be initialized in the `__init__` method of the subclass (which should call the base-class' `__init__` as appropriate). To keep back-compatibility with earlier versions of Matplotlib (which *required* `_renderer_init` to be overridden), a fully empty implementation (`def _renderer_init(self): pass`) may be kept and will not trigger the deprecation warning.

Path helpers in `bezier`

`bezier.make_path_regular` is deprecated. Use `Path.cleaned()` (or `Path.cleaned(curves=True)`, etc.) instead (but note that these methods add a `STOP` code at the end of the path).

`bezier.concatenate_paths` is deprecated. Use `Path.make_compound_path()` instead.

`animation.html_args` rcParam

The unused `animation.html_args` rcParam and `animation.HTMLWriter.args_key` attribute are deprecated.

`text.latex.preview` rcParam

This rcParam, which controlled the use of the preview.sty LaTeX package to align TeX string baselines, is deprecated, as Matplotlib's own dvi parser now computes baselines just as well as preview.sty.

`SubplotSpec.get_rows_columns`

This method is deprecated. Use the `GridSpec.nrows`, `GridSpec.ncols`, `SubplotSpec.rowspan`, and `SubplotSpec.colspan` properties instead.

Qt4-based backends

The qt4agg and qt4cairo backends are deprecated. Qt4 has reached its end-of-life in 2015 and there are no releases for recent versions of Python. Please consider switching to Qt5.

fontdict and *minor* parameters of `Axes.set_xticklabels` and `Axes.set_yticklabels` will become keyword-only

All parameters of `Figure.subplots` except *nrows* and *ncols* will become keyword-only

This avoids typing e.g. `subplots(1, 1, 1)` when meaning `subplot(1, 1, 1)`, but actually getting `subplots(1, 1, sharex=1)`.

RendererWx.get_gc

This method is deprecated. Access the `gc` attribute directly instead.

add_all parameter in axes_grid

The *add_all* parameter of *axes_grid1.axes_grid.Grid*, *axes_grid1.axes_grid.ImageGrid*, *axes_grid1.axes_rgb.make_rgb_axes* and *axes_grid1.axes_rgb.RGBAxes* is deprecated. Axes are now always added to the parent figure, though they can be later removed with `ax.remove()`.

BboxBase.inverse_transformed

`.BboxBase.inverse_transformed` is deprecated (call *BboxBase.transformed* on the *inverted()* transform instead).

orientation of eventplot() and EventCollection

Setting the *orientation* of an `eventplot()` or *EventCollection* to "none" or None is deprecated; set it to "horizontal" instead. Moreover, the two orientations ("horizontal" and "vertical") will become case-sensitive in the future.

minor kwarg to Axis.get_ticklocs will become keyword-only

Passing this argument positionally is deprecated.

Case-insensitive properties

Normalization of upper or mixed-case property names to lowercase in *Artist.set* and *Artist.update* is deprecated. In the future, property names will be passed as is, allowing one to pass names such as *patchA* or *UVC*.

ContourSet.ax, Quiver.ax

These attributes are deprecated in favor of `ContourSet.axes` and `Quiver.axes`, for consistency with other artists.

`Locator.refresh()` and associated methods

`Locator.refresh()` is deprecated. This method was called at certain places to let locators update their internal state, typically based on the axis limits. Locators should now always consult the axis limits when called, if needed.

The associated helper methods `NavigationToolbar2.draw()` and `ToolViewsPositions.refresh_locators()` are deprecated, and should be replaced by calls to `draw_idle()` on the corresponding canvas.

`ScalarMappable` checkers

The `add_checker` and `check_update` methods and `update_dict` attribute of *`ScalarMappable`* are deprecated.

`pyplot.tight_layout` and `ColorbarBase` parameters will become keyword-only

All parameters of *`pyplot.tight_layout`* and all parameters of `ColorbarBase` except for the first (*ax*) will become keyword-only, consistently with *`Figure.tight_layout`* and `Colorbar`, respectively.

`Axes.pie` radius and startangle

Passing `None` as either the `radius` or `startangle` of an *`Axes.pie`* is deprecated; use the explicit defaults of 1 and 0, respectively, instead.

`AxisArtist.dpi_transform`

... is deprecated. Scale `Figure.dpi_scale_trans` by 1/72 to achieve the same effect.

`offset_position` property of `Collection`

The `offset_position` property of *`Collection`* is deprecated. In the future, *`Collection`*s will always behave as if `offset_position` is set to "screen" (the default).

Support for passing `offset_position="data"` to the `draw_path_collection` of all renderer classes is deprecated.

`transforms.AffineDeltaTransform` can be used as a replacement. This API is experimental and may change in the future.

testing.compare.make_external_conversion_command

... is deprecated.

epoch2num and num2epoch are deprecated

These are unused and can be easily reproduced by other date tools. *get_epoch* will return Matplotlib's epoch.

axes_grid1.CbarAxes attributes

The `cbid` and `locator` attribute are deprecated. Use `mappable.colorbar_cid` and `colorbar.locator`, as for standard colorbars.

qt_compat.is_pyqt5

This function is deprecated in prevision of the future release of PyQt6. The Qt version can be checked using `QtCore.qVersion()`.

Reordering of parameters by Artist.set

In a future version, `Artist.set` will apply artist properties in the order in which they are given. This only affects the interaction between the *color*, *edgecolor*, *facecolor*, and, for *Collection*s, *alpha* properties: the *color* property now needs to be passed first in order not to override the other properties. This is consistent with e.g. *Artist.update*, which did not reorder the properties passed to it.

Passing multiple keys as a single comma-separated string or multiple arguments to ToolManager.update_keymap

This is deprecated; pass keys as a list of strings instead.

Statusbar classes and attributes

The `statusbar` attribute of *FigureManagerBase*, *StatusbarBase* and all its subclasses, and `StatusBarWx`, are deprecated, as messages are now displayed in the toolbar instead.

`ismath` parameter of `draw_tex`

The `ismath` parameter of the `draw_tex` method of all renderer classes is deprecated (as a call to `draw_tex` -- not to be confused with `draw_text`! -- means that the entire string should be passed to the `usetex` machinery anyways). Likewise, the text machinery will no longer pass the `ismath` parameter when calling `draw_tex` (this should only matter for backend implementers).

Passing `ismath="TeX!"` to *`RendererAgg.get_text_width_height_descent`* is deprecated. Pass `ismath="TeX"` instead, consistently with other low-level APIs which support the values True, False, and "TeX" for `ismath`.

`matplotlib.ttconv`

This module is deprecated.

Stricter PDF metadata keys in PGF

Saving metadata in PDF with the PGF backend currently normalizes all keys to lowercase, unlike the PDF backend, which only accepts the canonical case. This is deprecated; in a future version, only the canonically cased keys listed in the PDF specification (and the *`PdfPages`* documentation) will be accepted.

Qt modifier keys

The `MODIFIER_KEYS`, `SUPER`, `ALT`, `CTRL`, and `SHIFT` global variables of the `matplotlib.backends.backend_qt4agg`, `matplotlib.backends.backend_qt4cairo`, `matplotlib.backends.backend_qt5agg` and `matplotlib.backends.backend_qt5cairo` modules are deprecated.

`TexManager`

The `TexManager.serif`, `TexManager.sans_serif`, `TexManager.cursive` and `TexManager.monospace` attributes are deprecated.

16.3.3 Removals

The following deprecated APIs have been removed:

Modules

- `backends.qt_editor.formlayout` (use the formlayout module available on PyPI instead).

Classes, methods and attributes

- `artist.Artist.aname` property (no replacement)
- `axis.Axis.iter_ticks` (no replacement)
- Support for custom backends that do not provide a `backend_bases.GraphicsContextBase.set_hatch_color` method
- `backend_bases.RendererBase.strip_math()` (use `cbook.strip_math()` instead)
- `backend_wx.debug_on_error()` (no replacement)
- `backend_wx.raise_msg_to_str()` (no replacement)
- `backend_wx.fake_stderr` (no replacement)
- `backend_wx.MenuButtonWx` (no replacement)
- `backend_wx.PrintoutWx` (no replacement)
- `_backend_tk.NavigationToolbar2Tk.set_active()` (no replacement)
- `backend_ps.PsBackendHelper.gs_exe` property (no replacement)
- `backend_ps.PsBackendHelper.gs_version` property (no replacement)
- `backend_ps.PsBackendHelper.supports_ps2write` property (no replacement)
- `backend_ps.RendererPS.afmfontd` property (no replacement)
- `backend_ps.GraphicsContextPS.shouldstroke` property (no replacement)
- `backend_gtk3.FileChooserDialog` (no replacement)
- `backend_gtk3.SaveFigureGTK3.get_filechooser()` (no replacement)
- `backend_gtk3.NavigationToolbar2GTK3.get_filechooser()` (no replacement)
- `backend_gtk3cairo.FigureManagerGTK3Cairo` (use `backend_gtk3.FigureManagerGTK3` instead)
- `backend_pdf.RendererPdf.afm_font_cache` property (no replacement)
- `backend_pgf.LatexManagerFactory` (no replacement)
- `backend_qt5.NavigationToolbar2QT.buttons` property (no replacement)
- `backend_qt5.NavigationToolbar2QT.adj_window` property (no replacement)
- `bezier.find_r_to_boundary_of_closedpath()` (no replacement)
- `cbook.dedent()` (use `inspect.cleandoc` instead)

- `cbook.get_label()` (no replacement)
- `cbook.is_hashable()` (use `isinstance(..., collections.abc.Hashable)` instead)
- `cbook.iterable()` (use `numpy.iterable()` instead)
- `cbook.safezip()` (no replacement)
- `colorbar.ColorbarBase.get_cmap` (use `ScalarMappable.get_cmap` instead)
- `colorbar.ColorbarBase.set_cmap` (use `ScalarMappable.set_cmap` instead)
- `colorbar.ColorbarBase.get_clim` (use `ScalarMappable.get_clim` instead)
- `colorbar.ColorbarBase.set_clim` (use `ScalarMappable.set_clim` instead)
- `colorbar.ColorbarBase.set_norm` (use `ScalarMappable.set_norm` instead)
- `dates.seconds()` (no replacement)
- `dates.minutes()` (no replacement)
- `dates.hours()` (no replacement)
- `dates.weeks()` (no replacement)
- `dates.strpdate2num` and `dates.bytespdate2num` (use `time.strptime` or `dateutil.parser.parse` or *`dates.datestr2num`* instead)
- `docstring.Appender` (no replacement)
- `docstring.dedent()` (use `inspect.getdoc` instead)
- `docstring.copy_dedent()` (use `docstring.copy()` and `inspect.getdoc` instead)
- `font_manager.OSXInstalledFonts()` (no replacement)
- `image.BboxImage.interp_at_native` property (no replacement)
- `lines.Line2D.verticalOffset` property (no replacement)
- `matplotlib.checkdep_dvipng` (no replacement)
- `matplotlib.checkdep_ghostscript` (no replacement)
- `matplotlib.checkdep_pdftops` (no replacement)
- `matplotlib.checkdep_inkscape` (no replacement)
- `matplotlib.get_py2exe_datafiles` (no replacement)
- `matplotlib.tk_window_focus` (use `rcParams['tk.window_focus']` instead)
- `mlab.demean()` (use `mlab.detrend_mean()` instead)
- `path.get_paths_extents()` (use `path.get_path_collection_extents()` instead)
- `path.Path.has_nonfinite()` (use `not np.isfinite(self.vertices).all()` instead)
- `projections.process_projection_requirements()` (no replacement)
- `pyplot.plotfile()` (Instead, load the data using `pandas.read_csv` or `numpy.loadtxt` or similar and use regular pyplot functions to plot the loaded data.)

- `quiver.Quiver.color()` (use `Quiver.get_facecolor()` instead)
- `quiver.Quiver.keyvec` property (no replacement)
- `quiver.Quiver.keytext` property (no replacement)
- `rcsetup.validate_qt4()` (no replacement)
- `rcsetup.validate_qt5()` (no replacement)
- `rcsetup.validate_verbose()` (no replacement)
- `rcsetup.ValidateInterval` (no replacement)
- `scale.LogTransformBase` (use `scale.LogTransform` instead)
- `scale.InvertedLogTransformBase` (use `scale.InvertedLogTransform` instead)
- `scale.Log10Transform` (use `scale.LogTransform` instead)
- `scale.InvertedLog10Transform` (use `scale.InvertedLogTransform` instead)
- `scale.Log2Transform` (use `scale.LogTransform` instead)
- `scale.InvertedLog2Transform` (use `scale.InvertedLogTransform` instead)
- `scale.NaturalLogTransform` (use `scale.LogTransform` instead)
- `scale.InvertedNaturalLogTransform` (use `scale.InvertedLogTransform` instead)
- `scale.get_scale_docs()` (no replacement)
- `sphinxext.plot_directive.plot_directive()` (use the class `PlotDirective` instead)
- `sphinxext.mathmpl.math_directive()` (use the class `MathDirective` instead)
- `spines.Spine.is_frame_like()` (no replacement)
- `testing.decorators.switch_backend()` (use `@pytest.mark.backend` decorator instead)
- `text.Text.is_math_text()` (use `cbook.is_math_text()` instead)
- `text.TextWithDash()` (use `text.Annotation` instead)
- `textpath.TextPath.is_math_text()` (use `cbook.is_math_text()` instead)
- `textpath.TextPath.text_get_vertices_codes()` (use `textpath.text_to_path.get_text_path()` instead)
- `textpath.TextToPath.glyph_to_path()` (use `font.get_path()` and manual translation of the vertices instead)
- `ticker.OldScalarFormatter.pprint_val()` (no replacement)
- `ticker.ScalarFormatter.pprint_val()` (no replacement)
- `ticker.LogFormatter.pprint_val()` (no replacement)
- `ticker.decade_down()` (no replacement)
- `ticker.decade_up()` (no replacement)

- `Tick` properties `gridOn`, `tick1On`, `tick2On`, `label1On`, `label2On` (use `set_visible()` / `get_visible()` on `Tick.gridline`, `Tick.tick1line`, `Tick.tick2line`, `Tick.label1`, `Tick.label2` instead)
- `widgets.SpanSelector.buttonDown` property (no replacement)
- `mplot3d.proj3d.line2d()` (no replacement)
- `mplot3d.proj3d.line2d_dist()` (no replacement)
- `mplot3d.proj3d.line2d_seg_dist()` (no replacement)
- `mplot3d.proj3d.mod()` (use `numpy.linalg.norm` instead)
- `mplot3d.proj3d.proj_transform_vec()` (no replacement)
- `mplot3d.proj3d.proj_transform_vec_clip()` (no replacement)
- `mplot3d.proj3d.vec_pad_ones()` (no replacement)
- `mplot3d.proj3d.proj_trans_clip_points()` (no replacement)
- `mplot3d.art3d.norm_angle()` (no replacement)
- `mplot3d.art3d.norm_text_angle()` (no replacement)
- `mplot3d.art3d.path_to_3d_segment()` (no replacement)
- `mplot3d.art3d.paths_to_3d_segments()` (no replacement)
- `mplot3d.art3d.path_to_3d_segment_with_codes()` (no replacement)
- `mplot3d.art3d.paths_to_3d_segments_with_codes()` (no replacement)
- `mplot3d.art3d.get_patch_verts()` (no replacement)
- `mplot3d.art3d.get_colors()` (no replacement)
- `mplot3d.art3d.zalpha()` (no replacement)
- `mplot3d.axis3d.get_flip_min_max()` (no replacement)
- `mplot3d.axis3d.Axis.get_tick_positions()` (no replacement)
- `axisartist.axis_artist.UnimplementedException` (no replacement)
- `axisartist.axislines.SimpleChainedObjects` (use `axis_grid1.mpl_axes.SimpleChainedObjects` instead)
- `axisartist.axislines.Axes.AxisDict` (use `axis_grid1.mpl_axes.Axes.AxisDict` instead)

Arguments

- `Axes.text()` / `pyplot.text()` do not support the parameter `withdash` anymore. Use `Axes.annotate()` and `pyplot.annotate()` instead.
- The first parameter of *matplotlib.use* has been renamed from `arg` to `backend` (only relevant if you pass by keyword).
- The parameter `warn` of *matplotlib.use* has been removed. A failure to switch the backend will now always raise an `ImportError` if `force` is set; catch that error if necessary.
- All parameters of *matplotlib.use* except the first one are now keyword-only.
- The unused parameters `shape` and `imlim` of *imshow()* are now removed. All parameters beyond `extent` are now keyword-only.
- The unused parameter `interp_at_native` of *BboxImage* has been removed.
- The parameter `usetex` of *TextToPath.get_text_path* has been removed. Use `ismath='TeX'` instead.
- The parameter `block` of `show()` is now keyword-only, and arbitrary arguments or keyword arguments are no longer accepted.
- The parameter `frameon` of *Figure.savefig* has been removed. Use `facecolor="none"` to get a transparent background.
- Passing a `wx.EvtHandler` as the first argument to `backend_wx.TimerWx` is not supported anymore; the signature of `TimerWx` is now consistent with *TimerBase*.
- The `manage_xticks` parameter of *boxplot* and *bxp* has been renamed to `manage_ticks`.
- The `normed` parameter of *hist2d* has been renamed to `density`.
- The `s` parameter of *Annotation* has been renamed to `text`.
- For all functions in *bezier* that supported a `tolerence` parameter, this parameter has been renamed to `tolerance`.
- `axis("normal")` is not supported anymore. Use the equivalent `axis("auto")` instead.
- `axis()` does not accept arbitrary keyword arguments anymore.
- `Axis.set_ticklabels()` does not accept arbitrary positional arguments other than `ticklabels`.
- `mpl_toolkits.mplot3d.art3d.Poly3DCollection.set_zsort` does not accept the value `True` anymore. Pass the equivalent value 'average' instead.
- *AnchoredText* no longer accepts `horizontalalignment` or `verticalalignment` keyword arguments.
- *ConnectionPatch* no longer accepts the `arrow_transmuter` and `connector` keyword arguments, which did nothing since 3.0.

- *FancyArrowPatch* no longer accepts the `arrow_transmuter` and `connector` keyword arguments, which did nothing since 3.0.
- *TextPath* no longer accepts arbitrary positional or keyword arguments.
- *MaxNLocator.set_params()* no longer accepts arbitrary keyword arguments.
- *pie* no longer accepts and squeezes non-1D inputs; pass 1D input to the `x` argument.
- Passing (n, 1)-shaped error arrays to *Axes.errorbar()* is no longer supported; pass a 1D array instead.

rcParams

- The `text.latex.unicode` rcParam has been removed, with no replacement. Matplotlib now always supports unicode in usetex.
- The `savefig.frameon` rcParam has been removed. Set `rcParams["savefig.facecolor"]` (default: `'auto'`) to "none" to get a transparent background.
- The `pgf.debug`, `verbose.fileo` and `verbose.verbose.level` rcParams, which had no effect, have been removed.
- Support for setting `rcParams["mathtext.default"]` (default: `'it'`) to "circled" has been removed.

Environment variables

- `MATPLOTLIBDATA` (no replacement).

mathtext

- The `\stackrel` command (which behaved differently from its LaTeX version) has been removed. Use `\genfrac` instead.
- The `\mathcircled` command has been removed. Directly use Unicode characters, such as `'\N{CIRCLED LATIN CAPITAL LETTER A}'`, instead.

16.3.4 Development changes

Matplotlib now requires numpy>=1.15

Matplotlib now uses Pillow to save and read pngs

The builtin png encoder and decoder has been removed, and Pillow is now a dependency. Note that when reading 16-bit RGB(A) images, Pillow truncates them to 8-bit precision, whereas the old builtin decoder kept the full precision.

The deprecated wx backend (not wxagg!) now always uses wx's builtin jpeg and tiff support rather than relying on Pillow for writing these formats; this behavior is consistent with wx's png output.

See also the *API Changes*.

CHAPTER

SEVENTEEN

USAGE PATTERNS

Below we describe several common approaches to plotting with Matplotlib.

17.1 The pyplot API

matplotlib.pyplot is a collection of command style functions that make Matplotlib work like MATLAB. Each pyplot function makes some change to a figure: e.g., creates a figure, creates a plotting area in a figure, plots some lines in a plotting area, decorates the plot with labels, etc.

pyplot is mainly intended for interactive plots and simple cases of programmatic plot generation.

Further reading:

- The *matplotlib.pyplot* function reference
- *Pyplot tutorial*
- Pyplot examples

17.2 The object-oriented API

At its core, Matplotlib is object-oriented. We recommend directly working with the objects, if you need more control and customization of your plots.

In many cases you will create a *Figure* and one or more *Axes* using *pyplot.subplots* and from then on only work on these objects. However, it's also possible to create *Figure*s explicitly (e.g. when including them in GUI applications).

Further reading:

- *matplotlib.axes.Axes* and *matplotlib.figure.Figure* for an overview of plotting functions.
- Most of the examples use the object-oriented approach (except for the pyplot section)

17.3 The pylab API (disapproved)

Warning: Since heavily importing into the global namespace may result in unexpected behavior, the use of pylab is strongly discouraged. Use *matplotlib.pyplot* instead.

pylab is a module that includes *matplotlib.pyplot*, numpy, numpy.fft, numpy.linalg, numpy.random, and some additional functions, all within a single namespace. Its original purpose was to mimic a MATLAB-like way of working by importing all functions into the global namespace. This is considered bad style nowadays.

CHAPTER

EIGHTEEN

MODULES

Matplotlib consists of the following submodules:

18.1 matplotlib

18.1.1 Backend management

matplotlib.use(*backend*, *, *force=True*)

Select the backend used for rendering and GUI integration.

Parameters

backend

[str] The backend to switch to. This can either be one of the standard backend names, which are case-insensitive:

- interactive backends: GTK3Agg, GTK3Cairo, MacOSX, nbAgg, Qt4Agg, Qt4Cairo, Qt5Agg, Qt5Cairo, TkAgg, TkCairo, WebAgg, WX, WXAgg, WXCairo
- non-interactive backends: agg, cairo, pdf, pgf, ps, svg, template

or a string of the form: `module://my.module.name`.

force

[bool, default: True] If True (the default), raise an `ImportError` if the backend cannot be set up (either because it fails to import, or because an incompatible GUI interactive framework is already running); if False, ignore the failure.

See also:

Backends

matplotlib.get_backend

matplotlib.**get_backend**()
Return the name of the current backend.

See also:

matplotlib.use

matplotlib.**interactive**(*b*)
Set whether to redraw after every plotting command (e.g. *pyplot.xlabel*).

matplotlib.**is_interactive**()
Return whether to redraw after every plotting command.

18.1.2 Default values and styling

matplotlib.**rcParams**
An instance of *RcParams* for handling default Matplotlib values.

class matplotlib.**RcParams**(**args*, ***kwargs*)
A dictionary object including validation.

Validating functions are defined and associated with rc parameters in *matplotlib.rcsetup*.

See also:

The matplotlibrc file

find_all(*self*, *pattern*)
Return the subset of this RcParams dictionary whose keys match, using `re.search()`, the given `pattern`.

Note: Changes to the returned dictionary are *not* propagated to the parent RcParams dictionary.

---

matplotlib.**rc_context**(*rc=None*, *fname=None*)
Return a context manager for temporarily changing rcParams.

Parameters

rc

[dict] The rcParams to temporarily set.

fname

[str or path-like] A file with Matplotlib rc settings. If both *fname* and *rc* are given, settings from *rc* take precedence.

See also:

The matplotlibrc file

Examples

Passing explicit values via a dict:

```
with mpl.rc_context({'interactive': False}):
    fig, ax = plt.subplots()
    ax.plot(range(3), range(3))
    fig.savefig('example.png')
    plt.close(fig)
```

Loading settings from a file:

```
with mpl.rc_context(fname='print.rc'):
    plt.plot(x, y)  # uses 'print.rc'
```

matplotlib.**rc**(*group*, ***kwargs*)

Set the current *rcParams*. *group* is the grouping for the rc, e.g., for `lines.linewidth` the group is `lines`, for `axes.facecolor`, the group is `axes`, and so on. Group may also be a list or tuple of group names, e.g., (*xtick*, *ytick*). *kwargs* is a dictionary attribute name/value pairs, e.g.,:

```
rc('lines', linewidth=2, color='r')
```

sets the current *rcParams* and is equivalent to:

```
rcParams['lines.linewidth'] = 2
rcParams['lines.color'] = 'r'
```

The following aliases are available to save typing for interactive users:

Alias	Property
'lw'	'linewidth'
'ls'	'linestyle'
'c'	'color'
'fc'	'facecolor'
'ec'	'edgecolor'
'mew'	'markeredgewidth'
'aa'	'antialiased'

Thus you could abbreviate the above call as:

```
rc('lines', lw=2, c='r')
```

Note you can use python's kwargs dictionary facility to store dictionaries of default parameters. e.g., you can customize the font rc as follows:

```
font = {'family' : 'monospace',
        'weight' : 'bold',
        'size'   : 'larger'}
rc('font', **font)  # pass in the font dict as kwargs
```

This enables you to easily switch between several configurations. Use `matplotlib.style.use('default')` or *rcdefaults()* to restore the default *rcParams* after changes.

Notes

Similar functionality is available by using the normal dict interface, i.e. `rcParams.update({"lines.linewidth": 2, ...})` (but `rcParams.update` does not support abbreviations or grouping).

matplotlib.**rcdefaults**()

Restore the *rcParams* from Matplotlib's internal default style.

Style-blacklisted *rcParams* (defined in `matplotlib.style.core.STYLE_BLACKLIST`) are not updated.

See also:

matplotlib.rc_file_defaults

Restore the *rcParams* from the rc file originally loaded by Matplotlib.

matplotlib.style.use

Use a specific style file. Call `style.use('default')` to restore the default style.

matplotlib.**rc_file_defaults**()

Restore the *rcParams* from the original rc file loaded by Matplotlib.

Style-blacklisted *rcParams* (defined in `matplotlib.style.core.STYLE_BLACKLIST`) are not updated.

matplotlib.**rc_file**(*fname*, ***, *use_default_template=True*)

Update *rcParams* from file.

Style-blacklisted *rcParams* (defined in `matplotlib.style.core.STYLE_BLACKLIST`) are not updated.

Parameters

fname

[str or path-like] A file with Matplotlib rc settings.

use_default_template

[bool] If True, initialize with default parameters before updating with those in the given file. If False, the current configuration persists and only the parameters specified in the file are updated.

matplotlib.**rc_params**(*fail_on_error=False*)

Construct a *RcParams* instance from the default Matplotlib rc file.

matplotlib.**rc_params_from_file**(*fname*, *fail_on_error=False*, *use_default_template=True*)

Construct a *RcParams* from file *fname*.

Parameters

fname

[str or path-like] A file with Matplotlib rc settings.

fail_on_error

[bool] If True, raise an error when the parser fails to convert a parameter.

use_default_template

[bool] If True, initialize with default parameters before updating with those in the given file. If False, the configuration class only contains the parameters specified in the file. (Useful for updating dicts.)

matplotlib.**get_configdir**()

Return the string path of the the configuration directory.

The directory is chosen as follows:

1. If the MPLCONFIGDIR environment variable is supplied, choose that.
2. On Linux, follow the XDG specification and look first in `$XDG_CONFIG_HOME`, if defined, or `$HOME/.config`. On other platforms, choose `$HOME/.matplotlib`.
3. If the chosen directory exists and is writable, use that as the configuration directory.
4. Else, create a temporary directory, and use it as the configuration directory.

matplotlib.**matplotlib_fname**()

Get the location of the config file.

The file location is determined in the following order

- `$PWD/matplotlibrc`
- `$MATPLOTLIBRC` if it is not a directory
- `$MATPLOTLIBRC/matplotlibrc`
- `$MPLCONFIGDIR/matplotlibrc`

- **On Linux,**
  - `$XDG_CONFIG_HOME/matplotlib/matplotlibrc` (if `$XDG_CONFIG_HOME` is defined)
  - or `$HOME/.config/matplotlib/matplotlibrc` (if `$XDG_CONFIG_HOME` is not defined)
- On other platforms, - `$HOME/.matplotlib/matplotlibrc` if `$HOME` is defined
- Lastly, it looks in `$MATPLOTLIBDATA/matplotlibrc`, which should always exist.

matplotlib.**get_data_path**(*, *_from_rc=None*)

Return the path to Matplotlib data.

18.1.3 Logging

matplotlib.**set_loglevel**(*level*)

Set Matplotlib's root logger and root logger handler level, creating the handler if it does not exist yet.

Typically, one should call `set_loglevel("info")` or `set_loglevel("debug")` to get additional debugging information.

Parameters

level

[{"notset", "debug", "info", "warning", "error", "critical"}] The log level of the handler.

Notes

The first time this function is called, an additional handler is attached to Matplotlib's root handler; this handler is reused every time and this function simply manipulates the logger and handler's level.

18.1.4 Miscellaneous

matplotlib.**get_cachedir**()

Return the string path of the cache directory.

The procedure used to find the directory is the same as for _get_config_dir, except using `$XDG_CACHE_HOME`/`$HOME/.cache` instead.

18.2 matplotlib.afm

A python interface to Adobe Font Metrics Files.

Although a number of other python implementations exist, and may be more complete than this, it was decided not to go with them because they were either:

1) copyrighted or used a non-BSD compatible license
2) had too many dependencies and a free standing lib was needed
3) did more than needed and it was easier to write afresh rather than figure out how to get just what was needed.

It is pretty easy to use, and has no external dependencies:

```
>>> import matplotlib as mpl
>>> from pathlib import Path
>>> afm_path = Path(mpl.get_data_path(), 'fonts', 'afm', 'ptmr8a.afm')
>>>
>>> from matplotlib.afm import AFM
>>> with afm_path.open('rb') as fh:
...     afm = AFM(fh)
>>> afm.string_width_height('What the heck?')
(6220.0, 694)
>>> afm.get_fontname()
'Times-Roman'
>>> afm.get_kern_dist('A', 'f')
0
>>> afm.get_kern_dist('A', 'y')
-92.0
>>> afm.get_bbox_char('!')
[130, -9, 238, 676]
```

As in the Adobe Font Metrics File Format Specification, all dimensions are given in units of 1/1000 of the scale factor (point size) of the font being used.

class matplotlib.afm.**AFM**(*fh*)

Bases: object

Parse the AFM file in file object *fh*.

property family_name

The font family name, e.g., 'Times'.

get_angle(*self*)

Return the fontangle as float.

get_bbox_char(*self*, *c*, *isord=False*)

get_capheight(*self*)

Return the cap height as float.

get_familyname(*self*)

Return the font family name, e.g., 'Times'.

`get_fontname`(*self*)
Return the font name, e.g., 'Times-Roman'.

`get_fullname`(*self*)
Return the font full name, e.g., 'Times-Roman'.

`get_height_char`(*self*, *c*, *isord=False*)
Get the bounding box (ink) height of character *c* (space is 0).

`get_horizontal_stem_width`(*self*)
Return the standard horizontal stem width as float, or *None* if not specified in AFM file.

`get_kern_dist`(*self*, *c1*, *c2*)
Return the kerning pair distance (possibly 0) for chars *c1* and *c2*.

`get_kern_dist_from_name`(*self*, *name1*, *name2*)
Return the kerning pair distance (possibly 0) for chars *name1* and *name2*.

`get_name_char`(*self*, *c*, *isord=False*)
Get the name of the character, i.e., ';' is 'semicolon'.

`get_str_bbox`(*self*, *s*)
Return the string bounding box.

`get_str_bbox_and_descent`(*self*, *s*)
Return the string bounding box and the maximal descent.

`get_underline_thickness`(*self*)
Return the underline thickness as float.

`get_vertical_stem_width`(*self*)
Return the standard vertical stem width as float, or *None* if not specified in AFM file.

`get_weight`(*self*)
Return the font weight, e.g., 'Bold' or 'Roman'.

`get_width_char`(*self*, *c*, *isord=False*)
Get the width of the character from the character metric WX field.

`get_width_from_char_name`(*self*, *name*)
Get the width of the character from a type1 character name.

`get_xheight`(*self*)
Return the xheight as float.

`string_width_height`(*self*, *s*)
Return the string width (including kerning) and string height as a (*w*, *h*) tuple.

`class matplotlib.afm.CharMetrics`(*width*, *name*, *bbox*)
Bases: `tuple`

Represents the character metrics of a single character.

Notes

The fields do currently only describe a subset of character metrics information defined in the AFM standard.

Create new instance of CharMetrics(width, name, bbox)

bbox
The bbox of the character (B) as a tuple (*llx*, *lly*, *urx*, *ury*).

name
The character name (N).

width
The character width (WX).

class matplotlib.afm.CompositePart(*name*, *dx*, *dy*)
Bases: tuple

Represents the information on a composite element of a composite char.

Create new instance of CompositePart(name, dx, dy)

dx
x-displacement of the part from the origin.

dy
y-displacement of the part from the origin.

name
Name of the part, e.g. 'acute'.

18.3 matplotlib.animation

Table of Contents

18.3.1 Animation

The easiest way to make a live animation in matplotlib is to use one of the *Animation* classes.

Animation	A base class for Animations.
FuncAnimation	Makes an animation by repeatedly calling a function *func*.
ArtistAnimation	Animation using a fixed set of *Artist* objects.

matplotlib.animation.Animation

class matplotlib.animation.**Animation**(*fig*, *event_source=None*, *blit=False*)

A base class for Animations.

This class is not usable as is, and should be subclassed to provide needed behavior.

Parameters

fig

[*Figure*] The figure object used to get needed events, such as draw or resize.

event_source

[object, optional] A class that can run a callback when desired events are generated, as well as be stopped and started.

Examples include timers (see *TimedAnimation*) and file system notifications.

blit

[bool, default: False] Whether blitting is used to optimize drawing.

See also:

FuncAnimation, *ArtistAnimation*

__init__(*self*, *fig*, *event_source=None*, *blit=False*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(self, fig[, event_source, blit])	Initialize self.
new_frame_seq(self)	Return a new sequence of frame information.
new_saved_frame_seq(self)	Return a new sequence of saved/cached frame information.
save(self, filename[, writer, fps, dpi, ...])	Save the animation as a movie file by drawing every frame.
to_html5_video(self[, embed_limit])	Convert the animation to an HTML5 `<video>` tag.
to_jshtml(self[, fps, embed_frames, ...])	Generate HTML representation of the animation

new_frame_seq(*self*)
Return a new sequence of frame information.

new_saved_frame_seq(*self*)
Return a new sequence of saved/cached frame information.

save(*self*, *filename*, *writer=None*, *fps=None*, *dpi=None*, *codec=None*, *bitrate=None*, *extra_args=None*, *metadata=None*, *extra_anim=None*, *savefig_kwargs=None*, *, *progress_callback=None*)
Save the animation as a movie file by drawing every frame.

Parameters

filename

[str] The output filename, e.g., `mymovie.mp4`.

writer

[*MovieWriter* or str, default: `rcParams["animation.writer"]` (default: `'ffmpeg'`)] A *MovieWriter* instance to use or a key that identifies a class to use, such as 'ffmpeg'.

fps

[int, optional] Movie frame rate (per second). If not set, the frame rate from the animation's frame interval.

dpi

[float, default: `rcParams["savefig.dpi"]` (default: `'figure'`)] Controls the dots per inch for the movie frames. Together with the figure's size in inches, this controls the size of the movie.

codec

[str, default: `rcParams["animation.codec"]` (default: `'h264'`).] The video codec to use. Not all codecs are supported by a given *MovieWriter*.

bitrate

[int, default: `rcParams["animation.bitrate"]` (default: -1)] The bitrate of the movie, in kilobits per second. Higher values means higher quality movies, but increase the file size. A value of -1 lets the underlying movie encoder select the bitrate.

extra_args

[list of str or None, optional] Extra command-line arguments passed to the underlying movie encoder. The default, None, means to use `rcParams["animation.[name-of-encoder]_args"]` for the builtin writers.

metadata

[Dict[str, str], default {}] Dictionary of keys and values for metadata to include in the output file. Some keys that may be of use include: title, artist, genre, subject, copyright, srcform, comment.

extra_anim

[list, default: []] Additional *Animation* objects that should be included in the saved movie file. These need to be from the same *matplotlib.figure.Figure* instance. Also, animation frames will just be simply combined, so there should be a 1:1 correspondence between the frames from the different animations.

savefig_kwargs

[dict, default: {}] Keyword arguments passed to each *savefig* call used to save the individual frames.

progress_callback

[function, optional] A callback function that will be called for every frame to notify the saving progress. It must have the signature

```
def func(current_frame: int, total_frames: int) -> Any
```

where *current_frame* is the current frame number and *total_frames* is the total number of frames to be saved. *total_frames* is set to None, if the total number of frames can not be determined. Return values may exist but are ignored.

Example code to write the progress to stdout:

```
progress_callback =                     lambda i, n: print(f
↪'Saving frame {i} of {n}')
```

Notes

fps, *codec*, *bitrate*, *extra_args* and *metadata* are used to construct a *MovieWriter* instance and can only be passed if *writer* is a string. If they are passed as non-*None* and *writer* is a *MovieWriter*, a RuntimeError will be raised.

`to_html5_video`(*self*, *embed_limit=None*)
Convert the animation to an HTML5 `<video>` tag.

This saves the animation as an h264 video, encoded in base64 directly into the HTML5 video tag. This respects `rcParams["animation.writer"]` (default: `'ffmpeg'`) and `rcParams["animation.bitrate"]` (default: `-1`). This also makes use of the `interval` to control the speed, and uses the `repeat` parameter to decide whether to loop.

Parameters

embed_limit

[float, optional] Limit, in MB, of the returned animation. No animation is created if the limit is exceeded. Defaults to `rcParams["animation.embed_limit"]` (default: `20.0`) = 20.0.

Returns

str

An HTML5 video tag with the animation embedded as base64 encoded h264 video. If the *embed_limit* is exceeded, this returns the string "Video too large to embed."

`to_jshtml`(*self*, *fps=None*, *embed_frames=True*, *default_mode=None*)
Generate HTML representation of the animation

matplotlib.animation.FuncAnimation

class matplotlib.animation.**FuncAnimation**(*fig*, *func*, *frames=None*, *init_func=None*, *fargs=None*, *save_count=None*, ***, *cache_frame_data=True*, ***kwargs*)
Makes an animation by repeatedly calling a function *func*.

Parameters

fig

[*Figure*] The figure object used to get needed events, such as draw or resize.

func

[callable] The function to call at each frame. The first argument will be the next value in *frames*. Any additional positional arguments can be supplied via the *fargs* parameter.

The required signature is:

```
def func(frame, *fargs) -> iterable_of_artists
```

If `blit == True`, *func* must return an iterable of all artists that were modified or created. This information is used by the blitting algorithm to determine which parts of the figure have to be updated. The return value is unused if `blit == False` and may be omitted in that case.

frames

[iterable, int, generator function, or None, optional] Source of data to pass *func* and each frame of the animation

- If an iterable, then simply use the values provided. If the iterable has a length, it will override the *save_count* kwarg.
- If an integer, then equivalent to passing `range(frames)`
- If a generator function, then must have the signature:

```
def gen_function() -> obj
```

- If *None*, then equivalent to passing `itertools.count`.

In all of these cases, the values in *frames* is simply passed through to the user-supplied *func* and thus can be of any type.

init_func

[callable, optional] A function used to draw a clear frame. If not given, the results of drawing from the first item in the frames sequence will be used. This function will be called once before the first frame.

The required signature is:

```
def init_func() -> iterable_of_artists
```

If `blit == True`, *init_func* must return an iterable of artists to be re-drawn. This information is used by the blitting algorithm to determine which parts of the figure have to be updated. The return value is unused if `blit == False` and may be omitted in that case.

fargs

[tuple or None, optional] Additional arguments to pass to each call to *func*.

save_count

[int, default: 100] Fallback for the number of values from *frames* to cache. This is only used if the number of frames cannot be inferred from *frames*, i.e. when it's an iterator without length or a generator.

interval

[int, default: 200] Delay between frames in milliseconds.

repeat_delay

[int, default: 0] The delay in milliseconds between consecutive animation runs, if *repeat* is True.

repeat

[bool, default: True] Whether the animation repeats when the sequence of frames is completed.

blit

[bool, default: False] Whether blitting is used to optimize drawing. Note: when using blitting, any animated artists will be drawn according to their zorder; however, they will be drawn on top of any previous artists, regardless of their zorder.

cache_frame_data

[bool, default: True] Whether frame data is cached. Disabling cache might be helpful when frames contain large objects.

`__init__`(*self*, *fig*, *func*, *frames=None*, *init_func=None*, *fargs=None*, *save_count=None*, *, *cache_frame_data=True*, **kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(self, fig, func[, frames, ...])	Initialize self.
new_frame_seq(self)	Return a new sequence of frame information.
new_saved_frame_seq(self)	Return a new sequence of saved/cached frame information.
save(self, filename[, writer, fps, dpi, ...])	Save the animation as a movie file by drawing every frame.
to_html5_video(self[, embed_limit])	Convert the animation to an HTML5 <video> tag.

continues on next page

Table 3 – continued from previous page

`to_jshtml`(self[, fps, embed_frames, ...])	Generate HTML representation of the animation

`new_frame_seq`(*self*)
Return a new sequence of frame information.

`new_saved_frame_seq`(*self*)
Return a new sequence of saved/cached frame information.

matplotlib.animation.ArtistAnimation

`class matplotlib.animation.ArtistAnimation`(*fig*, *artists*, **args*, ***kwargs*)
Animation using a fixed set of *Artist* objects.

Before creating an instance, all plotting should have taken place and the relevant artists saved.

Parameters

fig

[*Figure*] The figure object used to get needed events, such as draw or resize.

artists

[list] Each list entry is a collection of artists that are made visible on the corresponding frame. Other artists are made invisible.

interval

[int, default: 200] Delay between frames in milliseconds.

repeat_delay

[int, default: 0] The delay in milliseconds between consecutive animation runs, if *repeat* is True.

repeat

[bool, default: True] Whether the animation repeats when the sequence of frames is completed.

blit

[bool, default: False] Whether blitting is used to optimize drawing.

`__init__`(*self*, *fig*, *artists*, **args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(self, fig, artists, *args, **kwargs)	Initialize self.
new_frame_seq(self)	Return a new sequence of frame information.
new_saved_frame_seq(self)	Return a new sequence of saved/cached frame information.
save(self, filename[, writer, fps, dpi, ...])	Save the animation as a movie file by drawing every frame.
to_html5_video(self[, embed_limit])	Convert the animation to an HTML5 <video> tag.
to_jshtml(self[, fps, embed_frames, ...])	Generate HTML representation of the animation

In both cases it is critical to keep a reference to the instance object. The animation is advanced by a timer (typically from the host GUI framework) which the *Animation* object holds the only reference to. If you do not hold a reference to the *Animation* object, it (and hence the timers), will be garbage collected which will stop the animation.

To save an animation to disk use *Animation.save* or *Animation.to_html5_video*

See *Helper Classes* below for details about what movie formats are supported.

FuncAnimation

The inner workings of *FuncAnimation* is more-or-less:

```
for d in frames:
   artists = func(d, *fargs)
   fig.canvas.draw_idle()
   fig.canvas.start_event_loop(interval)
```

with details to handle 'blitting' (to dramatically improve the live performance), to be non-blocking, not repeatedly start/stop the GUI event loop, handle repeats, multiple animated axes, and easily save the animation to a movie file.

'Blitting' is a standard technique in computer graphics. The general gist is to take an existing bit map (in our case a mostly rasterized figure) and then 'blit' one more artist on top. Thus, by managing a saved 'clean' bitmap, we can only re-draw the few artists that are changing at each frame and possibly save significant amounts of time. When we use blitting (by passing `blit=True`), the core loop of *FuncAnimation* gets a bit more complicated:

```
ax = fig.gca()

def update_blit(artists):
```

(continues on next page)

(continued from previous page)

```
    fig.canvas.restore_region(bg_cache)
    for a in artists:
        a.axes.draw_artist(a)

    ax.figure.canvas.blit(ax.bbox)

artists = init_func()

for a in artists:
   a.set_animated(True)

fig.canvas.draw()
bg_cache = fig.canvas.copy_from_bbox(ax.bbox)

for f in frames:
    artists = func(f, *fargs)
    update_blit(artists)
    fig.canvas.start_event_loop(interval)
```

This is of course leaving out many details (such as updating the background when the figure is resized or fully re-drawn). However, this hopefully minimalist example gives a sense of how `init_func` and `func` are used inside of *FuncAnimation* and the theory of how 'blitting' works.

The expected signature on `func` and `init_func` is very simple to keep *FuncAnimation* out of your book keeping and plotting logic, but this means that the callable objects you pass in must know what artists they should be working on. There are several approaches to handling this, of varying complexity and encapsulation. The simplest approach, which works quite well in the case of a script, is to define the artist at a global scope and let Python sort things out. For example

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation

fig, ax = plt.subplots()
xdata, ydata = [], []
ln, = plt.plot([], [], 'ro')

def init():
    ax.set_xlim(0, 2*np.pi)
    ax.set_ylim(-1, 1)
    return ln,

def update(frame):
    xdata.append(frame)
    ydata.append(np.sin(frame))
    ln.set_data(xdata, ydata)
    return ln,

ani = FuncAnimation(fig, update, frames=np.linspace(0, 2*np.pi, 128),
```

(continues on next page)

(continued from previous page)

```
                    init_func=init, blit=True)
plt.show()
```

The second method is to use `functools.partial` to 'bind' artists to function. A third method is to use closures to build up the required artists and functions. A fourth method is to create a class.

Examples

Decay

This example showcases: - using a generator to drive an animation, - changing axes limits during an animation.

```
import itertools

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation


def data_gen():
    for cnt in itertools.count():
        t = cnt / 10
        yield t, np.sin(2*np.pi*t) * np.exp(-t/10.)


def init():
    ax.set_ylim(-1.1, 1.1)
    ax.set_xlim(0, 10)
    del xdata[:]
    del ydata[:]
    line.set_data(xdata, ydata)
    return line,

fig, ax = plt.subplots()
line, = ax.plot([], [], lw=2)
ax.grid()
xdata, ydata = [], []


def run(data):
    # update the data
    t, y = data
    xdata.append(t)
    ydata.append(y)
    xmin, xmax = ax.get_xlim()

    if t >= xmax:
```

(continues on next page)

(continued from previous page)

```
        ax.set_xlim(xmin, 2*xmax)
        ax.figure.canvas.draw()
    line.set_data(xdata, ydata)

    return line,

ani = animation.FuncAnimation(fig, run, data_gen, interval=10, init_func=init)
plt.show()
```

Total running time of the script: (0 minutes 9.620 seconds)

The Bayes update

This animation displays the posterior estimate updates as it is refitted when new data arrives. The vertical line represents the theoretical value to which the plotted distribution should converge.

```
import math

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation


def beta_pdf(x, a, b):
    return (x**(a-1) * (1-x)**(b-1) * math.gamma(a + b)
            / (math.gamma(a) * math.gamma(b)))


class UpdateDist:
    def __init__(self, ax, prob=0.5):
        self.success = 0
        self.prob = prob
        self.line, = ax.plot([], [], 'k-')
        self.x = np.linspace(0, 1, 200)
        self.ax = ax

        # Set up plot parameters
        self.ax.set_xlim(0, 1)
        self.ax.set_ylim(0, 10)
        self.ax.grid(True)

        # This vertical line represents the theoretical value, to
        # which the plotted distribution should converge.
        self.ax.axvline(prob, linestyle='--', color='black')

    def __call__(self, i):
        # This way the plot can continuously run and we just keep
        # watching new realizations of the process
```

(continues on next page)

(continued from previous page)

```
        if i == 0:
            self.success = 0
            self.line.set_data([], [])
            return self.line,

        # Choose success based on exceed a threshold with a uniform pick
        if np.random.rand(1,) < self.prob:
            self.success += 1
        y = beta_pdf(self.x, self.success + 1, (i - self.success) + 1)
        self.line.set_data(self.x, y)
        return self.line,

# Fixing random state for reproducibility
np.random.seed(19680801)


fig, ax = plt.subplots()
ud = UpdateDist(ax, prob=0.7)
anim = FuncAnimation(fig, ud, frames=100, interval=100, blit=True)
plt.show()
```

Total running time of the script: (0 minutes 8.195 seconds)

The double pendulum problem

This animation illustrates the double pendulum problem.

Double pendulum formula translated from the C code at http://www.physics.usyd.edu.au/~wheat/dpend_html/solve_dpend.c

```
from numpy import sin, cos
import numpy as np
import matplotlib.pyplot as plt
import scipy.integrate as integrate
import matplotlib.animation as animation

G = 9.8  # acceleration due to gravity, in m/s^2
L1 = 1.0  # length of pendulum 1 in m
L2 = 1.0  # length of pendulum 2 in m
M1 = 1.0  # mass of pendulum 1 in kg
M2 = 1.0  # mass of pendulum 2 in kg
t_stop = 5  # how many seconds to simulate


def derivs(state, t):

    dydx = np.zeros_like(state)
    dydx[0] = state[1]

    delta = state[2] - state[0]
```

(continues on next page)

(continued from previous page)

```
    den1 = (M1+M2) * L1 - M2 * L1 * cos(delta) * cos(delta)
    dydx[1] = ((M2 * L1 * state[1] * state[1] * sin(delta) * cos(delta)
                + M2 * G * sin(state[2]) * cos(delta)
                + M2 * L2 * state[3] * state[3] * sin(delta)
                - (M1+M2) * G * sin(state[0]))
               / den1)

    dydx[2] = state[3]

    den2 = (L2/L1) * den1
    dydx[3] = ((- M2 * L2 * state[3] * state[3] * sin(delta) * cos(delta)
                + (M1+M2) * G * sin(state[0]) * cos(delta)
                - (M1+M2) * L1 * state[1] * state[1] * sin(delta)
                - (M1+M2) * G * sin(state[2]))
               / den2)

    return dydx

# create a time array from 0..100 sampled at 0.05 second steps
dt = 0.05
t = np.arange(0, t_stop, dt)

# th1 and th2 are the initial angles (degrees)
# w10 and w20 are the initial angular velocities (degrees per second)
th1 = 120.0
w1 = 0.0
th2 = -10.0
w2 = 0.0

# initial state
state = np.radians([th1, w1, th2, w2])

# integrate your ODE using scipy.integrate.
y = integrate.odeint(derivs, state, t)

x1 = L1*sin(y[:, 0])
y1 = -L1*cos(y[:, 0])

x2 = L2*sin(y[:, 2]) + x1
y2 = -L2*cos(y[:, 2]) + y1

fig = plt.figure(figsize=(5, 4))
ax = fig.add_subplot(111, autoscale_on=False, xlim=(-2, 2), ylim=(-2, 1))
ax.set_aspect('equal')
ax.grid()

line, = ax.plot([], [], 'o-', lw=2)
time_template = 'time = %.1fs'
time_text = ax.text(0.05, 0.9, '', transform=ax.transAxes)
```

(continues on next page)

(continued from previous page)

```
def animate(i):
    thisx = [0, x1[i], x2[i]]
    thisy = [0, y1[i], y2[i]]

    line.set_data(thisx, thisy)
    time_text.set_text(time_template % (i*dt))
    return line, time_text


ani = animation.FuncAnimation(
    fig, animate, len(y), interval=dt*1000, blit=True)
plt.show()
```

Total running time of the script: (0 minutes 9.800 seconds)

Animated histogram

Use a path patch to draw a bunch of rectangles for an animated histogram.

```
import numpy as np

import matplotlib.pyplot as plt
import matplotlib.patches as patches
import matplotlib.path as path
import matplotlib.animation as animation

# Fixing random state for reproducibility
np.random.seed(19680801)

# histogram our data with numpy
data = np.random.randn(1000)
n, bins = np.histogram(data, 100)

# get the corners of the rectangles for the histogram
left = bins[:-1]
right = bins[1:]
bottom = np.zeros(len(left))
top = bottom + n
nrects = len(left)
```

Here comes the tricky part -- we have to set up the vertex and path codes arrays using *Path.MOVETO*, *Path.LINETO* and *Path.CLOSEPOLY* for each rect.

- We need 1 `MOVETO` per rectangle, which sets the initial point.
- We need 3 `LINETO`'s, which tell Matplotlib to draw lines from vertex 1 to vertex 2, v2 to v3, and v3 to v4.
- We then need one `CLOSEPOLY` which tells Matplotlib to draw a line from the v4 to our initial vertex (the `MOVETO` vertex), in order to close the polygon.

Note: The vertex for `CLOSEPOLY` is ignored, but we still need a placeholder in the `verts` array to keep the codes aligned with the vertices.

```
nverts = nrects * (1 + 3 + 1)
verts = np.zeros((nverts, 2))
codes = np.full(nverts, path.Path.LINETO)
codes[0::5] = path.Path.MOVETO
codes[4::5] = path.Path.CLOSEPOLY
verts[0::5, 0] = left
verts[0::5, 1] = bottom
verts[1::5, 0] = left
verts[1::5, 1] = top
verts[2::5, 0] = right
verts[2::5, 1] = top
verts[3::5, 0] = right
verts[3::5, 1] = bottom
```

To animate the histogram, we need an `animate` function, which generates a random set of numbers and updates the locations of the vertices for the histogram (in this case, only the heights of each rectangle). `patch` will eventually be a *Patch* object.

```
patch = None


def animate(i):
    # simulate new data coming in
    data = np.random.randn(1000)
    n, bins = np.histogram(data, 100)
    top = bottom + n
    verts[1::5, 1] = top
    verts[2::5, 1] = top
    return [patch, ]
```

And now we build the *Path* and *Patch* instances for the histogram using our vertices and codes. We add the patch to the *Axes* instance, and setup the *FuncAnimation* with our `animate` function.

```
fig, ax = plt.subplots()
barpath = path.Path(verts, codes)
patch = patches.PathPatch(
    barpath, facecolor='green', edgecolor='yellow', alpha=0.5)
ax.add_patch(patch)

ax.set_xlim(left[0], right[-1])
ax.set_ylim(bottom.min(), top.max())

ani = animation.FuncAnimation(fig, animate, 50, repeat=False, blit=True)
plt.show()
```

Total running time of the script: (0 minutes 4.742 seconds)

Rain simulation

Simulates rain drops on a surface by animating the scale and opacity of 50 scatter points.

Author: Nicolas P. Rougier

Out:

```
/root/matplotlib/examples/animation/rain.py:28: FutureWarning: Passing (type, 1) or
↪'1type' as a synonym of type is deprecated; in a future version of numpy, it will be␣
↪understood as (type, (1,)) / '(1,)type'.
  rain_drops = np.zeros(n_drops, dtype=[('position', float, 2),
```

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation

# Fixing random state for reproducibility
np.random.seed(19680801)


# Create new Figure and an Axes which fills it.
fig = plt.figure(figsize=(7, 7))
ax = fig.add_axes([0, 0, 1, 1], frameon=False)
ax.set_xlim(0, 1), ax.set_xticks([])
ax.set_ylim(0, 1), ax.set_yticks([])

# Create rain data
n_drops = 50
rain_drops = np.zeros(n_drops, dtype=[('position', float, 2),
                                      ('size',     float, 1),
                                      ('growth',   float, 1),
                                      ('color',    float, 4)])

# Initialize the raindrops in random positions and with
# random growth rates.
rain_drops['position'] = np.random.uniform(0, 1, (n_drops, 2))
rain_drops['growth'] = np.random.uniform(50, 200, n_drops)

# Construct the scatter which we will update during animation
# as the raindrops develop.
scat = ax.scatter(rain_drops['position'][:, 0], rain_drops['position'][:, 1],
                  s=rain_drops['size'], lw=0.5, edgecolors=rain_drops['color'],
                  facecolors='none')
```

(continues on next page)

(continued from previous page)

```
def update(frame_number):
    # Get an index which we can use to re-spawn the oldest raindrop.
    current_index = frame_number % n_drops

    # Make all colors more transparent as time progresses.
    rain_drops['color'][:, 3] -= 1.0/len(rain_drops)
    rain_drops['color'][:, 3] = np.clip(rain_drops['color'][:, 3], 0, 1)

    # Make all circles bigger.
    rain_drops['size'] += rain_drops['growth']

    # Pick a new position for oldest rain drop, resetting its size,
    # color and growth factor.
    rain_drops['position'][current_index] = np.random.uniform(0, 1, 2)
    rain_drops['size'][current_index] = 5
    rain_drops['color'][current_index] = (0, 0, 0, 1)
    rain_drops['growth'][current_index] = np.random.uniform(50, 200)

    # Update the scatter collection, with the new colors, sizes and positions.
    scat.set_edgecolors(rain_drops['color'])
    scat.set_sizes(rain_drops['size'])
    scat.set_offsets(rain_drops['position'])


# Construct the animation, using the update function as the animation director.
animation = FuncAnimation(fig, update, interval=10)
plt.show()
```

Total running time of the script: (0 minutes 5.126 seconds)

Animated 3D random walk

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

# Fixing random state for reproducibility
np.random.seed(19680801)


def gen_rand_line(length, dims=2):
    """
    Create a line using a random walk algorithm.

    Parameters
    ----------
    length : int
        The number of points of the line.
    dims : int
```

(continues on next page)

(continued from previous page)

```
        The number of dimensions of the line.
    """
    line_data = np.empty((dims, length))
    line_data[:, 0] = np.random.rand(dims)
    for index in range(1, length):
        # scaling the random numbers by 0.1 so
        # movement is small compared to position.
        # subtraction by 0.5 is to change the range to [-0.5, 0.5]
        # to allow a line to move backwards.
        step = (np.random.rand(dims) - 0.5) * 0.1
        line_data[:, index] = line_data[:, index - 1] + step
    return line_data


def update_lines(num, data_lines, lines):
    for line, data in zip(lines, data_lines):
        # NOTE: there is no .set_data() for 3 dim data...
        line.set_data(data[0:2, :num])
        line.set_3d_properties(data[2, :num])
    return lines


# Attaching 3D axis to the figure
fig = plt.figure()
ax = fig.add_subplot(projection="3d")

# Fifty lines of random 3-D lines
data = [gen_rand_line(25, 3) for index in range(50)]

# Creating fifty line objects.
# NOTE: Can't pass empty arrays into 3d version of plot()
lines = [ax.plot(dat[0, 0:1], dat[1, 0:1], dat[2, 0:1])[0] for dat in data]

# Setting the axes properties
ax.set_xlim3d([0.0, 1.0])
ax.set_xlabel('X')

ax.set_ylim3d([0.0, 1.0])
ax.set_ylabel('Y')

ax.set_zlim3d([0.0, 1.0])
ax.set_zlabel('Z')

ax.set_title('3D Test')

# Creating the Animation object
line_ani = animation.FuncAnimation(
    fig, update_lines, 50, fargs=(data, lines), interval=50)

plt.show()
```

Total running time of the script: (0 minutes 8.248 seconds)

Animated line plot

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

fig, ax = plt.subplots()

x = np.arange(0, 2*np.pi, 0.01)
line, = ax.plot(x, np.sin(x))


def animate(i):
    line.set_ydata(np.sin(x + i / 50))  # update the data.
    return line,


ani = animation.FuncAnimation(
    fig, animate, interval=20, blit=True, save_count=50)

# To save the animation, use e.g.
#
# ani.save("movie.mp4")
#
# or
#
# writer = animation.FFMpegWriter(
#     fps=15, metadata=dict(artist='Me'), bitrate=1800)
# ani.save("movie.mp4", writer=writer)

plt.show()
```

Total running time of the script: (0 minutes 4.546 seconds)

Oscilloscope

Emulates an oscilloscope.

```
import numpy as np
from matplotlib.lines import Line2D
import matplotlib.pyplot as plt
import matplotlib.animation as animation


class Scope:
    def __init__(self, ax, maxt=2, dt=0.02):
        self.ax = ax
        self.dt = dt
        self.maxt = maxt
        self.tdata = [0]
```

(continues on next page)

(continued from previous page)

```
        self.ydata = [0]
        self.line = Line2D(self.tdata, self.ydata)
        self.ax.add_line(self.line)
        self.ax.set_ylim(-.1, 1.1)
        self.ax.set_xlim(0, self.maxt)

    def update(self, y):
        lastt = self.tdata[-1]
        if lastt > self.tdata[0] + self.maxt:  # reset the arrays
            self.tdata = [self.tdata[-1]]
            self.ydata = [self.ydata[-1]]
            self.ax.set_xlim(self.tdata[0], self.tdata[0] + self.maxt)
            self.ax.figure.canvas.draw()

        t = self.tdata[-1] + self.dt
        self.tdata.append(t)
        self.ydata.append(y)
        self.line.set_data(self.tdata, self.ydata)
        return self.line,


def emitter(p=0.1):
    """Return a random value in [0, 1) with probability p, else 0."""
    while True:
        v = np.random.rand(1)
        if v > p:
            yield 0.
        else:
            yield np.random.rand(1)


# Fixing random state for reproducibility
np.random.seed(19680801 // 10)


fig, ax = plt.subplots()
scope = Scope(ax)

# pass a generator in "emitter" to produce data for the update func
ani = animation.FuncAnimation(fig, scope.update, emitter, interval=50,
                              blit=True)

plt.show()
```

Total running time of the script: (0 minutes 8.739 seconds)

MATPLOTLIB UNCHAINED

Comparative path demonstration of frequency from a fake signal of a pulsar (mostly known because of the cover for Joy Division's Unknown Pleasures).

Author: Nicolas P. Rougier

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

# Fixing random state for reproducibility
np.random.seed(19680801)


# Create new Figure with black background
fig = plt.figure(figsize=(8, 8), facecolor='black')

# Add a subplot with no frame
ax = plt.subplot(111, frameon=False)

# Generate random data
data = np.random.uniform(0, 1, (64, 75))
X = np.linspace(-1, 1, data.shape[-1])
G = 1.5 * np.exp(-4 * X ** 2)

# Generate line plots
lines = []
for i in range(len(data)):
    # Small reduction of the X extents to get a cheap perspective effect
    xscale = 1 - i / 200.
    # Same for linewidth (thicker strokes on bottom)
    lw = 1.5 - i / 100.0
    line, = ax.plot(xscale * X, i + G * data[i], color="w", lw=lw)
    lines.append(line)

# Set y limit (or first line is cropped because of thickness)
ax.set_ylim(-1, 70)

# No ticks
ax.set_xticks([])
ax.set_yticks([])

# 2 part titles to get different font weights
ax.text(0.5, 1.0, "MATPLOTLIB ", transform=ax.transAxes,
        ha="right", va="bottom", color="w",
        family="sans-serif", fontweight="light", fontsize=16)
ax.text(0.5, 1.0, "UNCHAINED", transform=ax.transAxes,
        ha="left", va="bottom", color="w",
        family="sans-serif", fontweight="bold", fontsize=16)


def update(*args):
```

(continues on next page)

(continued from previous page)

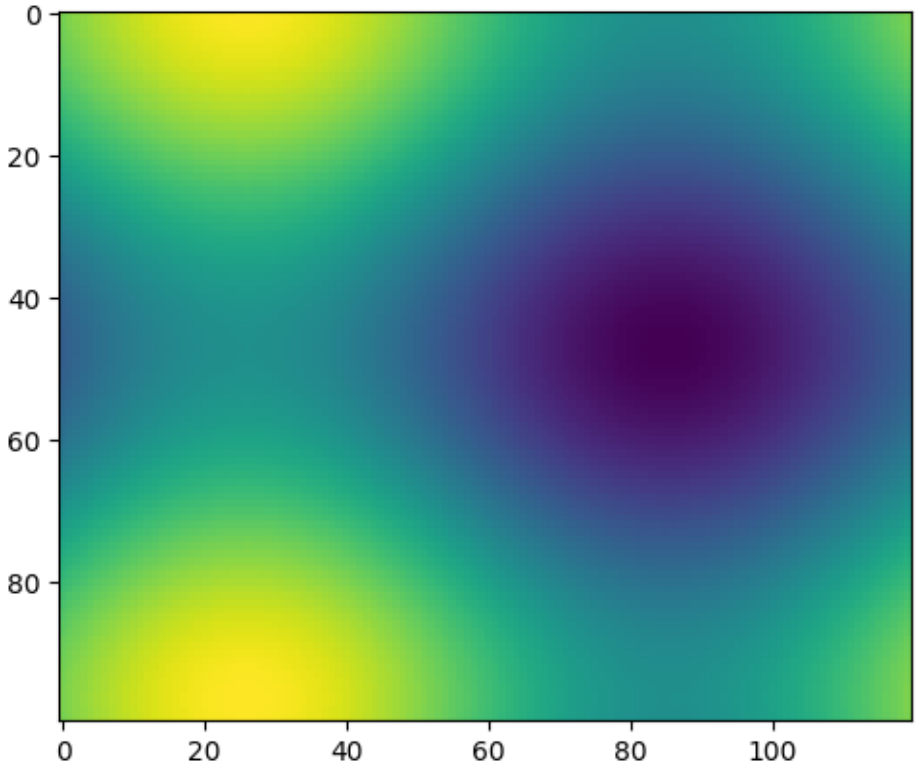
```
    # Shift all data to the right
    data[:, 1:] = data[:, :-1]

    # Fill-in new values
    data[:, 0] = np.random.uniform(0, 1, len(data))

    # Update data
    for i in range(len(data)):
        lines[i].set_ydata(i + G * data[i])

    # Return modified artists
    return lines

# Construct the animation, using the update function as the animation director.
anim = animation.FuncAnimation(fig, update, interval=10)
plt.show()
```

Total running time of the script: (0 minutes 9.183 seconds)

`ArtistAnimation`

Examples

Animated image using a precomputed list of images




```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

fig, ax = plt.subplots()


def f(x, y):
    return np.sin(x) + np.cos(y)

x = np.linspace(0, 2 * np.pi, 120)
y = np.linspace(0, 2 * np.pi, 100).reshape(-1, 1)

# ims is a list of lists, each row is a list of artists to draw in the
# current frame; here we are just animating one artist, the image, in
# each frame
ims = []
for i in range(60):
    x += np.pi / 15.
    y += np.pi / 20.
```

(continues on next page)

(continued from previous page)

```
    im = ax.imshow(f(x, y), animated=True)
    if i == 0:
        ax.imshow(f(x, y))  # show an initial one first
    ims.append([im])

ani = animation.ArtistAnimation(fig, ims, interval=50, blit=True,
                                repeat_delay=1000)

# To save the animation, use e.g.
#
# ani.save("movie.mp4")
#
# or
#
# writer = animation.FFMpegWriter(
#     fps=15, metadata=dict(artist='Me'), bitrate=1800)
# ani.save("movie.mp4", writer=writer)

plt.show()
```

18.3.2 Writer Classes

The provided writers fall into a few broad categories.

The Pillow writer relies on the Pillow library to write the animation, keeping all data in memory.

PillowWriter

matplotlib.animation.PillowWriter

class matplotlib.animation.**PillowWriter**(*fps=5*, *metadata=None*, *codec=None*, *bitrate=None*)

__init__(*self*, *fps=5*, *metadata=None*, *codec=None*, *bitrate=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(self[, fps, metadata, codec, bitrate])	Initialize self.
finish(self)	Finish any processing for writing the movie.
grab_frame(self, **savefig_kwargs)	Grab the image information from the figure and save as a movie frame.
isAvailable()	
saving(self, fig, outfile, dpi, *args, **kwargs)	Context manager to facilitate writing the movie file.
setup(self, fig, outfile[, dpi])	Setup for writing the movie file.

Attributes

frame_size	A tuple `(width, height)` in pixels of a movie frame.

`finish`(*self*)
: Finish any processing for writing the movie.

`grab_frame`(*self*, ***savefig_kwargs*)
: Grab the image information from the figure and save as a movie frame.

  All keyword arguments in *savefig_kwargs* are passed on to the *savefig* call that saves the figure.

`classmethod isAvailable`()

`setup`(*self*, *fig*, *outfile*, *dpi=None*)
: Setup for writing the movie file.

  Parameters

  fig

  [*Figure*] The figure object that contains the information for frames.

  outfile

  [str] The filename of the resulting movie file.

  dpi

  [float, default: `fig.dpi`] The DPI (or resolution) for the file. This controls the size in pixels of the resulting movie file.

The HTML writer generates JavaScript-based animations.

HTMLWriter	Writer for JavaScript-based HTML movies.

matplotlib.animation.HTMLWriter

class matplotlib.animation.**HTMLWriter**(*fps=30*, *codec=None*, *bitrate=None*, *extra_args=None*, *metadata=None*, *embed_frames=False*, *default_mode='loop'*, *embed_limit=None*)

Writer for JavaScript-based HTML movies.

Parameters

fps

[int, default: 5] Movie frame rate (per second).

codec

[str or None, default: `rcParams["animation.codec"]` (default: `'h264'`)] The codec to use.

bitrate

[int, default: `rcParams["animation.bitrate"]` (default: `-1`)] The bitrate of the movie, in kilobits per second. Higher values means higher quality movies, but increase the file size. A value of -1 lets the underlying movie encoder select the bitrate.

extra_args

[list of str or None, optional] Extra command-line arguments passed to the underlying movie encoder. The default, None, means to use `rcParams["animation.[name-of-encoder]_args"]` for the builtin writers.

metadata

[Dict[str, str], default: {}] A dictionary of keys and values for metadata to include in the output file. Some keys that may be of use include: title, artist, genre, subject, copyright, srcform, comment.

__init__(*self*, *fps=30*, *codec=None*, *bitrate=None*, *extra_args=None*, *metadata=None*, *embed_frames=False*, *default_mode='loop'*, *embed_limit=None*)

Parameters

fps

[int, default: 5] Movie frame rate (per second).

codec

[str or None, default: `rcParams["animation.codec"]` (default: `'h264'`)] The codec to use.

bitrate

[int, default: `rcParams["animation.bitrate"]` (default: `-1`)] The bitrate of the movie, in kilobits per second. Higher values means higher quality movies, but increase the file size. A value of -1 lets the underlying movie encoder select the bitrate.

extra_args

[list of str or None, optional] Extra command-line arguments passed to the underlying movie encoder. The default, None, means to use `rcParams["animation.[name-of-encoder]_args"]` for the builtin writers.

metadata

[Dict[str, str], default: {}] A dictionary of keys and values for metadata to include in the output file. Some keys that may be of use include: title, artist, genre, subject, copyright, srcform, comment.

Methods

__init__(self[, fps, codec, bitrate, ...])	**Parameters**
bin_path()	Return the binary path to the commandline tool used by a specific subclass.
cleanup(self)	Clean-up and collect the process used to write the movie file.
finish(self)	Finish any processing for writing the movie.
grab_frame(self, **savefig_kwargs)	Grab the image information from the figure and save as a movie frame.
isAvailable()	Return whether a MovieWriter subclass is actually available.
saving(self, fig, outfile, dpi, *args, **kwargs)	Context manager to facilitate writing the movie file.
setup(self, fig, outfile, dpi[, frame_dir])	Setup for writing the movie file.

Attributes

args_key	
clear_temp	
exec_key	
frame_format	Format (png, jpeg, etc.) to use for saving the frames, which can be decided by the individual subclasses.
frame_size	A tuple `(width, height)` in pixels of a movie frame.
supported_formats	

property args_key

finish(*self*)
Finish any processing for writing the movie.

grab_frame(*self*, ***savefig_kwargs*)
Grab the image information from the figure and save as a movie frame.

All keyword arguments in *savefig_kwargs* are passed on to the *savefig* call that saves the figure.

classmethod isAvailable()
Return whether a MovieWriter subclass is actually available.

setup(*self*, *fig*, *outfile*, *dpi*, *frame_dir=None*)
Setup for writing the movie file.

Parameters

fig

[*Figure*] The figure to grab the rendered frames from.

outfile

[str] The filename of the resulting movie file.

dpi

[float, optional] The dpi of the output file. This, with the figure size, controls the size in pixels of the resulting movie file. Default is `fig.dpi`.

frame_prefix

[str, optional] The filename prefix to use for temporary files. If None (the default), files are written to a temporary directory which is deleted by `cleanup` (regardless of the value of *clear_temp*).

clear_temp

[bool, optional] If the temporary files should be deleted after stitching the final result. Setting this to `False` can be useful for debugging. Defaults to `True`.

```
supported_formats = ['png', 'jpeg', 'tiff', 'svg']
```

The pipe-based writers stream the captured frames over a pipe to an external process. The pipe-based variants tend to be more performant, but may not work on all systems.

FFMpegWriter	Pipe-based ffmpeg writer.
ImageMagickWriter	Pipe-based animated gif.
AVConvWriter	Pipe-based avconv writer.

matplotlib.animation.FFMpegWriter

class matplotlib.animation.**FFMpegWriter**(*fps=5*, *codec=None*, *bitrate=None*, *extra_args=None*, *metadata=None*)

Pipe-based ffmpeg writer.

Frames are streamed directly to ffmpeg via a pipe and written in a single pass.

Parameters

fps

[int, default: 5] Movie frame rate (per second).

codec

[str or None, default: `rcParams["animation.codec"]` (default: `'h264'`)] The codec to use.

bitrate

[int, default: `rcParams["animation.bitrate"]` (default: `-1`)] The bitrate of the movie, in kilobits per second. Higher values means higher quality movies, but increase the file size. A value of -1 lets the underlying movie encoder select the bitrate.

extra_args

[list of str or None, optional] Extra command-line arguments passed to the underlying movie encoder. The default, None, means to use `rcParams["animation.[name-of-encoder]_args"]` for the builtin writers.

metadata

[Dict[str, str], default: {}] A dictionary of keys and values for metadata to include in the output file. Some keys that may be of use include: title, artist, genre, subject, copyright, srcform, comment.

__init__(*self*, *fps=5*, *codec=None*, *bitrate=None*, *extra_args=None*, *metadata=None*)

Parameters

fps

[int, default: 5] Movie frame rate (per second).

codec

[str or None, default: `rcParams["animation.codec"]` (default: `'h264'`)] The codec to use.

bitrate

[int, default: `rcParams["animation.bitrate"]` (default: `-1`)] The bitrate of the movie, in kilobits per second. Higher values means higher quality movies, but increase the file size. A value of -1 lets the underlying movie encoder select the bitrate.

extra_args

[list of str or None, optional] Extra command-line arguments passed to the underlying movie encoder. The default, None, means to use `rcParams["animation.[name-of-encoder]_args"]` for the builtin writers.

metadata

[Dict[str, str], default: {}] A dictionary of keys and values for metadata to include in the output file. Some keys that may be of use include: title, artist, genre, subject, copyright, srcform, comment.

Methods

__init__(self[, fps, codec, bitrate, ...])	**Parameters**
bin_path()	Return the binary path to the commandline tool used by a specific subclass.
cleanup(self)	Clean-up and collect the process used to write the movie file.
finish(self)	Finish any processing for writing the movie.
grab_frame(self, **savefig_kwargs)	Grab the image information from the figure and save as a movie frame.

continues on next page

Table 12 – continued from previous page

isAvailable()	Return whether a MovieWriter subclass is actually available.
saving(self, fig, outfile, dpi, *args, **kwargs)	Context manager to facilitate writing the movie file.
setup(self, fig, outfile[, dpi])	Setup for writing the movie file.

Attributes

args_key	
exec_key	
frame_size	A tuple (width, height) in pixels of a movie frame.
output_args	

matplotlib.animation.ImageMagickWriter

class matplotlib.animation.**ImageMagickWriter**(*fps=5*, *codec=None*, *bitrate=None*, *extra_args=None*, *metadata=None*)

Pipe-based animated gif.

Frames are streamed directly to ImageMagick via a pipe and written in a single pass.

Parameters

fps

[int, default: 5] Movie frame rate (per second).

codec

[str or None, default: rcParams["animation.codec"] (default: 'h264')] The codec to use.

bitrate

[int, default: rcParams["animation.bitrate"] (default: -1)] The bitrate of the movie, in kilobits per second. Higher values means higher quality movies, but increase the file size. A value of -1 lets the underlying movie encoder select the bitrate.

extra_args

[list of str or None, optional] Extra command-line arguments passed to the underlying movie encoder. The default, None, means to use rcParams["animation.[name-of-encoder]_args"] for the builtin writers.

metadata

[Dict[str, str], default: {}] A dictionary of keys and values for metadata to include in the output file. Some keys that may be of use include: title, artist, genre, subject, copyright, srcform, comment.

`__init__`(*self*, *fps=5*, *codec=None*, *bitrate=None*, *extra_args=None*, *metadata=None*)

Parameters

fps

[int, default: 5] Movie frame rate (per second).

codec

[str or None, default: `rcParams["animation.codec"]` (default: `'h264'`)] The codec to use.

bitrate

[int, default: `rcParams["animation.bitrate"]` (default: `-1`)] The bitrate of the movie, in kilobits per second. Higher values means higher quality movies, but increase the file size. A value of -1 lets the underlying movie encoder select the bitrate.

extra_args

[list of str or None, optional] Extra command-line arguments passed to the underlying movie encoder. The default, None, means to use `rcParams["animation.[name-of-encoder]_args"]` for the builtin writers.

metadata

[Dict[str, str], default: {}] A dictionary of keys and values for metadata to include in the output file. Some keys that may be of use include: title, artist, genre, subject, copyright, srcform, comment.

Methods

`__init__`(self[, fps, codec, bitrate, ...])	**Parameters**
`bin_path`()	Return the binary path to the commandline tool used by a specific subclass.

continues on next page

Table 14 – continued from previous page

`cleanup`(self)	Clean-up and collect the process used to write the movie file.
`finish`(self)	Finish any processing for writing the movie.
`grab_frame`(self, **savefig_kwargs)	Grab the image information from the figure and save as a movie frame.
`isAvailable`()	Return whether a MovieWriter subclass is actually available.
`saving`(self, fig, outfile, dpi, *args, **kwargs)	Context manager to facilitate writing the movie file.
`setup`(self, fig, outfile[, dpi])	Setup for writing the movie file.

Attributes

`args_key`	
`delay`	
`exec_key`	
`frame_size`	A tuple `(width, height)` in pixels of a movie frame.
`output_args`	

matplotlib.animation.AVConvWriter

class matplotlib.animation.**AVConvWriter**(**args*, ***kwargs*)

Pipe-based avconv writer.

Frames are streamed directly to avconv via a pipe and written in a single pass.

__init__(*self*, */*, **args*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(self, /, *args, **kwargs)	Initialize self.
`bin_path`()	Return the binary path to the commandline tool used by a specific subclass.
`cleanup`(self)	Clean-up and collect the process used to write the movie file.
`finish`(self)	Finish any processing for writing the movie.

continues on next page

Table 16 – continued from previous page

grab_frame(self, **savefig_kwargs)	Grab the image information from the figure and save as a movie frame.
isAvailable()	Return whether a MovieWriter subclass is actually available.
saving(self, fig, outfile, dpi, *args, **kwargs)	Context manager to facilitate writing the movie file.
setup(self, fig, outfile[, dpi])	Setup for writing the movie file.

Attributes

args_key	
exec_key	
frame_size	A tuple (width, height) in pixels of a movie frame.
output_args	

The file-based writers save temporary files for each frame which are stitched into a single file at the end. Although slower, these writers can be easier to debug.

FFMpegFileWriter	File-based ffmpeg writer.
ImageMagickFileWriter	File-based animated gif writer.
AVConvFileWriter	File-based avconv writer.

matplotlib.animation.FFMpegFileWriter

class matplotlib.animation.**FFMpegFileWriter**(**args*, ***kwargs*)

File-based ffmpeg writer.

Frames are written to temporary files on disk and then stitched together at the end.

Parameters

fps

[int, default: 5] Movie frame rate (per second).

codec

[str or None, default: rcParams["animation.codec"] (default: 'h264')] The codec to use.

bitrate

[int, default: rcParams["animation.bitrate"] (default: -1)] The bitrate of the movie, in kilobits per second. Higher values means

higher quality movies, but increase the file size. A value of -1 lets the underlying movie encoder select the bitrate.

extra_args

[list of str or None, optional] Extra command-line arguments passed to the underlying movie encoder. The default, None, means to use `rcParams["animation.[name-of-encoder]_args"]` for the builtin writers.

metadata

[Dict[str, str], default: {}] A dictionary of keys and values for metadata to include in the output file. Some keys that may be of use include: title, artist, genre, subject, copyright, srcform, comment.

`__init__`(*self*, **args*, ***kwargs*)

Parameters

fps

[int, default: 5] Movie frame rate (per second).

codec

[str or None, default: `rcParams["animation.codec"]` (default: `'h264'`)] The codec to use.

bitrate

[int, default: `rcParams["animation.bitrate"]` (default: `-1`)] The bitrate of the movie, in kilobits per second. Higher values means higher quality movies, but increase the file size. A value of -1 lets the underlying movie encoder select the bitrate.

extra_args

[list of str or None, optional] Extra command-line arguments passed to the underlying movie encoder. The default, None, means to use `rcParams["animation.[name-of-encoder]_args"]` for the builtin writers.

metadata

[Dict[str, str], default: {}] A dictionary of keys and values for metadata to include in the output file. Some keys that may be of use include: title, artist, genre, subject, copyright, srcform, comment.

Methods

__init__(self, *args, **kwargs) **Parameters**	
bin_path()	Return the binary path to the commandline tool used by a specific subclass.
cleanup(self)	Clean-up and collect the process used to write the movie file.
finish(self)	Finish any processing for writing the movie.
grab_frame(self, **savefig_kwargs)	Grab the image information from the figure and save as a movie frame.
isAvailable()	Return whether a MovieWriter subclass is actually available.
saving(self, fig, outfile, dpi, *args, **kwargs)	Context manager to facilitate writing the movie file.
setup(self, fig, outfile[, dpi, ...])	Setup for writing the movie file.

Attributes

args_key	
clear_temp	
exec_key	
frame_format	Format (png, jpeg, etc.) to use for saving the frames, which can be decided by the individual subclasses.
frame_size	A tuple `(width, height)` in pixels of a movie frame.
output_args	
supported_formats	

```
supported_formats = ['png', 'jpeg', 'ppm', 'tiff', 'sgi', 'bmp', 'pbm', 'raw', 'rgba']
```

matplotlib.animation.ImageMagickFileWriter

class matplotlib.animation.**ImageMagickFileWriter**(**args*, ***kwargs*)

File-based animated gif writer.

Frames are written to temporary files on disk and then stitched together at the end.

Parameters

fps

[int, default: 5] Movie frame rate (per second).

codec

[str or None, default: `rcParams["animation.codec"]` (default: `'h264'`)] The codec to use.

bitrate

[int, default: `rcParams["animation.bitrate"]` (default: `-1`)] The bitrate of the movie, in kilobits per second. Higher values means higher quality movies, but increase the file size. A value of -1 lets the underlying movie encoder select the bitrate.

extra_args

[list of str or None, optional] Extra command-line arguments passed to the underlying movie encoder. The default, None, means to use `rcParams["animation.[name-of-encoder]_args"]` for the builtin writers.

metadata

[Dict[str, str], default: {}] A dictionary of keys and values for metadata to include in the output file. Some keys that may be of use include: title, artist, genre, subject, copyright, srcform, comment.

__init__(*self*, **args*, ***kwargs*)

Parameters

fps

[int, default: 5] Movie frame rate (per second).

codec

[str or None, default: `rcParams["animation.codec"]` (default: `'h264'`)] The codec to use.

bitrate

[int, default: rcParams["animation.bitrate"] (default: -1)] The bitrate of the movie, in kilobits per second. Higher values means higher quality movies, but increase the file size. A value of -1 lets the underlying movie encoder select the bitrate.

extra_args

[list of str or None, optional] Extra command-line arguments passed to the underlying movie encoder. The default, None, means to use rcParams["animation.[name-of-encoder]_args"] for the builtin writers.

metadata

[Dict[str, str], default: {}] A dictionary of keys and values for metadata to include in the output file. Some keys that may be of use include: title, artist, genre, subject, copyright, srcform, comment.

Methods

__init__(self, *args, **kwargs)	**Parameters**
bin_path()	Return the binary path to the commandline tool used by a specific subclass.
cleanup(self)	Clean-up and collect the process used to write the movie file.
finish(self)	Finish any processing for writing the movie.
grab_frame(self, **savefig_kwargs)	Grab the image information from the figure and save as a movie frame.
isAvailable()	Return whether a MovieWriter subclass is actually available.
saving(self, fig, outfile, dpi, *args, **kwargs)	Context manager to facilitate writing the movie file.
setup(self, fig, outfile[, dpi, ...])	Setup for writing the movie file.

Attributes

args_key	
clear_temp	
delay	
exec_key	
frame_format	Format (png, jpeg, etc.) to use for saving the frames, which can be decided by the individual subclasses.
frame_size	A tuple (width, height) in pixels of a movie frame.
output_args	
supported_formats	

supported_formats = ['png', 'jpeg', 'ppm', 'tiff', 'sgi', 'bmp', 'pbm', 'raw', 'rgba']

matplotlib.animation.AVConvFileWriter

class matplotlib.animation.**AVConvFileWriter**(**args*, ***kwargs*)

File-based avconv writer.

Frames are written to temporary files on disk and then stitched together at the end.

__init__(*self*, */*, **args*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(self, /, *args, **kwargs)	Initialize self.
bin_path()	Return the binary path to the commandline tool used by a specific subclass.
cleanup(self)	Clean-up and collect the process used to write the movie file.
finish(self)	Finish any processing for writing the movie.
grab_frame(self, **savefig_kwargs)	Grab the image information from the figure and save as a movie frame.
isAvailable()	Return whether a MovieWriter subclass is actually available.
saving(self, fig, outfile, dpi, *args, **kwargs)	Context manager to facilitate writing the movie file.
setup(self, fig, outfile[, dpi, ...])	Setup for writing the movie file.

Attributes

args_key	
clear_temp	
exec_key	
frame_format	Format (png, jpeg, etc.) to use for saving the frames, which can be decided by the individual subclasses.
frame_size	A tuple `(width, height)` in pixels of a movie frame.
output_args	
supported_formats	

Fundamentally, a *MovieWriter* provides a way to grab sequential frames from the same underlying *Figure* object. The base class *MovieWriter* implements 3 methods and a context manager. The only difference between the pipe-based and file-based writers is in the arguments to their respective `setup` methods.

The `setup()` method is used to prepare the writer (possibly opening a pipe), successive calls to `grab_frame()` capture a single frame at a time and `finish()` finalizes the movie and writes the output file to disk. For example

```
moviewriter = MovieWriter(...)
moviewriter.setup(fig, 'my_movie.ext', dpi=100)
for j in range(n):
    update_figure(j)
    moviewriter.grab_frame()
moviewriter.finish()
```

If using the writer classes directly (not through *Animation.save*), it is strongly encouraged to use the `saving` context manager

```
with moviewriter.saving(fig, 'myfile.mp4', dpi=100):
    for j in range(n):
        update_figure(j)
        moviewriter.grab_frame()
```

to ensures that setup and cleanup are performed as necessary.

Examples

Frame grabbing

Use a MovieWriter directly to grab individual frames and write them to a file. This avoids any event loop integration, and thus works even with the Agg backend. This is not recommended for use in an interactive setting.

```
import numpy as np
import matplotlib
matplotlib.use("Agg")
import matplotlib.pyplot as plt
from matplotlib.animation import FFMpegWriter

# Fixing random state for reproducibility
np.random.seed(19680801)


metadata = dict(title='Movie Test', artist='Matplotlib',
                comment='Movie support!')
writer = FFMpegWriter(fps=15, metadata=metadata)

fig = plt.figure()
l, = plt.plot([], [], 'k-o')

plt.xlim(-5, 5)
plt.ylim(-5, 5)

x0, y0 = 0, 0

with writer.saving(fig, "writer_test.mp4", 100):
    for i in range(100):
        x0 += 0.1 * np.random.randn()
        y0 += 0.1 * np.random.randn()
        l.set_data(x0, y0)
        writer.grab_frame()
```

18.3.3 Helper Classes

Animation Base Classes

Animation	A base class for Animations.
TimedAnimation	*Animation* subclass for time-based animation.

matplotlib.animation.TimedAnimation

class matplotlib.animation.**TimedAnimation**(*fig*, *interval=200*, *repeat_delay=0*, *repeat=True*, *event_source=None*, **args*, ***kwargs*)

Animation subclass for time-based animation.

A new frame is drawn every *interval* milliseconds.

Parameters

fig

[*Figure*] The figure object used to get needed events, such as draw or resize.

interval

[int, default: 200] Delay between frames in milliseconds.

repeat_delay

[int, default: 0] The delay in milliseconds between consecutive animation runs, if *repeat* is True.

repeat

[bool, default: True] Whether the animation repeats when the sequence of frames is completed.

blit

[bool, default: False] Whether blitting is used to optimize drawing.

`__init__`(*self*, *fig*, *interval=200*, *repeat_delay=0*, *repeat=True*, *event_source=None*, **args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(self, fig[, interval, ...])	Initialize self.
new_frame_seq(self)	Return a new sequence of frame information.
new_saved_frame_seq(self)	Return a new sequence of saved/cached frame information.
save(self, filename[, writer, fps, dpi, ...])	Save the animation as a movie file by drawing every frame.
to_html5_video(self[, embed_limit])	Convert the animation to an HTML5 <video> tag.
to_jshtml(self[, fps, embed_frames, ...])	Generate HTML representation of the animation

Writer Registry

A module-level registry is provided to map between the name of the writer and the class to allow a string to be passed to *Animation.save* instead of a writer instance.

MovieWriterRegistry	Registry of available writer classes by human readable name.

matplotlib.animation.MovieWriterRegistry

class matplotlib.animation.**MovieWriterRegistry**

Registry of available writer classes by human readable name.

__init__(*self*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(self)	Initialize self.
ensure_not_dirty(self)	[*Deprecated*] If dirty, reasks the writers if they are available
is_available(self, name)	Check if given writer is available by name.
list(self)	Get a list of available MovieWriters.
register(self, name)	Decorator for registering a class under a name.
reset_available_writers(self)	[*Deprecated*] Reset the available state of all registered writers
set_dirty(self)	[*Deprecated*] Sets a flag to re-setup the writers.

Attributes

avail

property avail

ensure_not_dirty(*self*)

[*Deprecated*] If dirty, reasks the writers if they are available

Notes

Deprecated since version 3.2.

`is_available`(*self*, *name*)
Check if given writer is available by name.

Parameters

name

[str]

Returns

bool

`list`(*self*)
Get a list of available MovieWriters.

`register`(*self*, *name*)
Decorator for registering a class under a name.

Example use:

```
@registry.register(name)
class Foo:
    pass
```

`reset_available_writers`(*self*)
[*Deprecated*] Reset the available state of all registered writers

Notes

Deprecated since version 3.2.

`set_dirty`(*self*)
[*Deprecated*] Sets a flag to re-setup the writers.

Notes

Deprecated since version 3.2.

Writer Base Classes

To reduce code duplication base classes

AbstractMovieWriter	Abstract base class for writing movies.
MovieWriter	Base class for writing movies.
FileMovieWriter	*MovieWriter* for writing to individual files and stitching at the end.

matplotlib.animation.AbstractMovieWriter

class matplotlib.animation.**AbstractMovieWriter**(*fps=5*, *metadata=None*, *codec=None*, *bitrate=None*)

Abstract base class for writing movies. Fundamentally, what a MovieWriter does is provide is a way to grab frames by calling grab_frame().

setup() is called to start the process and finish() is called afterwards.

This class is set up to provide for writing movie frame data to a pipe. saving() is provided as a context manager to facilitate this process as:

```
with moviewriter.saving(fig, outfile='myfile.mp4', dpi=100):
    # Iterate over frames
    moviewriter.grab_frame(**savefig_kwargs)
```

The use of the context manager ensures that setup() and finish() are performed as necessary.

An instance of a concrete subclass of this class can be given as the `writer` argument of *Animation.save()*.

__init__(*self*, *fps=5*, *metadata=None*, *codec=None*, *bitrate=None*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(self[, fps, metadata, codec, bitrate])	Initialize self.
finish(self)	Finish any processing for writing the movie.
grab_frame(self, **savefig_kwargs)	Grab the image information from the figure and save as a movie frame.
saving(self, fig, outfile, dpi, *args, **kwargs)	Context manager to facilitate writing the movie file.
setup(self, fig, outfile[, dpi])	Setup for writing the movie file.

Attributes

frame_size	A tuple `(width, height)` in pixels of a movie frame.

`abstract finish(self)`
Finish any processing for writing the movie.

`property frame_size`
A tuple `(width, height)` in pixels of a movie frame.

`abstract grab_frame(self, **savefig_kwargs)`
Grab the image information from the figure and save as a movie frame.

All keyword arguments in *savefig_kwargs* are passed on to the *savefig* call that saves the figure.

`saving(self, fig, outfile, dpi, *args, **kwargs)`
Context manager to facilitate writing the movie file.

`*args, **kw` are any parameters that should be passed to *setup*.

`abstract setup(self, fig, outfile, dpi=None)`
Setup for writing the movie file.

Parameters

fig

[*Figure*] The figure object that contains the information for frames.

outfile

[str] The filename of the resulting movie file.

dpi

[float, default: `fig.dpi`] The DPI (or resolution) for the file. This controls the size in pixels of the resulting movie file.

matplotlib.animation.MovieWriter

`class matplotlib.animation.MovieWriter(fps=5, codec=None, bitrate=None, extra_args=None, metadata=None)`
Base class for writing movies.

This is a base class for MovieWriter subclasses that write a movie frame data to a pipe. You cannot instantiate this class directly. See examples for how to use its subclasses.

Attributes

frame_format

[str] The format used in writing frame data, defaults to 'rgba'.

fig

[*Figure*] The figure to capture data from. This must be provided by the sub-classes.

Parameters

fps

[int, default: 5] Movie frame rate (per second).

codec

[str or None, default: `rcParams["animation.codec"]` (default: `'h264'`)] The codec to use.

bitrate

[int, default: `rcParams["animation.bitrate"]` (default: `-1`)] The bitrate of the movie, in kilobits per second. Higher values means higher quality movies, but increase the file size. A value of -1 lets the underlying movie encoder select the bitrate.

extra_args

[list of str or None, optional] Extra command-line arguments passed to the underlying movie encoder. The default, None, means to use `rcParams["animation.[name-of-encoder]_args"]` for the builtin writers.

metadata

[Dict[str, str], default: {}] A dictionary of keys and values for metadata to include in the output file. Some keys that may be of use include: title, artist, genre, subject, copyright, srcform, comment.

`__init__`(*self*, *fps=5*, *codec=None*, *bitrate=None*, *extra_args=None*, *metadata=None*)

Parameters

fps

[int, default: 5] Movie frame rate (per second).

codec

[str or None, default: `rcParams["animation.codec"]` (default: `'h264'`)] The codec to use.

bitrate

[int, default: `rcParams["animation.bitrate"]` (default: `-1`)] The bitrate of the movie, in kilobits per second. Higher values means higher quality movies, but increase the file size. A value of -1 lets the underlying movie encoder select the bitrate.

extra_args

[list of str or None, optional] Extra command-line arguments passed to the underlying movie encoder. The default, None, means to use `rcParams["animation.[name-of-encoder]_args"]` for the builtin writers.

metadata

[Dict[str, str], default: {}] A dictionary of keys and values for metadata to include in the output file. Some keys that may be of use include: title, artist, genre, subject, copyright, srcform, comment.

Methods

__init__(self[, fps, codec, bitrate, ...])	**Parameters**
bin_path()	Return the binary path to the commandline tool used by a specific subclass.
cleanup(self)	Clean-up and collect the process used to write the movie file.
finish(self)	Finish any processing for writing the movie.
grab_frame(self, **savefig_kwargs)	Grab the image information from the figure and save as a movie frame.
isAvailable()	Return whether a MovieWriter subclass is actually available.
saving(self, fig, outfile, dpi, *args, **kwargs)	Context manager to facilitate writing the movie file.
setup(self, fig, outfile[, dpi])	Setup for writing the movie file.

Attributes

args_key	
exec_key	
frame_size	A tuple (width, height) in pixels of a movie frame.

property args_key

classmethod bin_path()
Return the binary path to the commandline tool used by a specific subclass. This is a class method so that the tool can be looked for before making a particular MovieWriter subclass available.

cleanup(*self*)
Clean-up and collect the process used to write the movie file.

property exec_key

finish(*self*)
Finish any processing for writing the movie.

grab_frame(*self*, ***savefig_kwargs*)
Grab the image information from the figure and save as a movie frame.

All keyword arguments in *savefig_kwargs* are passed on to the *savefig* call that saves the figure.

classmethod isAvailable()
Return whether a MovieWriter subclass is actually available.

setup(*self*, *fig*, *outfile*, *dpi=None*)
Setup for writing the movie file.

Parameters

fig

[*Figure*] The figure object that contains the information for frames.

outfile

[str] The filename of the resulting movie file.

dpi

[float, default: fig.dpi] The DPI (or resolution) for the file. This controls the size in pixels of the resulting movie file.

matplotlib.animation.FileMovieWriter

class matplotlib.animation.**FileMovieWriter**(**args*, ***kwargs*)

MovieWriter for writing to individual files and stitching at the end.

This must be sub-classed to be useful.

Parameters

fps

[int, default: 5] Movie frame rate (per second).

codec

[str or None, default: `rcParams["animation.codec"]` (default: `'h264'`)] The codec to use.

bitrate

[int, default: `rcParams["animation.bitrate"]` (default: `-1`)] The bitrate of the movie, in kilobits per second. Higher values means higher quality movies, but increase the file size. A value of -1 lets the underlying movie encoder select the bitrate.

extra_args

[list of str or None, optional] Extra command-line arguments passed to the underlying movie encoder. The default, None, means to use `rcParams["animation.[name-of-encoder]_args"]` for the builtin writers.

metadata

[Dict[str, str], default: {}] A dictionary of keys and values for metadata to include in the output file. Some keys that may be of use include: title, artist, genre, subject, copyright, srcform, comment.

__init__(*self*, **args*, ***kwargs*)

Parameters

fps

[int, default: 5] Movie frame rate (per second).

codec

[str or None, default: `rcParams["animation.codec"]` (default: `'h264'`)] The codec to use.

bitrate

[int, default: `rcParams["animation.bitrate"]` (default: `-1`)] The bitrate of the movie, in kilobits per second. Higher values

means higher quality movies, but increase the file size. A value of -1 lets the underlying movie encoder select the bitrate.

extra_args

[list of str or None, optional] Extra command-line arguments passed to the underlying movie encoder. The default, None, means to use `rcParams["animation.[name-of-encoder]_args"]` for the builtin writers.

metadata

[Dict[str, str], default: {}] A dictionary of keys and values for metadata to include in the output file. Some keys that may be of use include: title, artist, genre, subject, copyright, srcform, comment.

Methods

__init__(self, *args, **kwargs) **Parameters**	
`bin_path`()	Return the binary path to the commandline tool used by a specific subclass.
cleanup(self)	Clean-up and collect the process used to write the movie file.
finish(self)	Finish any processing for writing the movie.
grab_frame(self, **savefig_kwargs)	Grab the image information from the figure and save as a movie frame.
`isAvailable`()	Return whether a MovieWriter subclass is actually available.
`saving`(self, fig, outfile, dpi, *args, **kwargs)	Context manager to facilitate writing the movie file.
setup(self, fig, outfile[, dpi, ...])	Setup for writing the movie file.

Attributes

`args_key`
clear_temp
`exec_key`

continues on next page

Table 36 – continued from previous page

frame_format	Format (png, jpeg, etc.) to use for saving the frames, which can be decided by the individual subclasses.
`frame_size`	A tuple `(width, height)` in pixels of a movie frame.

`cleanup`(*self*)
Clean-up and collect the process used to write the movie file.

`property clear_temp`

`finish`(*self*)
Finish any processing for writing the movie.

`property frame_format`
Format (png, jpeg, etc.) to use for saving the frames, which can be decided by the individual subclasses.

`grab_frame`(*self*, ***savefig_kwargs*)
Grab the image information from the figure and save as a movie frame.

All keyword arguments in *savefig_kwargs* are passed on to the *savefig* call that saves the figure.

`setup`(*self*, *fig*, *outfile*, *dpi=None*, *frame_prefix=None*, *clear_temp=<deprecated parameter>*)
Setup for writing the movie file.

Parameters

fig

[*Figure*] The figure to grab the rendered frames from.

outfile

[str] The filename of the resulting movie file.

dpi

[float, optional] The dpi of the output file. This, with the figure size, controls the size in pixels of the resulting movie file. Default is `fig.dpi`.

frame_prefix

[str, optional] The filename prefix to use for temporary files. If None (the default), files are written to a temporary directory which is deleted by *cleanup* (regardless of the value of *clear_temp*).

clear_temp

[bool, optional] If the temporary files should be deleted after stitching the final result. Setting this to `False` can be useful for debugging. Defaults to `True`.

and mixins

AVConvBase	[*Deprecated*] Mixin class for avconv output.
FFMpegBase	Mixin class for FFMpeg output.
ImageMagickBase	Mixin class for ImageMagick output.

matplotlib.animation.AVConvBase

class matplotlib.animation.**AVConvBase**(**args*, ***kwargs*)

[*Deprecated*] Mixin class for avconv output.

To be useful this must be multiply-inherited from with a `MovieWriterBase` subclass.

Notes

Deprecated since version 3.3.

__init__(*self*, */*, **args*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(self, /, *args, **kwargs)	Initialize self.
isAvailable()	Return whether a MovieWriter subclass is actually available.

Attributes

output_args

classmethod isAvailable()

Return whether a MovieWriter subclass is actually available.

matplotlib.animation.FFMpegBase

class matplotlib.animation.FFMpegBase

Mixin class for FFMpeg output.

To be useful this must be multiply-inherited from with a `MovieWriterBase` subclass.

__init__(*self*, /, **args*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(self, /, *args, **kwargs)	Initialize self.
isAvailable()	

Attributes

output_args

classmethod isAvailable()

property output_args

matplotlib.animation.ImageMagickBase

class matplotlib.animation.ImageMagickBase

Mixin class for ImageMagick output.

To be useful this must be multiply-inherited from with a `MovieWriterBase` subclass.

__init__(*self*, /, **args*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(self, /, *args, **kwargs)	Initialize self.
bin_path()	
isAvailable()	

Attributes

delay
output_args

classmethod bin_path()

property delay

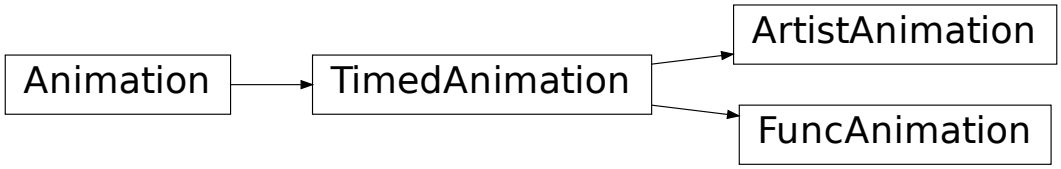
classmethod isAvailable()

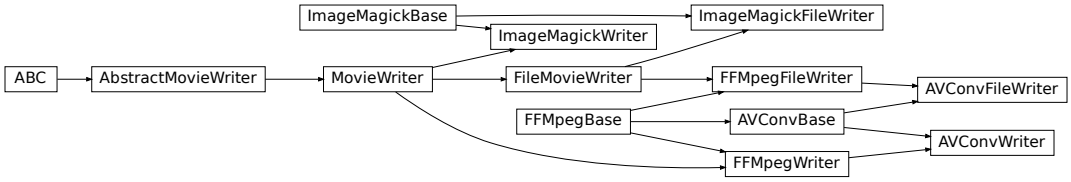
property output_args

are provided.

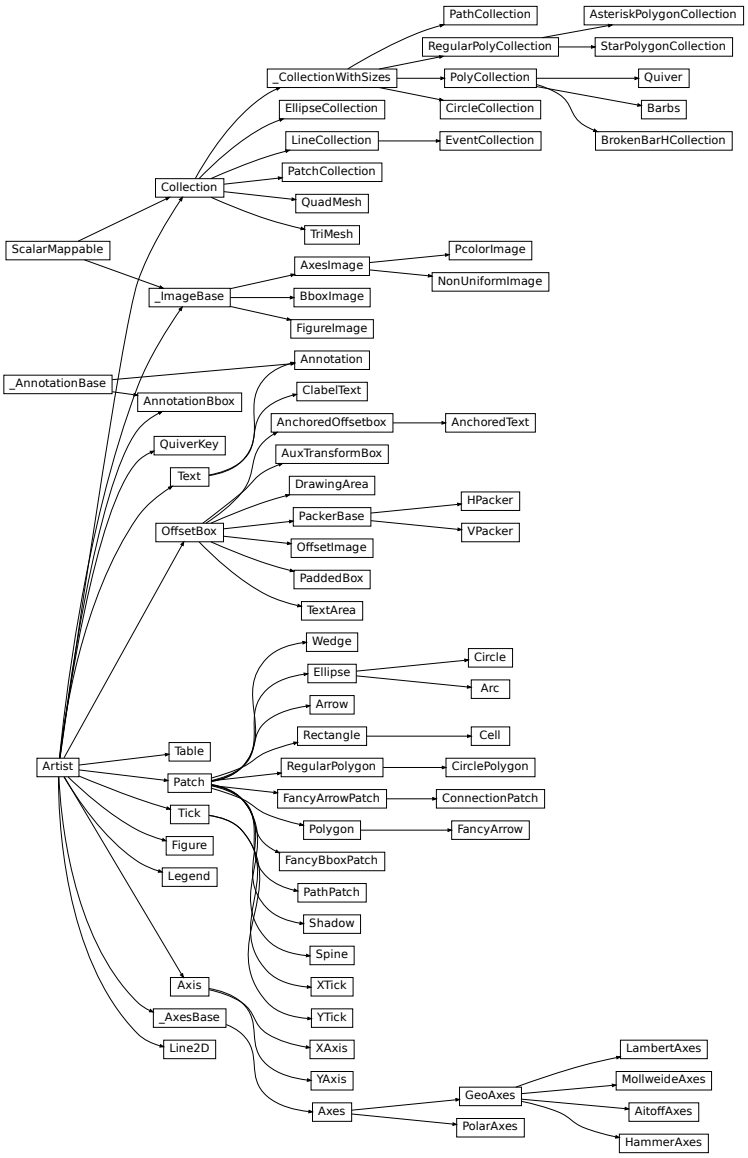
See the source code for how to easily implement new *MovieWriter* classes.

18.3.4 Inheritance Diagrams





18.4 matplotlib.artist



18.4.1 `Artist` class

class matplotlib.artist.**Artist**

Abstract base class for objects that render into a FigureCanvas.

Typically, all visible elements in a figure are subclasses of Artist.

Interactive

Artist.add_callback	Add a callback function that will be called whenever one of the *Artist*'s properties changes.
Artist.remove_callback	Remove a callback based on its observer id.
Artist.pchanged	Call all of the registered callbacks.
Artist.get_cursor_data	Return the cursor data for a given event.
Artist.format_cursor_data	Return a string representation of *data*.
Artist.mouseover	If this property is set to *True*, the artist will be queried for custom context information when the mouse cursor moves over it.
Artist.contains	Test whether the artist contains the mouse event.
Artist.set_contains	[*Deprecated*] Define a custom contains test for the artist.
Artist.get_contains	[*Deprecated*] Return the custom contains function of the artist if set, or *None*.
Artist.pick	Process a pick event.
Artist.pickable	Return whether the artist is pickable.
Artist.set_picker	Define the picking behavior of the artist.
Artist.get_picker	Return the picking behavior of the artist.

matplotlib.artist.Artist.add_callback

Artist.add_callback(*self*, *func*)

Add a callback function that will be called whenever one of the *Artist*'s properties changes.

Parameters

func

[callable] The callback function. It must have the signature:

```
def func(artist: Artist) -> Any
```

where *artist* is the calling *Artist*. Return values may exist but are ignored.

Returns

int

The observer id associated with the callback. This id can be used for removing the callback with *remove_callback* later.

See also:

remove_callback

matplotlib.artist.Artist.remove_callback

Artist.**remove_callback**(*self*, *oid*)

Remove a callback based on its observer id.

See also:

add_callback

matplotlib.artist.Artist.pchanged

Artist.**pchanged**(*self*)

Call all of the registered callbacks.

This function is triggered internally when a property is changed.

See also:

add_callback

remove_callback

matplotlib.artist.Artist.get_cursor_data

Artist.get_cursor_data(*self*, *event*)

Return the cursor data for a given event.

Note: This method is intended to be overridden by artist subclasses. As an end-user of Matplotlib you will most likely not call this method yourself.

Cursor data can be used by Artists to provide additional context information for a given event. The default implementation just returns *None*.

Subclasses can override the method and return arbitrary data. However, when doing so, they must ensure that *format_cursor_data* can convert the data to a string representation.

The only current use case is displaying the z-value of an *AxesImage* in the status bar of a plot window, while moving the mouse.

Parameters

event

[*matplotlib.backend_bases.MouseEvent*]

See also:

format_cursor_data

matplotlib.artist.Artist.format_cursor_data

Artist.format_cursor_data(*self*, *data*)

Return a string representation of *data*.

Note: This method is intended to be overridden by artist subclasses. As an end-user of Matplotlib you will most likely not call this method yourself.

The default implementation converts ints and floats and arrays of ints and floats into a comma-separated string enclosed in square brackets.

See also:

get_cursor_data

matplotlib.artist.Artist.mouseover

property Artist.mouseover

If this property is set to *True*, the artist will be queried for custom context information when the mouse cursor moves over it.

See also *get_cursor_data()*, *ToolCursorPosition* and *NavigationToolbar2*.

matplotlib.artist.Artist.contains

Artist.contains(*self*, *mouseevent*)

Test whether the artist contains the mouse event.

Parameters

mouseevent

[*matplotlib.backend_bases.MouseEvent*]

Returns

contains

[bool] Whether any values are within the radius.

details

[dict] An artist-specific dictionary of details of the event context, such as which points are contained in the pick radius. See the individual Artist subclasses for details.

matplotlib.artist.Artist.set_contains

Artist.set_contains(*self*, *picker*)

[*Deprecated*] Define a custom contains test for the artist.

The provided callable replaces the default *contains* method of the artist.

Parameters

picker

[callable] A custom picker function to evaluate if an event is within the artist. The function must have the signature:

```
def contains(artist: Artist, event: MouseEvent) -> bool, dict
```

that returns:

- a bool indicating if the event is within the artist

- a dict of additional information. The dict should at least return the same information as the default `contains()` implementation of the respective artist, but may provide additional information.

Notes

Deprecated since version 3.3.

matplotlib.artist.Artist.get_contains

`Artist.get_contains(`*self*`)`

[*Deprecated*] Return the custom contains function of the artist if set, or *None*.

See also:

set_contains

Notes

Deprecated since version 3.3.

matplotlib.artist.Artist.pick

`Artist.pick(`*self, mouseevent*`)`

Process a pick event.

Each child artist will fire a pick event if *mouseevent* is over the artist and the artist has picker set.

See also:

set_picker, *get_picker*, *pickable*

matplotlib.artist.Artist.pickable

`Artist.pickable(`*self*`)`

Return whether the artist is pickable.

See also:

set_picker, *get_picker*, *pick*

matplotlib.artist.Artist.set_picker

Artist.set_picker(*self*, *picker*)

Define the picking behavior of the artist.

Parameters

picker

[None or bool or callable] This can be one of the following:

- *None*: Picking is disabled for this artist (default).
- A boolean: If *True* then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist.
- A function: If picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

to determine the hit test. if the mouse event is over the artist, return *hit=True* and props is a dictionary of properties you want added to the PickEvent attributes.

- *deprecated*: For *Line2D* only, *picker* can also be a float that sets the tolerance for checking whether an event occurred "on" the line; this is deprecated. Use *Line2D.set_pickradius* instead.

matplotlib.artist.Artist.get_picker

Artist.get_picker(*self*)

Return the picking behavior of the artist.

The possible values are described in *set_picker*.

See also:

set_picker, *pickable*, *pick*

Clipping

Artist.set_clip_on	Set whether the artist uses clipping.
Artist.get_clip_on	Return whether the artist uses clipping.
Artist.set_clip_box	Set the artist's clip *Bbox*.
Artist.get_clip_box	Return the clipbox.
Artist.set_clip_path	Set the artist's clip path.
Artist.get_clip_path	Return the clip path.

matplotlib.artist.Artist.set_clip_on

Artist.set_clip_on(*self*, *b*)

Set whether the artist uses clipping.

When False artists will be visible outside of the axes which can lead to unexpected results.

Parameters

b

[bool]

matplotlib.artist.Artist.get_clip_on

Artist.get_clip_on(*self*)

Return whether the artist uses clipping.

matplotlib.artist.Artist.set_clip_box

Artist.set_clip_box(*self*, *clipbox*)

Set the artist's clip *Bbox*.

Parameters

clipbox

[*Bbox*]

matplotlib.artist.Artist.get_clip_box

Artist.get_clip_box(*self*)

Return the clipbox.

matplotlib.artist.Artist.set_clip_path

Artist.set_clip_path(*self*, *path*, *transform=None*)

Set the artist's clip path.

Parameters

path

[*Patch* or *Path* or *TransformedPath* or None] The clip path. If given a *Path*, *transform* must be provided as well. If *None*, a previously set clip path is removed.

transform

[*Transform*, optional] Only used if *path* is a *Path*, in which case the given *Path* is converted to a *TransformedPath* using *transform*.

Notes

For efficiency, if *path* is a *Rectangle* this method will set the clipping box to the corresponding rectangle and set the clipping path to `None`.

For technical reasons (support of *set*), a tuple (*path*, *transform*) is also accepted as a single positional parameter.

matplotlib.artist.Artist.get_clip_path

`Artist.get_clip_path`(*self*)

Return the clip path.

Bulk Properties

Artist.update	Update this artist's properties from the dict *props*.
Artist.update_from	Copy properties from *other* to *self*.
Artist.properties	Return a dictionary of all the properties of the artist.
Artist.set	A property batch setter.

matplotlib.artist.Artist.update

`Artist.update`(*self*, *props*)

Update this artist's properties from the dict *props*.

Parameters

props

[dict]

matplotlib.artist.Artist.update_from

Artist.update_from(*self*, *other*)
Copy properties from *other* to *self*.

matplotlib.artist.Artist.properties

Artist.properties(*self*)
Return a dictionary of all the properties of the artist.

matplotlib.artist.Artist.set

Artist.set(*self*, ***kwargs*)
A property batch setter. Pass *kwargs* to set properties.

Drawing

Artist.draw	Draw the Artist (and its children) using the given renderer.
Artist.set_animated	Set the artist's animation state.
Artist.get_animated	Return whether the artist is animated.
Artist.set_alpha	Set the alpha value used for blending - not supported on all backends.
Artist.get_alpha	Return the alpha value used for blending - not supported on all backends.
Artist.set_snap	Set the snapping behavior.
Artist.get_snap	Return the snap setting.
Artist.set_visible	Set the artist's visibility.
Artist.get_visible	Return the visibility.
Artist.zorder	
Artist.set_zorder	Set the zorder for the artist.
Artist.get_zorder	Return the artist's zorder.
Artist.set_agg_filter	Set the agg filter.
Artist.set_sketch_params	Sets the sketch parameters.
Artist.get_sketch_params	Return the sketch parameters for the artist.
Artist.set_rasterized	Force rasterized (bitmap) drawing in vector backend output.
Artist.get_rasterized	Return whether the artist is to be rasterized.
Artist.set_path_effects	Set the path effects.
Artist.get_path_effects	

continues on next page

Table 47 – continued from previous page

Artist.get_agg_filter	Return filter function to be used for agg filter.
Artist.get_window_extent	Get the axes bounding box in display space.
Artist.get_transformed_clip_path_and_affine	Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

matplotlib.artist.Artist.draw

Artist.**draw**(*self*, *renderer*, **args*, ***kwargs*)

Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

Parameters

renderer

[*RendererBase* subclass.]

Notes

This method is overridden in the Artist subclasses.

matplotlib.artist.Artist.set_animated

Artist.**set_animated**(*self*, *b*)

Set the artist's animation state.

Parameters

b

[bool]

matplotlib.artist.Artist.get_animated

Artist.get_animated(*self*)
Return whether the artist is animated.

matplotlib.artist.Artist.set_alpha

Artist.set_alpha(*self*, *alpha*)
Set the alpha value used for blending - not supported on all backends.

Parameters

alpha

[float or None]

matplotlib.artist.Artist.get_alpha

Artist.get_alpha(*self*)
Return the alpha value used for blending - not supported on all backends.

matplotlib.artist.Artist.set_snap

Artist.set_snap(*self*, *snap*)
Set the snapping behavior.

Snapping aligns positions with the pixel grid, which results in clearer images. For example, if a black line of 1px width was defined at a position in between two pixels, the resulting image would contain the interpolated value of that line in the pixel grid, which would be a grey value on both adjacent pixel positions. In contrast, snapping will move the line to the nearest integer pixel value, so that the resulting image will really contain a 1px wide black line.

Snapping is currently only supported by the Agg and MacOSX backends.

Parameters

snap

[bool or None] Possible values:

- *True*: Snap vertices to the nearest pixel center.
- *False*: Do not modify vertex positions.
- *None*: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center.

matplotlib.artist.Artist.get_snap

Artist.get_snap(*self*)
Return the snap setting.

See *set_snap* for details.

matplotlib.artist.Artist.set_visible

Artist.set_visible(*self*, *b*)
Set the artist's visibility.

Parameters

b

[bool]

matplotlib.artist.Artist.get_visible

Artist.get_visible(*self*)
Return the visibility.

matplotlib.artist.Artist.zorder

Artist.zorder = 0

matplotlib.artist.Artist.set_zorder

Artist.set_zorder(*self*, *level*)
Set the zorder for the artist. Artists with lower zorder values are drawn first.

Parameters

level

[float]

matplotlib.artist.Artist.get_zorder

Artist.get_zorder(*self*)
Return the artist's zorder.

matplotlib.artist.Artist.set_agg_filter

Artist.set_agg_filter(*self*, *filter_func*)
Set the agg filter.

Parameters

filter_func

[callable] A filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array.

matplotlib.artist.Artist.set_sketch_params

Artist.set_sketch_params(*self*, *scale=None*, *length=None*, *randomness=None*)
Sets the sketch parameters.

Parameters

scale

[float, optional] The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is `None`, or not provided, no sketch filter will be provided.

length

[float, optional] The length of the wiggle along the line, in pixels (default 128.0)

randomness

[float, optional] The scale factor by which the length is shrunken or expanded (default 16.0)

matplotlib.artist.Artist.get_sketch_params

Artist.get_sketch_params(*self*)

Return the sketch parameters for the artist.

Returns

tuple or None

A 3-tuple with the following elements:

- *scale*: The amplitude of the wiggle perpendicular to the source line.
- *length*: The length of the wiggle along the line.
- *randomness*: The scale factor by which the length is shrunken or expanded.

Returns *None* if no sketch parameters were set.

matplotlib.artist.Artist.set_rasterized

Artist.set_rasterized(*self*, *rasterized*)

Force rasterized (bitmap) drawing in vector backend output.

Defaults to None, which implies the backend's default behavior.

Parameters

rasterized

[bool or None]

matplotlib.artist.Artist.get_rasterized

Artist.get_rasterized(*self*)

Return whether the artist is to be rasterized.

matplotlib.artist.Artist.set_path_effects

Artist.set_path_effects(*self*, *path_effects*)

Set the path effects.

Parameters

path_effects

[*AbstractPathEffect*]

matplotlib.artist.Artist.get_path_effects

Artist.get_path_effects(*self*)

matplotlib.artist.Artist.get_agg_filter

Artist.get_agg_filter(*self*)
Return filter function to be used for agg filter.

matplotlib.artist.Artist.get_window_extent

Artist.get_window_extent(*self*, *renderer*)
Get the axes bounding box in display space.

The bounding box' width and height are nonnegative.

Subclasses should override for inclusion in the bounding box "tight" calculation. Default is to return an empty bounding box at 0, 0.

Be careful when using this function, the results will not update if the artist window extent of the artist changes. The extent can change due to any changes in the transform stack, such as changing the axes limits, the figure size, or the canvas used (as is done when saving a figure). This can lead to unexpected behavior where interactive figures will look fine on the screen, but will save incorrectly.

matplotlib.artist.Artist.get_transformed_clip_path_and_affine

Artist.get_transformed_clip_path_and_affine(*self*)
Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

Figure and Axes

Artist.remove	Remove the artist from the figure if possible.
Artist.axes	The *Axes* instance the artist resides in, or *None*.
Artist.set_figure	Set the *Figure* instance the artist belongs to.
Artist.get_figure	Return the *Figure* instance the artist belongs to.

matplotlib.artist.Artist.remove

Artist.**remove**(*self*)

Remove the artist from the figure if possible.

The effect will not be visible until the figure is redrawn, e.g., with *FigureCanvasBase.draw_idle*. Call *relim* to update the axes limits if desired.

Note: *relim* will not see collections even if the collection was added to the axes with *autolim* = True.

Note: there is no support for removing the artist's legend entry.

matplotlib.artist.Artist.axes

property Artist.**axes**

The *Axes* instance the artist resides in, or *None*.

matplotlib.artist.Artist.set_figure

Artist.**set_figure**(*self*, *fig*)

Set the *Figure* instance the artist belongs to.

Parameters

fig

[*Figure*]

matplotlib.artist.Artist.get_figure

Artist.**get_figure**(*self*)

Return the *Figure* instance the artist belongs to.

Children

Artist.get_children	Return a list of the child *Artist*s of this *Artist*.
Artist.findobj	Find artist objects.

matplotlib.artist.Artist.get_children

Artist.get_children(*self*)
Return a list of the child *Artist*s of this *Artist*.

matplotlib.artist.Artist.findobj

Artist.findobj(*self, match=None, include_self=True*)
Find artist objects.

Recursively find all *Artist* instances contained in the artist.

Parameters

match

A filter criterion for the matches. This can be

- *None*: Return all objects contained in artist.
- A function with signature `def match(artist: Artist) -> bool`. The result will only contain artists for which the function returns *True*.
- A class instance: e.g., *Line2D*. The result will only contain artists of this class or its subclasses (`isinstance` check).

include_self

[bool] Include *self* in the list to be checked for a match.

Returns

list of *Artist*

Transform

Artist.set_transform	Set the artist transform.
Artist.get_transform	Return the *Transform* instance used by this artist.
Artist.is_transform_set	Return whether the Artist has an explicitly set transform.

matplotlib.artist.Artist.set_transform

Artist.set_transform(*self*, *t*)
Set the artist transform.

Parameters

t

[*Transform*]

matplotlib.artist.Artist.get_transform

Artist.get_transform(*self*)
Return the *Transform* instance used by this artist.

matplotlib.artist.Artist.is_transform_set

Artist.is_transform_set(*self*)
Return whether the Artist has an explicitly set transform.

This is *True* after *set_transform* has been called.

Units

Artist.convert_xunits	Convert *x* using the unit type of the xaxis.
Artist.convert_yunits	Convert *y* using the unit type of the yaxis.
Artist.have_units	Return *True* if units are set on any axis.

matplotlib.artist.Artist.convert_xunits

Artist.convert_xunits(*self*, *x*)
Convert *x* using the unit type of the xaxis.

If the artist is not in contained in an Axes or if the xaxis does not have units, *x* itself is returned.

matplotlib.artist.Artist.convert_yunits

Artist.**convert_yunits**(*self*, *y*)

Convert *y* using the unit type of the yaxis.

If the artist is not in contained in an Axes or if the yaxis does not have units, *y* itself is returned.

matplotlib.artist.Artist.have_units

Artist.**have_units**(*self*)

Return *True* if units are set on any axis.

Metadata

Artist.set_gid	Set the (group) id for the artist.
Artist.get_gid	Return the group id.
Artist.set_label	Set a label that will be displayed in the legend.
Artist.get_label	Return the label used for this artist in the legend.
Artist.set_url	Set the url for the artist.
Artist.get_url	Return the url.

matplotlib.artist.Artist.set_gid

Artist.**set_gid**(*self*, *gid*)

Set the (group) id for the artist.

Parameters

gid

[str]

matplotlib.artist.Artist.get_gid

Artist.get_gid(*self*)
Return the group id.

matplotlib.artist.Artist.set_label

Artist.set_label(*self*, *s*)
Set a label that will be displayed in the legend.

Parameters

s

[object] *s* will be converted to a string by calling `str`.

matplotlib.artist.Artist.get_label

Artist.get_label(*self*)
Return the label used for this artist in the legend.

matplotlib.artist.Artist.set_url

Artist.set_url(*self*, *url*)
Set the url for the artist.

Parameters

url

[str]

matplotlib.artist.Artist.get_url

Artist.get_url(*self*)
Return the url.

Miscellaneous

Artist.sticky_edges	x and y sticky edge lists for autoscaling.
Artist.set_in_layout	Set if artist is to be included in layout calculations, E.g.
Artist.get_in_layout	Return boolean flag, True if artist is included in layout calculations.
Artist.stale	Whether the artist is 'stale' and needs to be re-drawn for the output to match the internal state of the artist.

matplotlib.artist.Artist.sticky_edges

property Artist.**sticky_edges**

x and y sticky edge lists for autoscaling.

When performing autoscaling, if a data limit coincides with a value in the corresponding sticky_edges list, then no margin will be added--the view limit "sticks" to the edge. A typical use case is histograms, where one usually expects no margin on the bottom edge (0) of the histogram.

This attribute cannot be assigned to; however, the x and y lists can be modified in place as needed.

Examples

```
>>> artist.sticky_edges.x[:] = (xmin, xmax)
>>> artist.sticky_edges.y[:] = (ymin, ymax)
```

matplotlib.artist.Artist.set_in_layout

Artist.**set_in_layout**(*self*, *in_layout*)

Set if artist is to be included in layout calculations, E.g. *Constrained Layout Guide*, *Figure.tight_layout()*, and `fig.savefig(fname, bbox_inches='tight')`.

Parameters

in_layout

[bool]

matplotlib.artist.Artist.get_in_layout

Artist.get_in_layout(*self*)

Return boolean flag, `True` if artist is included in layout calculations.

E.g. *Constrained Layout Guide*, *`Figure.tight_layout()`*, and `fig.savefig(fname, bbox_inches='tight')`.

matplotlib.artist.Artist.stale

property Artist.stale

Whether the artist is 'stale' and needs to be re-drawn for the output to match the internal state of the artist.

18.4.2 Functions

`allow_rasterization`	Decorator for Artist.draw method.
`get`	Return the value of an object's *property*, or print all of them.
`getp`	Return the value of an object's *property*, or print all of them.
`setp`	Set a property on an artist object.
`kwdoc`	Inspect an *`Artist`* class (using *`ArtistInspector`*) and return information about its settable properties and their current values.
`ArtistInspector`	A helper class to inspect an *`Artist`* and return information about its settable properties and their current values.

matplotlib.artist.allow_rasterization

matplotlib.artist.allow_rasterization(*draw*)

Decorator for Artist.draw method. Provides routines that run before and after the draw call. The before and after functions are useful for changing artist-dependent renderer attributes or making other setup function calls, such as starting and flushing a mixed-mode renderer.

matplotlib.artist.get

matplotlib.artist.get(*obj*, *property=None*)

Return the value of an object's *property*, or print all of them.

Parameters

obj

[*Artist*] The queried artist; e.g., a *Line2D*, a *Text*, or an *Axes*.

property

[str or None, default: None] If *property* is 'somename', this function returns `obj.get_somename()`.

If is is None (or unset), it *prints* all gettable properties from *obj*. Many properties have aliases for shorter typing, e.g. 'lw' is an alias for 'linewidth'. In the output, aliases and full property names will be listed as:

property or alias = value

e.g.:

linewidth or lw = 2

matplotlib.artist.getp

matplotlib.artist.getp(*obj*, *property=None*)

Return the value of an object's *property*, or print all of them.

Parameters

obj

[*Artist*] The queried artist; e.g., a *Line2D*, a *Text*, or an *Axes*.

property

[str or None, default: None] If *property* is 'somename', this function returns `obj.get_somename()`.

If is is None (or unset), it *prints* all gettable properties from *obj*. Many properties have aliases for shorter typing, e.g. 'lw' is an alias for 'linewidth'. In the output, aliases and full property names will be listed as:

property or alias = value

e.g.:

linewidth or lw = 2

matplotlib.artist.setp

matplotlib.artist.**setp**(*obj*, **args*, *file=None*, ***kwargs*)

Set a property on an artist object.

matplotlib supports the use of *setp()* ("set property") and *getp()* to set and get object properties, as well as to do introspection on the object. For example, to set the linestyle of a line to be dashed, you can do:

```
>>> line, = plot([1, 2, 3])
>>> setp(line, linestyle='--')
```

If you want to know the valid types of arguments, you can provide the name of the property you want to set without a value:

```
>>> setp(line, 'linestyle')
    linestyle: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
```

If you want to see all the properties that can be set, and their possible values, you can do:

```
>>> setp(line)
    ... long output listing omitted
```

By default *setp* prints to `sys.stdout`, but this can be modified using the *file* keyword-only argument:

```
>>> with fopen('output.log') as f:
>>>     setp(line, file=f)
```

setp() operates on a single instance or a iterable of instances. If you are in query mode introspecting the possible values, only the first instance in the sequence is used. When actually setting values, all the instances will be set. e.g., suppose you have a list of two lines, the following will make both lines thicker and red:

```
>>> x = arange(0, 1, 0.01)
>>> y1 = sin(2*pi*x)
>>> y2 = sin(4*pi*x)
>>> lines = plot(x, y1, x, y2)
>>> setp(lines, linewidth=2, color='r')
```

setp() works with the MATLAB style string/value pairs or with python kwargs. For example, the following are equivalent:

```
>>> setp(lines, 'linewidth', 2, 'color', 'r')  # MATLAB style
>>> setp(lines, linewidth=2, color='r')        # python style
```

matplotlib.artist.kwdoc

matplotlib.artist.**kwdoc**(*artist*)

Inspect an *Artist* class (using *ArtistInspector*) and return information about its settable properties and their current values.

Parameters

artist

[*Artist* or an iterable of *Artist*s]

Returns

str

The settable properties of *artist*, as plain text if `rcParams["docstring.hardcopy"]` (default: `False`) is False and as a rst table (intended for use in Sphinx) if it is True.

matplotlib.artist.ArtistInspector

class matplotlib.artist.**ArtistInspector**(*o*)

A helper class to inspect an *Artist* and return information about its settable properties and their current values.

Initialize the artist inspector with an *Artist* or an iterable of *Artist*s. If an iterable is used, we assume it is a homogeneous sequence (all *Artist*s are of the same type) and it is your responsibility to make sure this is so.

__init__(*self*, *o*)

Initialize the artist inspector with an *Artist* or an iterable of *Artist*s. If an iterable is used, we assume it is a homogeneous sequence (all *Artist*s are of the same type) and it is your responsibility to make sure this is so.

Methods

__init__(self, o)	Initialize the artist inspector with an *Artist* or an iterable of *Artist*s.
aliased_name(self, s)	Return 'PROPNAME or alias' if *s* has an alias, else return 'PROPNAME'.
aliased_name_rest(self, s, target)	Return 'PROPNAME or alias' if *s* has an alias, else return 'PROPNAME', formatted for reST.

continues on next page

Table 55 – continued from previous page

get_aliases(self)	Get a dict mapping property fullnames to sets of aliases for each alias in the *ArtistInspector*.
get_setters(self)	Get the attribute strings with setters for object.
get_valid_values(self, attr)	Get the legal arguments for the setter associated with *attr*.
is_alias(self, o)	Return whether method object *o* is an alias for another method.
pprint_getters(self)	Return the getters and actual values as list of strings.
pprint_setters(self[, prop, leadingspace])	If *prop* is *None*, return a list of strings of all settable properties and their valid values.
pprint_setters_rest(self[, prop, leadingspace])	If *prop* is *None*, return a list of reST-formatted strings of all settable properties and their valid values.
properties(self)	Return a dictionary mapping property name -> value.

aliased_name(*self*, *s*)
Return 'PROPNAME or alias' if *s* has an alias, else return 'PROPNAME'.

e.g., for the line markerfacecolor property, which has an alias, return 'markerfacecolor or mfc' and for the transform property, which does not, return 'transform'.

aliased_name_rest(*self*, *s*, *target*)
Return 'PROPNAME or alias' if *s* has an alias, else return 'PROPNAME', formatted for reST.

e.g., for the line markerfacecolor property, which has an alias, return 'markerfacecolor or mfc' and for the transform property, which does not, return 'transform'.

get_aliases(*self*)
Get a dict mapping property fullnames to sets of aliases for each alias in the *ArtistInspector*.

e.g., for lines:

```
{'markerfacecolor': {'mfc'},
 'linewidth'      : {'lw'},
}
```

get_setters(*self*)
Get the attribute strings with setters for object.

For example, for a line, return `['markerfacecolor', 'linewidth', ....]`.

`get_valid_values`(*self*, *attr*)
: Get the legal arguments for the setter associated with *attr*.

  This is done by querying the docstring of the setter for a line that begins with "ACCEPTS:" or ".. ACCEPTS:", and then by looking for a numpydoc-style documentation for the setter's first argument.

`is_alias`(*self*, *o*)
: Return whether method object *o* is an alias for another method.

`pprint_getters`(*self*)
: Return the getters and actual values as list of strings.

`pprint_setters`(*self*, *prop=None*, *leadingspace=2*)
: If *prop* is *None*, return a list of strings of all settable properties and their valid values.

  If *prop* is not *None*, it is a valid property name and that property will be returned as a string of property : valid values.

`pprint_setters_rest`(*self*, *prop=None*, *leadingspace=4*)
: If *prop* is *None*, return a list of reST-formatted strings of all settable properties and their valid values.

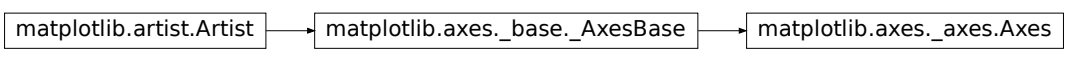
  If *prop* is not *None*, it is a valid property name and that property will be returned as a string of "property : valid" values.

`properties`(*self*)
: Return a dictionary mapping property name -> value.

18.5 `matplotlib.axes`

Table of Contents

18.5.1 Inheritance



18.5.2 The Axes class

class matplotlib.axes.**Axes**(*fig*, *rect*, *facecolor=None*, *frameon=True*, *sharex=None*, *sharey=None*, *label=''*, *xscale=None*, *yscale=None*, *box_aspect=None*, ***kwargs*)

Bases: matplotlib.axes._base._AxesBase

The *Axes* contains most of the figure elements: *Axis*, *Tick*, *Line2D*, *Text*, *Polygon*, etc., and sets the coordinate system.

The *Axes* instance supports callbacks through a callbacks attribute which is a *CallbackRegistry* instance. The events you can connect to are 'xlim_changed' and 'ylim_changed' and the callback will be called with func(*ax*) where *ax* is the *Axes* instance.

Attributes

dataLim

[*Bbox*] The bounding box enclosing all data displayed in the Axes.

viewLim

[*Bbox*] The view limits in data coordinates.

Build an axes in a figure.

Parameters

fig

[*Figure*] The axes is build in the *Figure* *fig*.

rect

[[left, bottom, width, height]] The axes is build in the rectangle *rect*. *rect* is in *Figure* coordinates.

sharex, sharey

[*Axes*, optional] The x or y *axis* is shared with the x or y axis in the input *Axes*.

frameon

[bool, default: True] Whether the axes frame is visible.

box_aspect

[None, or a number, optional] Sets the aspect of the axes box. See *set_box_aspect* for details.

****kwargs**

Other optional keyword arguments:

Property	Description
adjustable	{'box', 'datalim'}
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and r
alpha	float or None
anchor	2-tuple of floats or {'C', 'SW', 'S', 'SE', ...}
animated	bool
aspect	{'auto'} or num
autoscale_on	bool
autoscalex_on	bool
autoscaley_on	bool
axes_locator	Callable[[Axes, Renderer], Bbox]
axisbelow	bool or 'line'
box_aspect	None, or a number
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
contains	unknown
facecolor or fc	color
figure	*Figure*
frame_on	bool
gid	str
in_layout	bool
label	object
navigate	bool
navigate_mode	unknown
path_effects	*AbstractPathEffect*
picker	None or bool or callable
position	[left, bottom, width, height] or *Bbox*
prop_cycle	unknown
rasterization_zorder	float or None
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
title	str
transform	*Transform*
url	str
visible	bool
xbound	unknown
xlabel	str
xlim	(bottom: float, top: float)
xmargin	float greater than -0.5
xscale	{"linear", "log", "symlog", "logit", ...}
xticklabels	unknown
xticks	unknown

Table 56 – continued from previous page

Property	Description
ybound	unknown
ylabel	str
ylim	(bottom: float, top: float)
ymargin	float greater than -0.5
yscale	{"linear", "log", "symlog", "logit", ...}
yticklabels	unknown
yticks	unknown
zorder	float

Returns

Axes

The new *Axes* object.

18.5.3 Subplots

SubplotBase	Base class for subplots, which are *Axes* instances with additional methods to facilitate generating and manipulating a set of *Axes* within a figure.
subplot_class_factory	Make a new class that inherits from *SubplotBase* and the given axes_class (which is assumed to be a subclass of *axes.Axes*).

matplotlib.axes.SubplotBase

class matplotlib.axes.**SubplotBase**(*fig*, **args*, ***kwargs*)

Bases: object

Base class for subplots, which are *Axes* instances with additional methods to facilitate generating and manipulating a set of *Axes* within a figure.

Parameters

fig

[*matplotlib.figure.Figure*]

***args**

[tuple (*nrows*, *ncols*, *index*) or int] The array of subplots in the figure has dimensions `(nrows, ncols)`, and *index* is the index of

the subplot being created. *index* starts at 1 in the upper left corner and increases to the right.

If *nrows*, *ncols*, and *index* are all single digit numbers, then *args* can be passed as a single 3-digit number (e.g. 234 for (2, 3, 4)).

****kwargs**

Keyword arguments are passed to the Axes (sub)class constructor.

`__dict__ = mappingproxy({'__module__': 'matplotlib.axes._subplots', '__doc__': '\n Base c`

`__init__`(*self*, *fig*, **args*, ***kwargs*)

Parameters

fig

[*matplotlib.figure.Figure*]

***args**

[tuple (*nrows*, *ncols*, *index*) or int] The array of subplots in the figure has dimensions `(nrows, ncols)`, and *index* is the index of the subplot being created. *index* starts at 1 in the upper left corner and increases to the right.

If *nrows*, *ncols*, and *index* are all single digit numbers, then *args* can be passed as a single 3-digit number (e.g. 234 for (2, 3, 4)).

****kwargs**

Keyword arguments are passed to the Axes (sub)class constructor.

`__module__ = 'matplotlib.axes._subplots'`

`__reduce__`(*self*)
Helper for pickle.

`__repr__`(*self*)
Return repr(self).

`__weakref__`
list of weak references to the object (if defined)

`change_geometry`(*self*, *numrows*, *numcols*, *num*)
Change subplot geometry, e.g., from (1, 1, 1) to (2, 2, 3).

`property colNum`

`get_geometry`(*self*)
Get the subplot geometry, e.g., (2, 2, 3).

`get_gridspec`(*self*)
Return the *GridSpec* instance associated with the subplot.

`get_subplotspec`(*self*)
Return the *SubplotSpec* instance associated with the subplot.

`is_first_col`(*self*)

`is_first_row`(*self*)

`is_last_col`(*self*)

`is_last_row`(*self*)

`label_outer`(*self*)
Only show "outer" labels and tick labels.

x-labels are only kept for subplots on the last row; y-labels only for subplots on the first column.

`property rowNum`

`set_subplotspec`(*self*, *subplotspec*)
Set the *SubplotSpec*. instance associated with the subplot.

`update_params`(*self*)
Update the subplot position from `self.figure.subplotpars`.

Examples using `matplotlib.axes.SubplotBase`

- sphx_glr_gallery_text_labels_and_annotations_usetex_baseline_test.py
- sphx_glr_gallery_axes_grid1_parasite_simple2.py
- sphx_glr_gallery_axisartist_axis_direction_demo_step01.py
- sphx_glr_gallery_axisartist_axis_direction_demo_step02.py
- sphx_glr_gallery_axisartist_axis_direction_demo_step03.py
- sphx_glr_gallery_axisartist_axis_direction_demo_step04.py
- sphx_glr_gallery_axisartist_demo_axis_direction.py
- sphx_glr_gallery_axisartist_demo_axisline_style.py
- sphx_glr_gallery_axisartist_demo_curvelinear_grid.py
- sphx_glr_gallery_axisartist_demo_curvelinear_grid2.py
- sphx_glr_gallery_axisartist_demo_floating_axes.py
- sphx_glr_gallery_axisartist_demo_floating_axis.py
- sphx_glr_gallery_axisartist_demo_ticklabel_alignment.py
- sphx_glr_gallery_axisartist_demo_ticklabel_direction.py

- sphx_glr_gallery_axisartist_simple_axis_direction01.py
- sphx_glr_gallery_axisartist_simple_axis_direction03.py
- sphx_glr_gallery_axisartist_simple_axis_pad.py
- sphx_glr_gallery_axisartist_simple_axisartist1.py
- sphx_glr_gallery_axisartist_simple_axisline.py
- sphx_glr_gallery_axisartist_simple_axisline2.py
- sphx_glr_gallery_axisartist_simple_axisline3.py

matplotlib.axes.subplot_class_factory

matplotlib.axes.**subplot_class_factory**(*axes_class=None*)

Make a new class that inherits from *SubplotBase* and the given axes_class (which is assumed to be a subclass of *axes.Axes*). This is perhaps a little bit roundabout to make a new class on the fly like this, but it means that a new Subplot class does not have to be created for every type of Axes.

Examples using `matplotlib.axes.subplot_class_factory`

- sphx_glr_gallery_text_labels_and_annotations_usetex_baseline_test.py

18.5.4 Plotting

Basic

Axes.plot	Plot y versus x as lines and/or markers.
Axes.errorbar	Plot y versus x as lines and/or markers with attached errorbars.
Axes.scatter	A scatter plot of *y* vs.
Axes.plot_date	Plot data that contains dates.
Axes.step	Make a step plot.
Axes.loglog	Make a plot with log scaling on both the x and y axis.
Axes.semilogx	Make a plot with log scaling on the x axis.
Axes.semilogy	Make a plot with log scaling on the y axis.
Axes.fill_between	Fill the area between two horizontal curves.

continues on next page

Table 58 – continued from previous page

Axes.fill_betweenx	Fill the area between two vertical curves.
Axes.bar	Make a bar plot.
Axes.barh	Make a horizontal bar plot.
Axes.stem	Create a stem plot.
Axes.eventplot	Plot identical parallel lines at the given positions.
Axes.pie	Plot a pie chart.
Axes.stackplot	Draw a stacked area plot.
Axes.broken_barh	Plot a horizontal sequence of rectangles.
Axes.vlines	Plot vertical lines.
Axes.hlines	Plot horizontal lines at each *y* from *xmin* to *xmax*.
Axes.fill	Plot filled polygons.

matplotlib.axes.Axes.plot

Axes.**plot**(*self*, **args*, *scalex=True*, *scaley=True*, *data=None*, ***kwargs*)

Plot y versus x as lines and/or markers.

Call signatures:

```
plot([x], y, [fmt], *, data=None, **kwargs)
plot([x], y, [fmt], [x2], y2, [fmt2], ..., **kwargs)
```

The coordinates of the points or line nodes are given by *x*, *y*.

The optional parameter *fmt* is a convenient way for defining basic formatting like color, marker and linestyle. It's a shortcut string notation described in the *Notes* section below.

```
>>> plot(x, y)        # plot x and y using default line style and color
>>> plot(x, y, 'bo')  # plot x and y using blue circle markers
>>> plot(y)           # plot y using x as index array 0..N-1
>>> plot(y, 'r+')     # ditto, but with red plusses
```

You can use *Line2D* properties as keyword arguments for more control on the appearance. Line properties and *fmt* can be mixed. The following two calls yield identical results:

```
>>> plot(x, y, 'go--', linewidth=2, markersize=12)
>>> plot(x, y, color='green', marker='o', linestyle='dashed',
...      linewidth=2, markersize=12)
```

When conflicting with *fmt*, keyword arguments take precedence.

Plotting labelled data

There's a convenient way for plotting objects with labelled data (i.e. data that can be accessed by index `obj['y']`). Instead of giving the data in *x* and *y*, you can provide the object in the *data* parameter and just give the labels for *x* and *y*:

```
>>> plot('xlabel', 'ylabel', data=obj)
```

All indexable objects are supported. This could e.g. be a `dict`, a `pandas.DataFrame` or a structured numpy array.

Plotting multiple sets of data

There are various ways to plot multiple sets of data.

- The most straight forward way is just to call *plot* multiple times. Example:

  ```
  >>> plot(x1, y1, 'bo')
  >>> plot(x2, y2, 'go')
  ```

- Alternatively, if your data is already a 2d array, you can pass it directly to *x*, *y*. A separate data set will be drawn for every column.

  Example: an array `a` where the first column represents the *x* values and the other columns are the *y* columns:

  ```
  >>> plot(a[0], a[1:])
  ```

- The third way is to specify multiple sets of *[x], y, [fmt]* groups:

  ```
  >>> plot(x1, y1, 'g^', x2, y2, 'g-')
  ```

  In this case, any additional keyword argument applies to all datasets. Also this syntax cannot be combined with the *data* parameter.

By default, each line is assigned a different style specified by a 'style cycle'. The *fmt* and line property parameters are only necessary if you want explicit deviations from these defaults. Alternatively, you can also change the style cycle using `rcParams["axes.prop_cycle"]` (default: `cycler('color', ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728', '#9467bd', '#8c564b', '#e377c2', '#7f7f7f', '#bcbd22', '#17becf'])`).

Parameters

x, y

[array-like or scalar] The horizontal / vertical coordinates of the data points. *x* values are optional and default to `range(len(y))`.

Commonly, these parameters are 1D arrays.

They can also be scalars, or two-dimensional (in that case, the columns represent separate data sets).

These arguments cannot be passed as keywords.

fmt

[str, optional] A format string, e.g. 'ro' for red circles. See the *Notes* section for a full description of the format strings.

Format strings are just an abbreviation for quickly setting basic line properties. All of these and more can also be controlled by keyword arguments.

This argument cannot be passed as keyword.

data

[indexable object, optional] An object with labelled data. If given, provide the label names to plot in *x* and *y*.

Note: Technically there's a slight ambiguity in calls where the second label is a valid *fmt*. `plot('n', 'o', data=obj)` could be `plt(x, y)` or `plt(y, fmt)`. In such cases, the former interpretation is chosen, but a warning is issued. You may suppress the warning by adding an empty format string `plot('n', 'o', '', data=obj)`.

Returns

list of *Line2D*

A list of lines representing the plotted data.

Other Parameters

scalex, scaley

[bool, default: True] These parameters determine if the view limits are adapted to the data limits. The values are passed on to *autoscale_view*.

****kwargs**

[*Line2D* properties, optional] *kwargs* are used to specify properties like a line label (for auto legends), linewidth, antialiasing, marker face color. Example:

```
>>> plot([1, 2, 3], [1, 2, 3], 'go-', label='line 1', linewidth=2)
>>> plot([1, 2, 3], [1, 4, 9], 'rs', label='line 2')
```

If you make multiple lines with one plot call, the kwargs apply to all those lines.

Here is a list of available *Line2D* properties:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value
alpha	float or None
animated	bool
antialiased or aa	bool
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color or c	color
contains	unknown
dash_capstyle	{'butt', 'round', 'projecting'}
dash_joinstyle	{'miter', 'round', 'bevel'}
dashes	sequence of floats (on/off ink in points) or (None, None)
data	(2, N) array or two 1D arrays
drawstyle or ds	{'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'd
figure	*Figure*
fillstyle	{'full', 'left', 'right', 'bottom', 'top', 'none'}
gid	str
in_layout	bool
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float
marker	marker style string, *Path* or *MarkerStyle*
markeredgecolor or mec	color
markeredgewidth or mew	float
markerfacecolor or mfc	color
markerfacecoloralt or mfcalt	color
markersize or ms	float
markevery	None or int or (int, int) or slice or List[int] or float or (float, float)
path_effects	*AbstractPathEffect*
picker	unknown
pickradius	float
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
solid_capstyle	{'butt', 'round', 'projecting'}
solid_joinstyle	{'miter', 'round', 'bevel'}
transform	*matplotlib.transforms.Transform*
url	str
visible	bool
xdata	1D array
ydata	1D array
zorder	float

See also:

scatter

XY scatter plot with markers of varying size and/or color (sometimes also called bubble chart).

Notes

Format Strings

A format string consists of a part for color, marker and line:

```
fmt = '[marker][line][color]'
```

Each of them is optional. If not provided, the value from the style cycle is used. Exception: If `line` is given, but no `marker`, the data will be a line without markers.

Other combinations such as `[color][marker][line]` are also supported, but note that their parsing may be ambiguous.

Markers

character	description
`'.'`	point marker
`','`	pixel marker
`'o'`	circle marker
`'v'`	triangle_down marker
`'^'`	triangle_up marker
`'<'`	triangle_left marker
`'>'`	triangle_right marker
`'1'`	tri_down marker
`'2'`	tri_up marker
`'3'`	tri_left marker
`'4'`	tri_right marker
`'s'`	square marker
`'p'`	pentagon marker
`'*'`	star marker
`'h'`	hexagon1 marker
`'H'`	hexagon2 marker
`'+'`	plus marker
`'x'`	x marker
`'D'`	diamond marker
`'d'`	thin_diamond marker
`'|'`	vline marker
`'_'`	hline marker

Line Styles

character	description
'-'	solid line style
'--'	dashed line style
'-.'	dash-dot line style
':'	dotted line style

Example format strings:

```
'b'    # blue markers with default shape
'or'   # red circles
'-g'   # green solid line
'--'   # dashed line with default color
'^k:'  # black triangle_up markers connected by a dotted line
```

Colors

The supported color abbreviations are the single letter codes

character	color
'b'	blue
'g'	green
'r'	red
'c'	cyan
'm'	magenta
'y'	yellow
'k'	black
'w'	white

and the 'CN' colors that index into the default property cycle.

If the color is the only part of the format string, you can additionally use any *matplotlib.colors* spec, e.g. full names ('green') or hex strings ('#008000').

Examples using `matplotlib.axes.Axes.plot`

- sphx_glr_gallery_lines_bars_and_markers_categorical_variables.py
- sphx_glr_gallery_lines_bars_and_markers_csd_demo.py
- sphx_glr_gallery_lines_bars_and_markers_curve_error_band.py
- sphx_glr_gallery_lines_bars_and_markers_eventcollection_demo.py
- sphx_glr_gallery_lines_bars_and_markers_fill_between_demo.py
- sphx_glr_gallery_lines_bars_and_markers_fill_betweenx_demo.py
- sphx_glr_gallery_lines_bars_and_markers_joinstyle.py
- sphx_glr_gallery_lines_bars_and_markers_line_demo_dash_control.py

- sphx_glr_gallery_lines_bars_and_markers_marker_reference.py
- sphx_glr_gallery_lines_bars_and_markers_markevery_demo.py
- sphx_glr_gallery_lines_bars_and_markers_markevery_prop_cycle.py
- sphx_glr_gallery_lines_bars_and_markers_psd_demo.py
- sphx_glr_gallery_lines_bars_and_markers_simple_plot.py
- sphx_glr_gallery_lines_bars_and_markers_span_regions.py
- sphx_glr_gallery_lines_bars_and_markers_step_demo.py
- sphx_glr_gallery_lines_bars_and_markers_timeline.py
- sphx_glr_gallery_lines_bars_and_markers_vline_hline_demo.py
- sphx_glr_gallery_images_contours_and_fields_contour_corner_mask.py
- sphx_glr_gallery_images_contours_and_fields_image_demo.py
- sphx_glr_gallery_images_contours_and_fields_irregulardatagrid.py
- sphx_glr_gallery_images_contours_and_fields_pcolormesh_grids.py
- sphx_glr_gallery_images_contours_and_fields_plot_streamplot.py
- sphx_glr_gallery_images_contours_and_fields_specgram_demo.py
- sphx_glr_gallery_images_contours_and_fields_triinterp_demo.py
- sphx_glr_gallery_images_contours_and_fields_watermark_image.py
- sphx_glr_gallery_subplots_axes_and_figures_align_labels_demo.py
- sphx_glr_gallery_subplots_axes_and_figures_axes_box_aspect.py
- sphx_glr_gallery_subplots_axes_and_figures_axes_demo.py
- sphx_glr_gallery_subplots_axes_and_figures_axes_margins.py
- sphx_glr_gallery_subplots_axes_and_figures_axes_props.py
- sphx_glr_gallery_subplots_axes_and_figures_axhspan_demo.py
- sphx_glr_gallery_subplots_axes_and_figures_broken_axis.py
- sphx_glr_gallery_subplots_axes_and_figures_demo_constrained_layout.py
- sphx_glr_gallery_subplots_axes_and_figures_demo_tight_layout.py
- sphx_glr_gallery_subplots_axes_and_figures_invert_axes.py
- sphx_glr_gallery_subplots_axes_and_figures_secondary_axis.py
- sphx_glr_gallery_subplots_axes_and_figures_subplot.py
- sphx_glr_gallery_subplots_axes_and_figures_subplot_demo.py
- sphx_glr_gallery_subplots_axes_and_figures_subplots_demo.py
- sphx_glr_gallery_subplots_axes_and_figures_two_scales.py

- sphx_glr_gallery_statistics_boxplot_demo.py
- sphx_glr_gallery_statistics_histogram_cumulative.py
- sphx_glr_gallery_statistics_histogram_features.py
- sphx_glr_gallery_pie_and_polar_charts_polar_demo.py
- sphx_glr_gallery_pie_and_polar_charts_polar_legend.py
- sphx_glr_gallery_text_labels_and_annotations_accented_text.py
- sphx_glr_gallery_text_labels_and_annotations_annotation_demo.py
- sphx_glr_gallery_text_labels_and_annotations_custom_legends.py
- sphx_glr_gallery_text_labels_and_annotations_date.py
- sphx_glr_gallery_text_labels_and_annotations_date_index_formatter.py
- sphx_glr_gallery_text_labels_and_annotations_demo_annotation_box.py
- sphx_glr_gallery_text_labels_and_annotations_engineering_formatter.py
- sphx_glr_gallery_text_labels_and_annotations_legend.py
- sphx_glr_gallery_text_labels_and_annotations_legend_demo.py
- sphx_glr_gallery_text_labels_and_annotations_mathtext_demo.py
- sphx_glr_gallery_text_labels_and_annotations_tex_demo.py
- sphx_glr_gallery_text_labels_and_annotations_text_rotation_relative_to_line.py
- sphx_glr_gallery_text_labels_and_annotations_titles_demo.py
- sphx_glr_gallery_text_labels_and_annotations_watermark_text.py
- sphx_glr_gallery_pyplots_align_ylabels.py
- sphx_glr_gallery_pyplots_annotate_transform.py
- sphx_glr_gallery_pyplots_annotation_basic.py
- sphx_glr_gallery_pyplots_annotation_polar.py
- sphx_glr_gallery_pyplots_auto_subplots_adjust.py
- sphx_glr_gallery_pyplots_dollar_ticks.py
- sphx_glr_gallery_pyplots_fig_axes_labels_simple.py
- sphx_glr_gallery_pyplots_pyplot_formatstr.py
- sphx_glr_gallery_pyplots_pyplot_three.py
- sphx_glr_gallery_pyplots_text_commands.py
- sphx_glr_gallery_pyplots_whats_new_98_4_fill_between.py
- sphx_glr_gallery_pyplots_whats_new_99_spines.py
- sphx_glr_gallery_color_color_demo.py

- sphx_glr_gallery_color_color_by_yvalue.py
- sphx_glr_gallery_shapes_and_collections_marker_path.py
- sphx_glr_gallery_shapes_and_collections_path_patch.py
- sphx_glr_gallery_shapes_and_collections_quad_bezier.py
- sphx_glr_gallery_style_sheets_dark_background.py
- sphx_glr_gallery_style_sheets_fivethirtyeight.py
- sphx_glr_gallery_style_sheets_ggplot.py
- sphx_glr_gallery_axes_grid1_demo_fixed_size_axes.py
- sphx_glr_gallery_axes_grid1_parasite_simple.py
- sphx_glr_gallery_axes_grid1_simple_axisline4.py
- sphx_glr_gallery_axisartist_demo_axisline_style.py
- sphx_glr_gallery_axisartist_demo_parasite_axes.py
- sphx_glr_gallery_axisartist_demo_parasite_axes2.py
- sphx_glr_gallery_axisartist_simple_axisline.py
- sphx_glr_gallery_axisartist_simple_axisline2.py
- sphx_glr_gallery_showcase_anatomy.py
- sphx_glr_gallery_showcase_bachelors_degrees_by_gender.py
- sphx_glr_gallery_showcase_integral.py
- sphx_glr_gallery_showcase_xkcd.py
- *Decay*
- *The Bayes update*
- *The double pendulum problem*
- *Animated 3D random walk*
- *Animated line plot*
- *MATPLOTLIB UNCHAINED*
- sphx_glr_gallery_event_handling_coords_demo.py
- sphx_glr_gallery_event_handling_data_browser.py
- sphx_glr_gallery_event_handling_keypress_demo.py
- sphx_glr_gallery_event_handling_legend_picking.py
- sphx_glr_gallery_event_handling_looking_glass.py
- sphx_glr_gallery_event_handling_path_editor.py
- sphx_glr_gallery_event_handling_pick_event_demo2.py

- sphx_glr_gallery_event_handling_resample.py
- sphx_glr_gallery_event_handling_timers.py
- sphx_glr_gallery_frontpage_histogram.py
- sphx_glr_gallery_frontpage_membrane.py
- sphx_glr_gallery_misc_agg_buffer_to_array.py
- sphx_glr_gallery_misc_bbox_intersect.py
- sphx_glr_gallery_misc_cursor_demo.py
- sphx_glr_gallery_misc_load_converter.py
- sphx_glr_gallery_misc_patheffect_demo.py
- sphx_glr_gallery_misc_pythonic_matplotlib.py
- sphx_glr_gallery_misc_svg_filter_line.py
- sphx_glr_gallery_misc_zorder_demo.py
- sphx_glr_gallery_mplot3d_2dcollections3d.py
- sphx_glr_gallery_mplot3d_lines3d.py
- sphx_glr_gallery_mplot3d_lorenz_attractor.py
- sphx_glr_gallery_mplot3d_mixed_subplots.py
- sphx_glr_gallery_recipes_centered_spines_with_arrows.py
- sphx_glr_gallery_recipes_common_date_problems.py
- sphx_glr_gallery_recipes_create_subplots.py
- sphx_glr_gallery_recipes_fill_between_alpha.py
- sphx_glr_gallery_recipes_share_axis_lims_views.py
- sphx_glr_gallery_scales_aspect_loglog.py
- sphx_glr_gallery_scales_scales.py
- sphx_glr_gallery_specialty_plots_anscombe.py
- sphx_glr_gallery_specialty_plots_radar_chart.py
- sphx_glr_gallery_ticks_and_spines_centered_ticklabels.py
- sphx_glr_gallery_ticks_and_spines_date_concise_formatter.py
- sphx_glr_gallery_ticks_and_spines_date_index_formatter2.py
- sphx_glr_gallery_ticks_and_spines_date_precision_and_epochs.py
- sphx_glr_gallery_ticks_and_spines_major_minor_demo.py
- sphx_glr_gallery_ticks_and_spines_multiple_yaxis_with_spines.py
- sphx_glr_gallery_ticks_and_spines_scalarformatter.py

- sphx_glr_gallery_ticks_and_spines_spine_placement_demo.py
- sphx_glr_gallery_ticks_and_spines_spines.py
- sphx_glr_gallery_ticks_and_spines_spines_bounds.py
- sphx_glr_gallery_ticks_and_spines_tick_label_right.py
- sphx_glr_gallery_ticks_and_spines_tick_labels_from_values.py
- sphx_glr_gallery_ticks_and_spines_tick_xlabel_top.py
- sphx_glr_gallery_units_evans_test.py
- sphx_glr_gallery_user_interfaces_canvasagg.py
- sphx_glr_gallery_userdemo_annotate_explain.py
- sphx_glr_gallery_userdemo_connect_simple01.py
- sphx_glr_gallery_userdemo_connectionstyle_demo.py
- sphx_glr_gallery_userdemo_demo_gridspec06.py
- sphx_glr_gallery_userdemo_simple_annotate01.py
- sphx_glr_gallery_userdemo_simple_legend02.py
- sphx_glr_gallery_widgets_check_buttons.py
- sphx_glr_gallery_widgets_cursor.py
- sphx_glr_gallery_widgets_multicursor.py
- sphx_glr_gallery_widgets_radio_buttons.py
- sphx_glr_gallery_widgets_rectangle_selector.py
- sphx_glr_gallery_widgets_span_selector.py
- sphx_glr_gallery_widgets_textbox.py
- *Usage Guide*
- *Artist tutorial*
- *Styling with cycler*
- *Customizing Figure Layouts Using GridSpec and Other Functions*
- *Constrained Layout Guide*
- *Tight Layout guide*
- *Blitting tutorial*
- *Path Tutorial*
- *Transformations Tutorial*
- *Specifying Colors*
- *Text in Matplotlib Plots*

matplotlib.axes.Axes.errorbar

Axes.errorbar(*self*, *x*, *y*, *yerr=None*, *xerr=None*, *fmt=''*, *ecolor=None*, *elinewidth=None*, *capsize=None*, *barsabove=False*, *lolims=False*, *uplims=False*, *xlolims=False*, *xuplims=False*, *errorevery=1*, *capthick=None*, ***, *data=None*, ***kwargs*)

Plot y versus x as lines and/or markers with attached errorbars.

x, *y* define the data locations, *xerr*, *yerr* define the errorbar sizes. By default, this draws the data markers/lines as well the errorbars. Use fmt='none' to draw errorbars without any data markers.

Parameters

x, y

[float or array-like] The data positions.

xerr, yerr

[float or array-like, shape(N,) or shape(2, N), optional] The errorbar sizes:

- scalar: Symmetric +/- values for all data points.
- shape(N,): Symmetric +/-values for each data point.
- shape(2, N): Separate - and + values for each bar. First row contains the lower errors, the second row contains the upper errors.
- *None*: No errorbar.

Note that all error arrays should have *positive* values.

See /gallery/statistics/errorbar_features for an example on the usage of `xerr` and `yerr`.

fmt

[str, default: ''] The format for the data points / data lines. See *plot* for details.

Use 'none' (case insensitive) to plot errorbars without any data markers.

ecolor

[color, default: None] The color of the errorbar lines. If None, use the color of the line connecting the markers.

elinewidth

[float, default: None] The linewidth of the errorbar lines. If None, the linewidth of the current style is used.

capsize

[float, default: `rcParams["errorbar.capsize"]` (default: `0.0`)] The length of the error bar caps in points.

capthick

[float, default: None] An alias to the keyword argument *markeredgewidth* (a.k.a. *mew*). This setting is a more sensible name for the property that controls the thickness of the error bar cap in points. For backwards compatibility, if *mew* or *markeredgewidth* are given, then they will over-ride *capthick*. This may change in future releases.

barsabove

[bool, default: False] If True, will plot the errorbars above the plot symbols. Default is below.

lolims, uplims, xlolims, xuplims

[bool, default: False] These arguments can be used to indicate that a value gives only upper/lower limits. In that case a caret symbol is used to indicate this. *lims*-arguments may be scalars, or array-likes of the same length as *xerr* and *yerr*. To use limits with inverted axes, *set_xlim* or *set_ylim* must be called before *errorbar()*. Note the tricky parameter names: setting e.g. *lolims* to True means that the y-value is a *lower* limit of the True value, so, only an *upward*-pointing arrow will be drawn!

errorevery

[int or (int, int), default: 1] draws error bars on a subset of the data. *errorevery* =N draws error bars on the points (x[::N], y[::N]). *errorevery* =(start, N) draws error bars on the points (x[start::N], y[start::N]). e.g. errorevery=(6, 3) adds error bars to the data at (x[6], x[9], x[12], x[15], ...). Used to avoid overlapping error bars when two series share x-axis values.

Returns

ErrorbarContainer

The container contains:

- plotline: *Line2D* instance of x, y plot markers and/or line.
- caplines: A tuple of *Line2D* instances of the error bar caps.
- barlinecols: A tuple of *LineCollection* with the horizontal and vertical error ranges.

Other Parameters

****kwargs**

All other keyword arguments are passed on to the *plot* call drawing the markers. For example, this code makes big red squares with thick green edges:

```
x, y, yerr = rand(3, 10)
errorbar(x, y, yerr, marker='s', mfc='red',
         mec='green', ms=20, mew=4)
```

where *mfc*, *mec*, *ms* and *mew* are aliases for the longer property names, *markerfacecolor*, *markeredgecolor*, *markersize* and *markeredgewidth*.

Valid kwargs for the marker properties are *Line2D* properties:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value
alpha	float or None
animated	bool
antialiased or aa	bool
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color or c	color
contains	unknown
dash_capstyle	{'butt', 'round', 'projecting'}
dash_joinstyle	{'miter', 'round', 'bevel'}
dashes	sequence of floats (on/off ink in points) or (None, None)
data	(2, N) array or two 1D arrays
drawstyle or ds	{'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'd
figure	*Figure*
fillstyle	{'full', 'left', 'right', 'bottom', 'top', 'none'}
gid	str
in_layout	bool
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float
marker	marker style string, *Path* or *MarkerStyle*
markeredgecolor or mec	color
markeredgewidth or mew	float
markerfacecolor or mfc	color
markerfacecoloralt or mfcalt	color
markersize or ms	float
markevery	None or int or (int, int) or slice or List[int] or float or (float, float)
path_effects	*AbstractPathEffect*
picker	unknown

Table 60 – continued from previous page

Property	Description
pickradius	float
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
solid_capstyle	{'butt', 'round', 'projecting'}
solid_joinstyle	{'miter', 'round', 'bevel'}
transform	*matplotlib.transforms.Transform*
url	str
visible	bool
xdata	1D array
ydata	1D array
zorder	float

Notes

Note: In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, the following arguments can also be string `s`, which is interpreted as `data[s]` (unless this raises an exception): *x*, *y*, *xerr*, *yerr*.

Objects passed as **data** must support item access (`data[s]`) and membership test (`s in data`).

Examples using `matplotlib.axes.Axes.errorbar`

- sphx_glr_gallery_lines_bars_and_markers_errorbar_limits_simple.py
- sphx_glr_gallery_lines_bars_and_markers_errorbar_subsample.py
- sphx_glr_gallery_statistics_errorbar.py
- sphx_glr_gallery_statistics_errorbar_features.py
- sphx_glr_gallery_statistics_errorbar_limits.py
- sphx_glr_gallery_statistics_errorbars_and_boxes.py
- sphx_glr_gallery_text_labels_and_annotations_legend_demo.py
- sphx_glr_gallery_axes_grid1_parasite_simple2.py
- sphx_glr_gallery_scales_log_demo.py

matplotlib.axes.Axes.scatter

Axes.scatter(*self*, *x*, *y*, *s=None*, *c=None*, *marker=None*, *cmap=None*, *norm=None*, *vmin=None*, *vmax=None*, *alpha=None*, *linewidths=None*, *verts=<deprecated parameter>*, *edgecolors=None*, ***, *plotnonfinite=False*, *data=None*, ***kwargs*)

A scatter plot of *y* vs. *x* with varying marker size and/or color.

Parameters

x, y

[float or array-like, shape (n,)] The data positions.

s

[float or array-like, shape (n,), optional] The marker size in points**2. Default is `rcParams['lines.markersize'] ** 2`.

c

[array-like or list of colors or color, optional] The marker colors. Possible values:

- A scalar or sequence of n numbers to be mapped to colors using *cmap* and *norm*.
- A 2-D array in which the rows are RGB or RGBA.
- A sequence of colors of length n.
- A single color format string.

Note that *c* should not be a single numeric RGB or RGBA sequence because that is indistinguishable from an array of values to be colormapped. If you want to specify the same RGB or RGBA value for all points, use a 2-D array with a single row. Otherwise, value- matching will have precedence in case of a size matching with *x* and *y*.

If you wish to specify a single color for all points prefer the *color* keyword argument.

Defaults to `None`. In that case the marker color is determined by the value of *color*, *facecolor* or *facecolors*. In case those are not specified or `None`, the marker color is determined by the next color of the `Axes`' current "shape and fill" color cycle. This cycle defaults to `rcParams["axes.prop_cycle"]` (default: `cycler('color', ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728', '#9467bd', '#8c564b', '#e377c2', '#7f7f7f', '#bcbd22', '#17becf'])`).

marker

[*MarkerStyle*, default: `rcParams["scatter.marker"]` (default: `'o'`)] The marker style. *marker* can be either an instance of the class

or the text shorthand for a particular marker. See *matplotlib.markers* for more information about marker styles.

cmap

[str or *Colormap*, default: `rcParams["image.cmap"]` (default: `'viridis'`)] A *Colormap* instance or registered colormap name. *cmap* is only used if *c* is an array of floats.

norm

[*Normalize*, default: None] If *c* is an array of floats, *norm* is used to scale the color data, *c*, in the range 0 to 1, in order to map into the colormap *cmap*. If *None*, use the default *colors.Normalize*.

vmin, vmax

[float, default: None] *vmin* and *vmax* are used in conjunction with the default norm to map the color array *c* to the colormap *cmap*. If None, the respective min and max of the color array is used. It is deprecated to use *vmin*/*vmax* when *norm* is given.

alpha

[float, default: None] The alpha blending value, between 0 (transparent) and 1 (opaque).

linewidths

[float or array-like, default: `rcParams["lines.linewidth"]` (default: `1.5`)] The linewidth of the marker edges. Note: The default *edgecolors* is 'face'. You may want to change this as well.

edgecolors

[{'face', 'none', *None*} or color or sequence of color, default: `rcParams["scatter.edgecolors"]` (default: `'face'`)] The edge color of the marker. Possible values:

- 'face': The edge color will always be the same as the face color.
- 'none': No patch boundary will be drawn.
- A color or sequence of colors.

For non-filled markers, the *edgecolors* kwarg is ignored and forced to 'face' internally.

plotnonfinite

[bool, default: False] Set to plot points with nonfinite *c*, in conjunction with *set_bad*.

Returns

PathCollection

Other Parameters

****kwargs**

[*Collection* properties]

See also:

plot

To plot scatter plots when markers are identical in size and color.

Notes

- The *plot* function will be faster for scatterplots where markers don't vary in size or color.
- Any or all of *x*, *y*, *s*, and *c* may be masked arrays, in which case all masks will be combined and only unmasked points will be plotted.
- Fundamentally, scatter works with 1-D arrays; *x*, *y*, *s*, and *c* may be input as N-D arrays, but within scatter they will be flattened. The exception is *c*, which will be flattened only if its size matches the size of *x* and *y*.

Note: In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, the following arguments can also be string `s`, which is interpreted as `data[s]` (unless this raises an exception): *x*, *y*, *s*, *linewidths*, *edgecolors*, *c*, *facecolor*, *facecolors*, *color*.

Objects passed as **data** must support item access (`data[s]`) and membership test (`s in data`).

Examples using `matplotlib.axes.Axes.scatter`

- sphx_glr_gallery_lines_bars_and_markers_scatter_custom_symbol.py
- sphx_glr_gallery_lines_bars_and_markers_scatter_demo2.py
- sphx_glr_gallery_lines_bars_and_markers_scatter_hist.py
- sphx_glr_gallery_lines_bars_and_markers_scatter_piecharts.py
- sphx_glr_gallery_lines_bars_and_markers_scatter_with_legend.py
- sphx_glr_gallery_images_contours_and_fields_quiver_demo.py
- sphx_glr_gallery_subplots_axes_and_figures_axes_box_aspect.py
- sphx_glr_gallery_subplots_axes_and_figures_axis_labels_demo.py

- sphx_glr_gallery_statistics_confidence_ellipse.py
- sphx_glr_gallery_statistics_customized_violin.py
- sphx_glr_gallery_pie_and_polar_charts_polar_scatter.py
- sphx_glr_gallery_text_labels_and_annotations_legend_demo.py
- sphx_glr_gallery_shapes_and_collections_scatter.py
- sphx_glr_gallery_axes_grid1_scatter_hist_locatable_axes.py
- sphx_glr_gallery_axisartist_demo_floating_axes.py
- *Rain simulation*
- sphx_glr_gallery_event_handling_zoom_window.py
- sphx_glr_gallery_misc_keyword_plotting.py
- sphx_glr_gallery_misc_zorder_demo.py
- sphx_glr_gallery_mplot3d_2dcollections3d.py
- sphx_glr_gallery_mplot3d_scatter3d.py
- sphx_glr_gallery_ticks_and_spines_auto_ticks.py
- sphx_glr_gallery_units_units_scatter.py
- sphx_glr_gallery_userdemo_annotate_text_arrow.py
- sphx_glr_gallery_widgets_polygon_selector_demo.py
- *Choosing Colormaps in Matplotlib*

matplotlib.axes.Axes.plot_date

`Axes.plot_date`(*self*, *x*, *y*, *fmt='o'*, *tz=None*, *xdate=True*, *ydate=False*, ***, *data=None*, ***kwargs*)

Plot data that contains dates.

Similar to *plot*, this plots *y* vs. *x* as lines or markers. However, the axis labels are formatted as dates depending on *xdate* and *ydate*.

Parameters

x, y

[array-like] The coordinates of the data points. If *xdate* or *ydate* is *True*, the respective values *x* or *y* are interpreted as *Matplotlib dates*.

fmt

[str, optional] The plot format string. For details, see the corresponding parameter in *plot*.

tz

[timezone string or `datetime.tzinfo`, default: `rcParams["timezone"]` (default: `'UTC'`)] The time zone to use in labeling dates.

xdate

[bool, default: True] If *True*, the *x*-axis will be interpreted as Matplotlib dates.

ydate

[bool, default: False] If *True*, the *y*-axis will be interpreted as Matplotlib dates.

Returns

lines

A list of *Line2D* objects representing the plotted data.

Other Parameters

****kwargs**

Keyword arguments control the *Line2D* properties:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value
alpha	float or None
animated	bool
antialiased or aa	bool
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color or c	color
contains	unknown
dash_capstyle	{'butt', 'round', 'projecting'}
dash_joinstyle	{'miter', 'round', 'bevel'}
dashes	sequence of floats (on/off ink in points) or (None, None)
data	(2, N) array or two 1D arrays
drawstyle or ds	{'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'd
figure	*Figure*
fillstyle	{'full', 'left', 'right', 'bottom', 'top', 'none'}
gid	str
in_layout	bool
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}

Table 61 – continued from previous page

Property	Description
linewidth or lw	float
marker	marker style string, *Path* or *MarkerStyle*
markeredgecolor or mec	color
markeredgewidth or mew	float
markerfacecolor or mfc	color
markerfacecoloralt or mfcalt	color
markersize or ms	float
markevery	None or int or (int, int) or slice or List[int] or float or (float, float)
path_effects	*AbstractPathEffect*
picker	unknown
pickradius	float
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
solid_capstyle	{'butt', 'round', 'projecting'}
solid_joinstyle	{'miter', 'round', 'bevel'}
transform	*matplotlib.transforms.Transform*
url	str
visible	bool
xdata	1D array
ydata	1D array
zorder	float

See also:

matplotlib.dates

Helper functions on dates.

matplotlib.dates.date2num

Convert dates to num.

matplotlib.dates.num2date

Convert num to dates.

matplotlib.dates.drange

Create an equally spaced sequence of dates.

Notes

If you are using custom date tickers and formatters, it may be necessary to set the formatters/locators after the call to *plot_date*. *plot_date* will set the default tick locator to *AutoDateLocator* (if the tick locator is not already set to a *DateLocator* instance) and the default tick formatter to *AutoDateFormatter* (if the tick formatter is not already set to a *DateFormatter* instance).

Note: In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, the following arguments can also be string `s`, which is interpreted as `data[s]` (unless this raises an exception): *x*, *y*.

Objects passed as **data** must support item access (`data[s]`) and membership test (`s in data`).

Examples using `matplotlib.axes.Axes.plot_date`

- sphx_glr_gallery_ticks_and_spines_date_demo_convert.py

matplotlib.axes.Axes.step

`Axes.step`(*self*, *x*, *y*, **args*, *where='pre'*, *data=None*, ***kwargs*)

Make a step plot.

Call signatures:

```
step(x, y, [fmt], *, data=None, where='pre', **kwargs)
step(x, y, [fmt], x2, y2, [fmt2], ..., *, where='pre', **kwargs)
```

This is just a thin wrapper around *plot* which changes some formatting options. Most of the concepts and parameters of plot can be used here as well.

Parameters

x

[array-like] 1-D sequence of x positions. It is assumed, but not checked, that it is uniformly increasing.

y

[array-like] 1-D sequence of y levels.

fmt

[str, optional] A format string, e.g. 'g' for a green line. See *plot* for a more detailed description.

Note: While full format strings are accepted, it is recommended to only specify the color. Line styles are currently ignored (use the keyword argument *linestyle* instead). Markers are accepted and plotted on the given positions, however, this is a rarely needed feature for step plots.

data

[indexable object, optional] An object with labelled data. If given, provide the label names to plot in *x* and *y*.

where

[{'pre', 'post', 'mid'}, default: 'pre'] Define where the steps should be placed:

- 'pre': The y value is continued constantly to the left from every *x* position, i.e. the interval `(x[i-1], x[i]]` has the value `y[i]`.
- 'post': The y value is continued constantly to the right from every *x* position, i.e. the interval `[x[i], x[i+1])` has the value `y[i]`.
- 'mid': Steps occur half-way between the *x* positions.

Returns

lines

A list of *Line2D* objects representing the plotted data.

Other Parameters

****kwargs**

Additional parameters are the same as those for *plot*.

Notes

Examples using `matplotlib.axes.Axes.step`

- sphx_glr_gallery_lines_bars_and_markers_step_demo.py

matplotlib.axes.Axes.loglog

Axes.**loglog**(*self*, **args*, ***kwargs*)

Make a plot with log scaling on both the x and y axis.

Call signatures:

```
loglog([x], y, [fmt], data=None, **kwargs)
loglog([x], y, [fmt], [x2], y2, [fmt2], ..., **kwargs)
```

This is just a thin wrapper around *plot* which additionally changes both the x-axis and the y-axis to log scaling. All of the concepts and parameters of plot can be used here as well.

The additional parameters *base*, *subs* and *nonpositive* control the x/y-axis properties. They are just forwarded to *Axes.set_xscale* and *Axes.set_yscale*. To use different properties on the x-axis and the y-axis, use e.g. `ax.set_xscale("log", base=10); ax.set_yscale("log", base=2)`.

Parameters

base

[float, default: 10] Base of the logarithm.

subs

[sequence, optional] The location of the minor ticks. If *None*, reasonable locations are automatically chosen depending on the number of decades in the plot. See *Axes.set_xscale*/*Axes.set_yscale* for details.

nonpositive

[{'mask', 'clip'}, default: 'mask'] Non-positive values can be masked as invalid, or clipped to a very small positive number.

Returns

lines

A list of *Line2D* objects representing the plotted data.

Other Parameters

****kwargs**

All parameters supported by *plot*.

Examples using `matplotlib.axes.Axes.loglog`

- sphx_glr_gallery_subplots_axes_and_figures_secondary_axis.py
- sphx_glr_gallery_scales_log_demo.py

matplotlib.axes.Axes.semilogx

Axes.**semilogx**(*self*, **args*, ***kwargs*)

Make a plot with log scaling on the x axis.

Call signatures:

```
semilogx([x], y, [fmt], data=None, **kwargs)
semilogx([x], y, [fmt], [x2], y2, [fmt2], ..., **kwargs)
```

This is just a thin wrapper around *plot* which additionally changes the x-axis to log scaling. All of the concepts and parameters of plot can be used here as well.

The additional parameters *base*, *subs*, and *nonpositive* control the x-axis properties. They are just forwarded to *Axes.set_xscale*.

Parameters

base

[float, default: 10] Base of the x logarithm.

subs

[array-like, optional] The location of the minor xticks. If *None*, reasonable locations are automatically chosen depending on the number of decades in the plot. See *Axes.set_xscale* for details.

nonpositive

[{'mask', 'clip'}, default: 'mask'] Non-positive values in x can be masked as invalid, or clipped to a very small positive number.

Returns

lines

A list of *Line2D* objects representing the plotted data.

Other Parameters

****kwargs**

All parameters supported by *plot*.

Examples using `matplotlib.axes.Axes.semilogx`

- sphx_glr_gallery_scales_log_demo.py
- sphx_glr_gallery_scales_log_test.py
- *Transformations Tutorial*

matplotlib.axes.Axes.semilogy

`Axes.semilogy(self, *args, **kwargs)`

Make a plot with log scaling on the y axis.

Call signatures:

```
semilogy([x], y, [fmt], data=None, **kwargs)
semilogy([x], y, [fmt], [x2], y2, [fmt2], ..., **kwargs)
```

This is just a thin wrapper around *plot* which additionally changes the y-axis to log scaling. All of the concepts and parameters of plot can be used here as well.

The additional parameters *base*, *subs*, and *nonpositive* control the y-axis properties. They are just forwarded to *Axes.set_yscale*.

Parameters

base

[float, default: 10] Base of the y logarithm.

subs

[array-like, optional] The location of the minor yticks. If *None*, reasonable locations are automatically chosen depending on the number of decades in the plot. See *Axes.set_yscale* for details.

nonpositive

[{'mask', 'clip'}, default: 'mask'] Non-positive values in y can be masked as invalid, or clipped to a very small positive number.

Returns

lines

A list of *Line2D* objects representing the plotted data.

Other Parameters

****kwargs**

All parameters supported by *plot*.

Examples using `matplotlib.axes.Axes.semilogy`

- sphx_glr_gallery_scales_log_demo.py
- sphx_glr_gallery_specialty_plots_skewt.py

matplotlib.axes.Axes.fill_between

`Axes.fill_between`(*self*, *x*, *y1*, *y2=0*, *where=None*, *interpolate=False*, *step=None*, ***, *data=None*, ***kwargs*)

Fill the area between two horizontal curves.

The curves are defined by the points (*x*, *y1*) and (*x*, *y2*). This creates one or multiple polygons describing the filled area.

You may exclude some horizontal sections from filling using *where*.

By default, the edges connect the given points directly. Use *step* if the filling should be a step function, i.e. constant in between *x*.

Parameters

x

[array (length N)] The x coordinates of the nodes defining the curves.

y1

[array (length N) or scalar] The y coordinates of the nodes defining the first curve.

y2

[array (length N) or scalar, default: 0] The y coordinates of the nodes defining the second curve.

where

[array of bool (length N), optional] Define *where* to exclude some horizontal regions from being filled. The filled regions are defined by the coordinates `x[where]`. More precisely, fill between `x[i]` and `x[i+1]` if `where[i] and where[i+1]`. Note that this definition implies that an isolated *True* value between two *False* values in *where* will not result in filling. Both sides of the *True* position remain unfilled due to the adjacent *False* values.

interpolate

[bool, default: False] This option is only relevant if *where* is used and the two curves are crossing each other.

Semantically, *where* is often used for *y1* > *y2* or similar. By default, the nodes of the polygon defining the filled region will only

be placed at the positions in the *x* array. Such a polygon cannot describe the above semantics close to the intersection. The x-sections containing the intersection are simply clipped.

Setting *interpolate* to *True* will calculate the actual intersection point and extend the filled region up to this point.

step

[{'pre', 'post', 'mid'}, optional] Define *step* if the filling should be a step function, i.e. constant in between *x*. The value determines where the step will occur:

- 'pre': The y value is continued constantly to the left from every *x* position, i.e. the interval `(x[i-1], x[i]]` has the value `y[i]`.
- 'post': The y value is continued constantly to the right from every *x* position, i.e. the interval `[x[i], x[i+1])` has the value `y[i]`.
- 'mid': Steps occur half-way between the *x* positions.

Returns

PolyCollection

A *PolyCollection* containing the plotted polygons.

Other Parameters

****kwargs**

All other keyword arguments are passed on to *PolyCollection*. They control the *Polygon* properties:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a
alpha	float or None
animated	bool
antialiased or aa or antialiaseds	bool or list of bools
array	ndarray
capstyle	{'butt', 'round', 'projecting'}
clim	(vmin: float, vmax: float)
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
cmap	*Colormap* or str or None
color	color or list of rgba tuples
contains	unknown
edgecolor or ec or edgecolors	color or list of colors or 'face'

Table 62 – continued from previous page

Property	Description
facecolor or facecolors or fc	color or list of colors
figure	*Figure*
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or dashes or linestyles or ls	str or tuple or list thereof
linewidth or linewidths or lw	float or list of floats
norm	*Normalize* or None
offset_position	unknown
offsets	array-like (N, 2) or (2,)
path_effects	*AbstractPathEffect*
picker	None or bool or callable
pickradius	unknown
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
urls	list of str or None
visible	bool
zorder	float

See also:

fill_between

Fill between two sets of y-values.

fill_betweenx

Fill between two sets of x-values.

Notes

Note: In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, the following arguments can also be string `s`, which is interpreted as `data[s]` (unless this raises an exception): *x*, *y1*, *y2*, *where*.

Objects passed as **data** must support item access (`data[s]`) and membership test (`s in data`).

Examples using `matplotlib.axes.Axes.fill_between`

- sphx_glr_gallery_lines_bars_and_markers_fill_between_demo.py
- sphx_glr_gallery_lines_bars_and_markers_filled_step.py
- sphx_glr_gallery_pyplots_whats_new_98_4_fill_between.py
- sphx_glr_gallery_recipes_fill_between_alpha.py

matplotlib.axes.Axes.fill_betweenx

`Axes.fill_betweenx`(*self*, *y*, *x1*, *x2=0*, *where=None*, *step=None*, *interpolate=False*, *, *data=None*, ***kwargs*)

Fill the area between two vertical curves.

The curves are defined by the points (*y*, *x1*) and (*y*, *x2*). This creates one or multiple polygons describing the filled area.

You may exclude some vertical sections from filling using *where*.

By default, the edges connect the given points directly. Use *step* if the filling should be a step function, i.e. constant in between *y*.

Parameters

y

[array (length N)] The y coordinates of the nodes defining the curves.

x1

[array (length N) or scalar] The x coordinates of the nodes defining the first curve.

x2

[array (length N) or scalar, default: 0] The x coordinates of the nodes defining the second curve.

where

[array of bool (length N), optional] Define *where* to exclude some vertical regions from being filled. The filled regions are defined by the coordinates `y[where]`. More precisely, fill between `y[i]` and `y[i+1]` if `where[i] and where[i+1]`. Note that this definition implies that an isolated *True* value between two *False* values in *where* will not result in filling. Both sides of the *True* position remain unfilled due to the adjacent *False* values.

interpolate

[bool, default: False] This option is only relevant if *where* is used and the two curves are crossing each other.

Semantically, *where* is often used for *x1 > x2* or similar. By default, the nodes of the polygon defining the filled region will only be placed at the positions in the *y* array. Such a polygon cannot describe the above semantics close to the intersection. The y-sections containing the intersection are simply clipped.

Setting *interpolate* to *True* will calculate the actual intersection point and extend the filled region up to this point.

step

[{'pre', 'post', 'mid'}, optional] Define *step* if the filling should be a step function, i.e. constant in between *y*. The value determines where the step will occur:

- 'pre': The y value is continued constantly to the left from every *x* position, i.e. the interval `(x[i-1], x[i]]` has the value `y[i]`.
- 'post': The y value is continued constantly to the right from every *x* position, i.e. the interval `[x[i], x[i+1])` has the value `y[i]`.
- 'mid': Steps occur half-way between the *x* positions.

Returns

PolyCollection

A *PolyCollection* containing the plotted polygons.

Other Parameters

****kwargs**

All other keyword arguments are passed on to *PolyCollection*. They control the *Polygon* properties:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a
alpha	float or None
animated	bool
antialiased or aa or antialiaseds	bool or list of bools
array	ndarray
capstyle	{'butt', 'round', 'projecting'}
clim	(vmin: float, vmax: float)
clip_box	*Bbox*
clip_on	bool

Table 63 – continued from previous page

Property	Description
clip_path	Patch or (Path, Transform) or None
cmap	*Colormap* or str or None
color	color or list of rgba tuples
contains	unknown
edgecolor or ec or edgecolors	color or list of colors or 'face'
facecolor or facecolors or fc	color or list of colors
figure	*Figure*
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or dashes or linestyles or ls	str or tuple or list thereof
linewidth or linewidths or lw	float or list of floats
norm	*Normalize* or None
offset_position	unknown
offsets	array-like (N, 2) or (2,)
path_effects	*AbstractPathEffect*
picker	None or bool or callable
pickradius	unknown
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
urls	list of str or None
visible	bool
zorder	float

See also:

fill_between

Fill between two sets of y-values.

fill_betweenx

Fill between two sets of x-values.

Notes

Note: In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, the following arguments can also be string `s`, which is interpreted as `data[s]` (unless this raises an exception): *y*, *x1*, *x2*, *where*.

Objects passed as **data** must support item access (`data[s]`) and membership test (`s in data`).

Examples using `matplotlib.axes.Axes.fill_betweenx`

- sphx_glr_gallery_lines_bars_and_markers_fill_betweenx_demo.py
- sphx_glr_gallery_lines_bars_and_markers_filled_step.py

matplotlib.axes.Axes.bar

`Axes.bar`(*self*, *x*, *height*, *width=0.8*, *bottom=None*, *, *align='center'*, *data=None*, ***kwargs*)

Make a bar plot.

The bars are positioned at *x* with the given *align*ment. Their dimensions are given by *height* and *width*. The vertical baseline is *bottom* (default 0).

Many parameters can take either a single value applying to all bars or a sequence of values, one for each bar.

Parameters

x

[float or array-like] The x coordinates of the bars. See also *align* for the alignment of the bars to the coordinates.

height

[float or array-like] The height(s) of the bars.

width

[float or array-like, default: 0.8] The width(s) of the bars.

bottom

[float or array-like, default: 0] The y coordinate(s) of the bars bases.

align

[{'center', 'edge'}, default: 'center'] Alignment of the bars to the *x* coordinates:

- 'center': Center the base on the *x* positions.
- 'edge': Align the left edges of the bars with the *x* positions.

To align the bars on the right edge pass a negative *width* and `align='edge'`.

Returns

BarContainer

Container with all the bars and optionally errorbars.

Other Parameters

color

[color or list of color, optional] The colors of the bar faces.

edgecolor

[color or list of color, optional] The colors of the bar edges.

linewidth

[float or array-like, optional] Width of the bar edge(s). If 0, don't draw edges.

tick_label

[str or list of str, optional] The tick labels of the bars. Default: None (Use default numeric labels.)

xerr, yerr

[float or array-like of shape(N,) or shape(2, N), optional] If not *None*, add horizontal / vertical errorbars to the bar tips. The values are +/- sizes relative to the data:

- scalar: symmetric +/- values for all bars
- shape(N,): symmetric +/- values for each bar
- shape(2, N): Separate - and + values for each bar. First row contains the lower errors, the second row contains the upper errors.
- *None*: No errorbar. (Default)

See /gallery/statistics/errorbar_features for an example on the usage of `xerr` and `yerr`.

ecolor

[color or list of color, default: 'black'] The line color of the errorbars.

capsize

[float, default: `rcParams["errorbar.capsize"]` (default: `0.0`)] The length of the error bar caps in points.

error_kw

[dict, optional] Dictionary of kwargs to be passed to the *errorbar* method. Values of *ecolor* or *capsize* defined here take precedence over the independent kwargs.

log

[bool, default: False] If *True*, set the y-axis to be log scale.

****kwargs**

[*Rectangle* properties]

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and retur
alpha	float or None
animated	bool
antialiased or aa	unknown
capstyle	{'butt', 'round', 'projecting'}
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color	color
contains	unknown
edgecolor or ec	color or None or 'auto'
facecolor or fc	color or None
figure	*Figure*
fill	bool
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float or None
path_effects	*AbstractPathEffect*
picker	None or bool or callable
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)

Table 64 – continued from previous page

Property	Description
snap	bool or None
transform	*Transform*
url	str
visible	bool
zorder	float

See also:

barh

Plot a horizontal bar plot.

Notes

Stacked bars can be achieved by passing individual *bottom* values per bar. See /gallery/lines_bars_and_markers/bar_stacked.

Note: In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, every other argument can also be string `s`, which is interpreted as `data[s]` (unless this raises an exception).

Objects passed as **data** must support item access (`data[s]`) and membership test (`s in data`).

Examples using `matplotlib.axes.Axes.bar`

- sphx_glr_gallery_lines_bars_and_markers_bar_stacked.py
- sphx_glr_gallery_lines_bars_and_markers_barchart.py
- sphx_glr_gallery_statistics_barchart_demo.py
- sphx_glr_gallery_pie_and_polar_charts_bar_of_pie.py
- sphx_glr_gallery_pie_and_polar_charts_nested_pie.py
- sphx_glr_gallery_pie_and_polar_charts_polar_bar.py
- sphx_glr_gallery_text_labels_and_annotations_legend_demo.py
- sphx_glr_gallery_shapes_and_collections_hatch_demo.py
- sphx_glr_gallery_style_sheets_ggplot.py
- sphx_glr_gallery_axisartist_demo_floating_axes.py
- sphx_glr_gallery_showcase_xkcd.py

- sphx_glr_gallery_mplot3d_bars3d.py
- sphx_glr_gallery_scales_log_bar.py
- sphx_glr_gallery_ticks_and_spines_custom_ticker1.py
- sphx_glr_gallery_units_bar_unit_demo.py
- *Artist tutorial*
- *Path Tutorial*

matplotlib.axes.Axes.barh

`Axes.barh`(*self*, *y*, *width*, *height=0.8*, *left=None*, *, *align='center'*, **kwargs*)

Make a horizontal bar plot.

The bars are positioned at *y* with the given *align*ment. Their dimensions are given by *width* and *height*. The horizontal baseline is *left* (default 0).

Many parameters can take either a single value applying to all bars or a sequence of values, one for each bar.

Parameters

y

[float or array-like] The y coordinates of the bars. See also *align* for the alignment of the bars to the coordinates.

width

[float or array-like] The width(s) of the bars.

height

[float or array-like, default: 0.8] The heights of the bars.

left

[float or array-like, default: 0] The x coordinates of the left sides of the bars.

align

[{'center', 'edge'}, default: 'center'] Alignment of the base to the *y* coordinates*:

- 'center': Center the bars on the *y* positions.
- 'edge': Align the bottom edges of the bars with the *y* positions.

To align the bars on the top edge pass a negative *height* and `align='edge'`.

Returns

BarContainer

Container with all the bars and optionally errorbars.

Other Parameters

color

[color or list of color, optional] The colors of the bar faces.

edgecolor

[color or list of color, optional] The colors of the bar edges.

linewidth

[float or array-like, optional] Width of the bar edge(s). If 0, don't draw edges.

tick_label

[str or list of str, optional] The tick labels of the bars. Default: None (Use default numeric labels.)

xerr, yerr

[float or array-like of shape(N,) or shape(2, N), optional] If not `None`, add horizontal / vertical errorbars to the bar tips. The values are +/- sizes relative to the data:

- scalar: symmetric +/- values for all bars
- shape(N,): symmetric +/- values for each bar
- shape(2, N): Separate - and + values for each bar. First row contains the lower errors, the second row contains the upper errors.
- *None*: No errorbar. (default)

See /gallery/statistics/errorbar_features for an example on the usage of `xerr` and `yerr`.

ecolor

[color or list of color, default: 'black'] The line color of the errorbars.

capsize

[float, default: `rcParams["errorbar.capsize"]` (default: `0.0`)] The length of the error bar caps in points.

error_kw

[dict, optional] Dictionary of kwargs to be passed to the *errorbar* method. Values of *ecolor* or *capsize* defined here take precedence over the independent kwargs.

log

[bool, default: False] If `True`, set the x-axis to be log scale.

****kwargs**

[*Rectangle* properties]

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and retur
alpha	float or None
animated	bool
antialiased or aa	unknown
capstyle	{'butt', 'round', 'projecting'}
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color	color
contains	unknown
edgecolor or ec	color or None or 'auto'
facecolor or fc	color or None
figure	*Figure*
fill	bool
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float or None
path_effects	*AbstractPathEffect*
picker	None or bool or callable
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
visible	bool
zorder	float

See also:

bar

Plot a vertical bar plot.

Notes

Stacked bars can be achieved by passing individual *left* values per bar. See /gallery/lines_bars_and_markers/horizontal_barchart_distribution .

Examples using `matplotlib.axes.Axes.barh`

- sphx_glr_gallery_lines_bars_and_markers_barh.py
- sphx_glr_gallery_lines_bars_and_markers_horizontal_barchart_distribution.py
- sphx_glr_gallery_statistics_multiple_histograms_side_by_side.py
- *The Lifecycle of a Plot*

matplotlib.axes.Axes.stem

Axes.**stem**(*self*, **args*, *linefmt=None*, *markerfmt=None*, *basefmt=None*, *bottom=0*, *label=None*, *use_line_collection=True*, *data=None*)

Create a stem plot.

A stem plot plots vertical lines at each *x* location from the baseline to *y*, and places a marker there.

Call signature:

```
stem([x,] y, linefmt=None, markerfmt=None, basefmt=None)
```

The x-positions are optional. The formats may be provided either as positional or as keyword-arguments.

Parameters

x

[array-like, optional] The x-positions of the stems. Default: (0, 1, ..., len(y) - 1).

y

[array-like] The y-values of the stem heads.

linefmt

[str, optional] A string defining the properties of the vertical lines. Usually, this will be a color or a color and a linestyle:

Character	Line Style
'-'	solid line
'--'	dashed line
'-.'	dash-dot line
':'	dotted line

Default: 'C0-', i.e. solid line with the first color of the color cycle.

Note: While it is technically possible to specify valid formats other than color or color and linestyle (e.g. 'rx' or '-.'), this is beyond the intention of the method and will most likely not result in a reasonable plot.

markerfmt

[str, optional] A string defining the properties of the markers at the stem heads. Default: 'C0o', i.e. filled circles with the first color of the color cycle.

basefmt

[str, default: 'C3-' ('C2-' in classic mode)] A format string defining the properties of the baseline.

bottom

[float, default: 0] The y-position of the baseline.

label

[str, default: None] The label to use for the stems in legends.

use_line_collection

[bool, default: True] If `True`, store and plot the stem lines as a *LineCollection* instead of individual lines, which significantly increases performance. If `False`, defaults to the old behavior of using a list of *Line2D* objects. This parameter may be deprecated in the future.

Returns

StemContainer

The container may be treated like a tuple (*markerline*, *stemlines*, *baseline*)

Notes

See also:

The MATLAB function stem which inspired this method.

Note: In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, every other argument can also be string `s`, which is interpreted as `data[s]` (unless this raises an exception).

Objects passed as **data** must support item access (`data[s]`) and membership test (`s in data`).

Examples using `matplotlib.axes.Axes.stem`

- sphx_glr_gallery_lines_bars_and_markers_stem_plot.py
- sphx_glr_gallery_text_labels_and_annotations_legend_demo.py

matplotlib.axes.Axes.eventplot

Axes.eventplot(*self*, *positions*, *orientation='horizontal'*, *lineoffsets=1*, *linelengths=1*, *linewidths=None*, *colors=None*, *linestyles='solid'*, ***, *data=None*, ***kwargs*)

Plot identical parallel lines at the given positions.

This type of plot is commonly used in neuroscience for representing neural events, where it is usually called a spike raster, dot raster, or raster plot.

However, it is useful in any situation where you wish to show the timing or position of multiple sets of discrete events, such as the arrival times of people to a business on each day of the month or the date of hurricanes each year of the last century.

Parameters

positions

[array-like or list of array-like] A 1D array-like defines the positions of one sequence of events.

Multiple groups of events may be passed as a list of array-likes. Each group can be styled independently by passing lists of values to *lineoffsets*, *linelengths*, *linewidths*, *colors* and *linestyles*.

Note that *positions* can be a 2D array, but in practice different event groups usually have different counts so that one will use a list of different-length arrays rather than a 2D array.

orientation

[{'horizontal', 'vertical'}, default: 'horizontal'] The direction of the event sequence:

- 'horizontal': the events are arranged horizontally. The indicator lines are vertical.
- 'vertical': the events are arranged vertically. The indicator lines are horizontal.

lineoffsets

[float or array-like, default: 1] The offset of the center of the lines from the origin, in the direction orthogonal to *orientation*.

If *positions* is 2D, this can be a sequence with length matching the length of *positions*.

linelengths

[float or array-like, default: 1] The total height of the lines (i.e. the lines stretches from `lineoffset - linelength/2` to `lineoffset + linelength/2`).

If *positions* is 2D, this can be a sequence with length matching the length of *positions*.

linewidths

[float or array-like, default: `rcParams["lines.linewidth"]` (default: `1.5`)] The line width(s) of the event lines, in points.

If *positions* is 2D, this can be a sequence with length matching the length of *positions*.

colors

[color or list of colors, default: `rcParams["lines.color"]` (default: `'C0'`)] The color(s) of the event lines.

If *positions* is 2D, this can be a sequence with length matching the length of *positions*.

linestyles

[str or tuple or list of such values, default: 'solid'] Default is 'solid'. Valid strings are ['solid', 'dashed', 'dashdot', 'dotted', '-', '--', '-.', ':']. Dash tuples should be of the form:

```
(offset, onoffseq),
```

where *onoffseq* is an even length tuple of on and off ink in points.

If *positions* is 2D, this can be a sequence with length matching the length of *positions*.

****kwargs**

Other keyword arguments are line collection properties. See *LineCollection* for a list of the valid properties.

Returns

list of *EventCollection*

The *EventCollection* that were added.

Notes

For *linelengths*, *linewidths*, *colors*, and *linestyles*, if only a single value is given, that value is applied to all lines. If an array-like is given, it must have the same length as *positions*, and each value will be applied to the corresponding row of the array.

Examples

Note: In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, the following arguments can also be string `s`, which is interpreted as `data[s]` (unless this raises an exception): *positions*, *lineoffsets*, *linelengths*, *linewidths*, *colors*, *linestyles*.

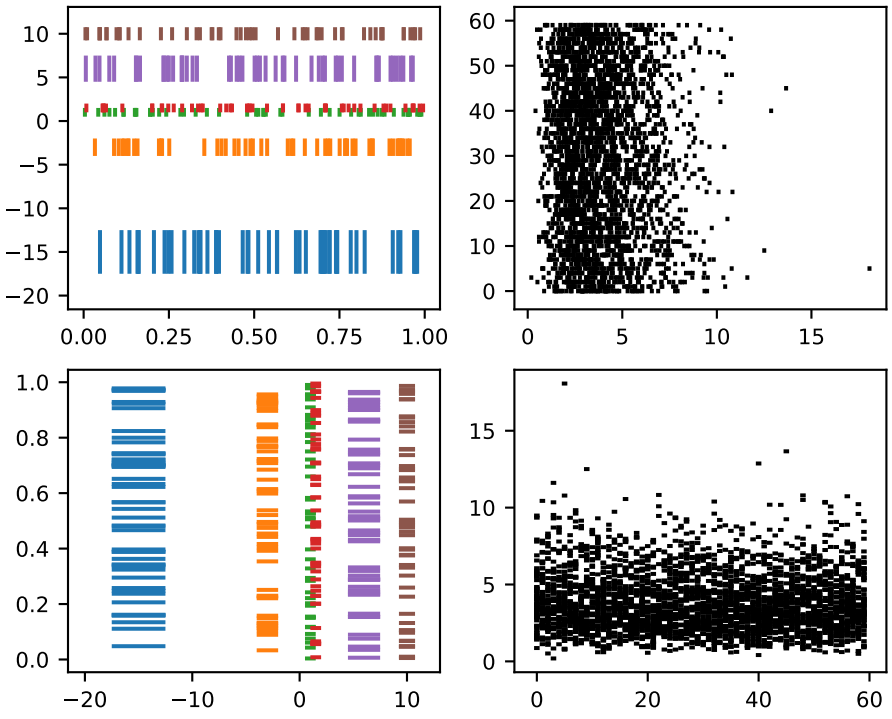
Objects passed as **data** must support item access (`data[s]`) and membership test (`s in data`).

Examples using `matplotlib.axes.Axes.eventplot`

matplotlib.axes.Axes.pie

`Axes.pie`(*self*, *x*, *explode=None*, *labels=None*, *colors=None*, *autopct=None*, *pctdistance=0.6*, *shadow=False*, *labeldistance=1.1*, *startangle=0*, *radius=1*, *counterclock=True*, *wedgeprops=None*, *textprops=None*, *center=0, 0*, *frame=False*, *rotatelabels=False*, ***, *normalize=None*, *data=None*)

Plot a pie chart.



Make a pie chart of array *x*. The fractional area of each wedge is given by `x/sum(x)`. If `sum(x) < 1`, then the values of *x* give the fractional area directly and the array will not be normalized. The resulting pie will have an empty wedge of size `1 - sum(x)`.

The wedges are plotted counterclockwise, by default starting from the x-axis.

Parameters

x

[1D array-like] The wedge sizes.

explode

[array-like, default: None] If not *None*, is a `len(x)` array which specifies the fraction of the radius with which to offset each wedge.

labels

[list, default: None] A sequence of strings providing the labels for each wedge

colors

[array-like, default: None] A sequence of colors through which the pie chart will cycle. If *None*, will use the colors in the currently active cycle.

autopct

[None or str or callable, default: None] If not *None*, is a string or function used to label the wedges with their numeric value. The label will be placed inside the wedge. If it is a format string, the label will be `fmt % pct`. If it is a function, it will be called.

pctdistance

[float, default: 0.6] The ratio between the center of each pie slice and the start of the text generated by *autopct*. Ignored if *autopct* is *None*.

shadow

[bool, default: False] Draw a shadow beneath the pie.

normalize: None or bool, default: None

When *True*, always make a full pie by normalizing x so that `sum(x) == 1`. *False* makes a partial pie if `sum(x) <= 1` and raises a `ValueError` for `sum(x) > 1`.

When *None*, defaults to *True* if `sum(x) >= 1` and *False* if `sum(x) < 1`.

Please note that the previous default value of *None* is now deprecated, and the default will change to *True* in the next release. Please pass `normalize=False` explicitly if you want to draw a partial pie.

labeldistance

[float or None, default: 1.1] The radial distance at which the pie labels are drawn. If set to `None`, label are not drawn, but are stored for use in `legend()`

startangle

[float, default: 0 degrees] The angle by which the start of the pie is rotated, counterclockwise from the x-axis.

radius

[float, default: 1] The radius of the pie.

counterclock

[bool, default: True] Specify fractions direction, clockwise or counterclockwise.

wedgeprops

[dict, default: None] Dict of arguments passed to the wedge objects making the pie. For example, you can pass in `wedgeprops = {'linewidth': 3}` to set the width of the wedge border lines equal to 3. For more details, look at the doc/arguments of the wedge object. By default `clip_on=False`.

textprops

[dict, default: None] Dict of arguments to pass to the text objects.

center

[(float, float), default: (0, 0)] The coordinates of the center of the chart.

frame

[bool, default: False] Plot axes frame with the chart if true.

rotatelabels

[bool, default: False] Rotate each label to the angle of the corresponding slice if true.

Returns

patches

[list] A sequence of *matplotlib.patches.Wedge* instances

texts

[list] A list of the label *Text* instances.

autotexts

[list] A list of *Text* instances for the numeric labels. This will only be returned if the parameter *autopct* is not *None*.

Notes

The pie chart will probably look best if the figure and axes are square, or the Axes aspect is equal. This method sets the aspect ratio of the axis to "equal". The axes aspect ratio can be controlled with *Axes.set_aspect*.

Note: In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, the following arguments can also be string `s`, which is interpreted as `data[s]` (unless this raises an exception): *x*, *explode*, *labels*, *colors*.

Objects passed as **data** must support item access (`data[s]`) and membership test (`s in data`).

Examples using `matplotlib.axes.Axes.pie`

- sphx_glr_gallery_pie_and_polar_charts_pie_features.py
- sphx_glr_gallery_pie_and_polar_charts_pie_demo2.py
- sphx_glr_gallery_pie_and_polar_charts_bar_of_pie.py
- sphx_glr_gallery_pie_and_polar_charts_nested_pie.py
- sphx_glr_gallery_pie_and_polar_charts_pie_and_donut_labels.py
- sphx_glr_gallery_misc_svg_filter_pie.py

matplotlib.axes.Axes.stackplot

Axes.stackplot(*axes*, *x*, **args*, *labels=()*, *colors=None*, *baseline='zero'*, *data=None*, ***kwargs*)

Draw a stacked area plot.

Parameters

x

[1d array of dimension N]

y

[2d array (dimension MxN), or sequence of 1d arrays (each dimension 1xN)] The data is assumed to be unstacked. Each of the following calls is legal:

```
stackplot(x, y)               # where y is MxN
stackplot(x, y1, y2, y3, y4)  # where y1, y2, y3, y4, are all 1xNm
```

baseline

[{'zero', 'sym', 'wiggle', 'weighted_wiggle'}] Method used to calculate the baseline:

- `'zero'`: Constant zero baseline, i.e. a simple stacked plot.
- `'sym'`: Symmetric around zero and is sometimes called 'ThemeRiver'.
- `'wiggle'`: Minimizes the sum of the squared slopes.

- `'weighted_wiggle'`: Does the same but weights to account for size of each layer. It is also called 'Streamgraph'-layout. More details can be found at http://leebyron.com/streamgraph/.

labels

[Length N sequence of strings] Labels to assign to each data series.

colors

[Length N sequence of colors] A list or tuple of colors. These will be cycled through and used to colour the stacked areas.

****kwargs**

All other keyword arguments are passed to *Axes.fill_between*.

Returns

list of *PolyCollection*

A list of *PolyCollection* instances, one for each element in the stacked area plot.

Notes

Note: In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, every other argument can also be string `s`, which is interpreted as `data[s]` (unless this raises an exception).

Objects passed as **data** must support item access (`data[s]`) and membership test (`s in data`).

Examples using `matplotlib.axes.Axes.stackplot`

- sphx_glr_gallery_lines_bars_and_markers_stackplot_demo.py

matplotlib.axes.Axes.broken_barh

Axes.**broken_barh**(*self*, *xranges*, *yrange*, *, *data=None*, **kwargs*)

Plot a horizontal sequence of rectangles.

A rectangle is drawn for each element of *xranges*. All rectangles have the same vertical position and size defined by *yrange*.

This is a convenience function for instantiating a *BrokenBarHCollection*, adding it to the axes and autoscaling the view.

Parameters

xranges

[sequence of tuples (*xmin*, *xwidth*)] The x-positions and extends of the rectangles. For each tuple (*xmin*, *xwidth*) a rectangle is drawn from *xmin* to *xmin* + *xwidth*.

yrange

[(*ymin*, *yheight*)] The y-position and extend for all the rectangles.

Returns

BrokenBarHCollection

Other Parameters

**kwargs

[*BrokenBarHCollection* properties] Each *kwarg* can be either a single argument applying to all rectangles, e.g.:

```
facecolors='black'
```

or a sequence of arguments over which is cycled, e.g.:

```
facecolors=('black', 'blue')
```

would create interleaving black and blue rectangles.

Supported keywords:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a
alpha	float or None
animated	bool
antialiased or aa or antialiaseds	bool or list of bools
array	ndarray

Table 66 – continued from previous page

Property	Description
capstyle	{'butt', 'round', 'projecting'}
clim	(vmin: float, vmax: float)
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
cmap	*Colormap* or str or None
color	color or list of rgba tuples
contains	unknown
edgecolor or ec or edgecolors	color or list of colors or 'face'
facecolor or facecolors or fc	color or list of colors
figure	*Figure*
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or dashes or linestyles or ls	str or tuple or list thereof
linewidth or linewidths or lw	float or list of floats
norm	*Normalize* or None
offset_position	unknown
offsets	array-like (N, 2) or (2,)
path_effects	*AbstractPathEffect*
picker	None or bool or callable
pickradius	unknown
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
urls	list of str or None
visible	bool
zorder	float

Notes

Note: In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, every other argument can also be string `s`, which is interpreted as `data[s]` (unless this raises an exception).

Objects passed as **data** must support item access (`data[s]`) and membership test (`s in data`).

Examples using `matplotlib.axes.Axes.broken_barh`

- sphx_glr_gallery_lines_bars_and_markers_broken_barh.py

matplotlib.axes.Axes.vlines

`Axes.vlines`(*self*, *x*, *ymin*, *ymax*, *colors=None*, *linestyles='solid'*, *label=''*, ***, *data=None*, ***kwargs*)

Plot vertical lines.

Plot vertical lines at each *x* from *ymin* to *ymax*.

Parameters

x

[float or array-like] x-indexes where to plot the lines.

ymin, ymax

[float or array-like] Respective beginning and end of each line. If scalars are provided, all lines will have same length.

colors

[list of colors, default: `rcParams["lines.color"]` (default: `'C0'`)]

linestyles

[{'solid', 'dashed', 'dashdot', 'dotted'}, optional]

label

[str, default: '']

Returns

LineCollection

Other Parameters

****kwargs**

[*LineCollection* properties.]

See also:

hlines

horizontal lines

axvline

vertical line across the axes

Notes

Note: In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, the following arguments can also be string `s`, which is interpreted as `data[s]` (unless this raises an exception): *x*, *ymin*, *ymax*, *colors*.

Objects passed as **data** must support item access (`data[s]`) and membership test (`s in data`).

Examples using `matplotlib.axes.Axes.vlines`

- sphx_glr_gallery_lines_bars_and_markers_timeline.py
- sphx_glr_gallery_lines_bars_and_markers_vline_hline_demo.py
- sphx_glr_gallery_statistics_customized_violin.py

matplotlib.axes.Axes.hlines

`Axes.hlines`(*self*, *y*, *xmin*, *xmax*, *colors=None*, *linestyles='solid'*, *label=''*, ***, *data=None*, ***kwargs*)

Plot horizontal lines at each *y* from *xmin* to *xmax*.

Parameters

y

[float or array-like] y-indexes where to plot the lines.

xmin, xmax

[float or array-like] Respective beginning and end of each line. If scalars are provided, all lines will have same length.

colors

[list of colors, default: `rcParams["lines.color"]` (default: `'C0'`)]

linestyles

[{'solid', 'dashed', 'dashdot', 'dotted'}, optional]

label

[str, default: '']

Returns

LineCollection

Other Parameters

****kwargs**

[*LineCollection* properties.]

See also:

vlines

vertical lines

axhline

horizontal line across the axes

Notes

Note: In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, the following arguments can also be string `s`, which is interpreted as `data[s]` (unless this raises an exception): *y*, *xmin*, *xmax*, *colors*.

Objects passed as **data** must support item access (`data[s]`) and membership test (`s in data`).

Examples using `matplotlib.axes.Axes.hlines`

- sphx_glr_gallery_lines_bars_and_markers_vline_hline_demo.py

matplotlib.axes.Axes.fill

`Axes.fill`(*self*, **args*, *data=None*, ***kwargs*)

Plot filled polygons.

Parameters

***args**

[sequence of x, y, [color]] Each polygon is defined by the lists of *x* and *y* positions of its nodes, optionally followed by a *color* specifier. See *matplotlib.colors* for supported color specifiers. The standard color cycle is used for polygons without a color specifier.

You can plot multiple polygons by providing multiple *x*, *y*, *[color]* groups.

For example, each of the following is legal:

```
ax.fill(x, y)                    # a polygon with default color
ax.fill(x, y, "b")               # a blue polygon
ax.fill(x, y, x2, y2)            # two polygons
ax.fill(x, y, "b", x2, y2, "r")  # a blue and a red polygon
```

data

[indexable object, optional] An object with labelled data. If given, provide the label names to plot in *x* and *y*, e.g.:

```
ax.fill("time", "signal",
        data={"time": [0, 1, 2], "signal": [0, 1, 0]})
```

Returns

list of *Polygon*

Other Parameters

****kwargs**

[*Polygon* properties]

Notes

Use *fill_between()* if you would like to fill the region between two curves.

Examples using `matplotlib.axes.Axes.fill`

- sphx_glr_gallery_lines_bars_and_markers_fill.py
- sphx_glr_gallery_shapes_and_collections_hatch_demo.py
- sphx_glr_gallery_specialty_plots_radar_chart.py
- sphx_glr_gallery_units_ellipse_with_units.py

Spans

Axes.axhline	Add a horizontal line across the axis.
Axes.axhspan	Add a horizontal span (rectangle) across the axis.
Axes.axvline	Add a vertical line across the axes.
Axes.axvspan	Add a vertical span (rectangle) across the axes.
Axes.axline	Add an infinitely long straight line.

matplotlib.axes.Axes.axhline

Axes.**axhline**(*self*, *y=0*, *xmin=0*, *xmax=1*, ***kwargs*)

Add a horizontal line across the axis.

Parameters

y

[float, default: 0] y position in data coordinates of the horizontal line.

xmin

[float, default: 0] Should be between 0 and 1, 0 being the far left of the plot, 1 the far right of the plot.

xmax

[float, default: 1] Should be between 0 and 1, 0 being the far left of the plot, 1 the far right of the plot.

Returns

Line2D

Other Parameters

****kwargs**

Valid keyword arguments are *Line2D* properties, with the exception of 'transform':

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value
alpha	float or None
animated	bool

Table 68 – continued from previous page

Property	Description
antialiased or aa	bool
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color or c	color
contains	unknown
dash_capstyle	{'butt', 'round', 'projecting'}
dash_joinstyle	{'miter', 'round', 'bevel'}
dashes	sequence of floats (on/off ink in points) or (None, None)
data	(2, N) array or two 1D arrays
drawstyle or ds	{'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'd
figure	*Figure*
fillstyle	{'full', 'left', 'right', 'bottom', 'top', 'none'}
gid	str
in_layout	bool
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float
marker	marker style string, *Path* or *MarkerStyle*
markeredgecolor or mec	color
markeredgewidth or mew	float
markerfacecolor or mfc	color
markerfacecoloralt or mfcalt	color
markersize or ms	float
markevery	None or int or (int, int) or slice or List[int] or float or (float, float)
path_effects	*AbstractPathEffect*
picker	unknown
pickradius	float
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
solid_capstyle	{'butt', 'round', 'projecting'}
solid_joinstyle	{'miter', 'round', 'bevel'}
transform	*matplotlib.transforms.Transform*
url	str
visible	bool
xdata	1D array
ydata	1D array
zorder	float

See also:

hlines

Add horizontal lines in data coordinates.

axhspan

Add a horizontal span (rectangle) across the axis.

axline

Add a line with an arbitrary slope.

Examples

- draw a thick red hline at 'y' = 0 that spans the xrange:

```
>>> axhline(linewidth=4, color='r')
```

- draw a default hline at 'y' = 1 that spans the xrange:

```
>>> axhline(y=1)
```

- draw a default hline at 'y' = .5 that spans the middle half of the xrange:

```
>>> axhline(y=.5, xmin=0.25, xmax=0.75)
```

Examples using `matplotlib.axes.Axes.axhline`

- sphx_glr_gallery_lines_bars_and_markers_fill_between_demo.py
- sphx_glr_gallery_lines_bars_and_markers_span_regions.py
- sphx_glr_gallery_subplots_axes_and_figures_axhspan_demo.py
- sphx_glr_gallery_statistics_confidence_ellipse.py
- sphx_glr_gallery_text_labels_and_annotations_usetex_baseline_test.py
- sphx_glr_gallery_pyplots_axline.py
- sphx_glr_gallery_color_color_cycle_default.py
- sphx_glr_gallery_misc_cursor_demo.py
- *Transformations Tutorial*
- *Specifying Colors*

matplotlib.axes.Axes.axhspan

Axes.**axhspan**(*self*, *ymin*, *ymax*, *xmin=0*, *xmax=1*, ***kwargs*)

Add a horizontal span (rectangle) across the axis.

The rectangle spans from *ymin* to *ymax* vertically, and, by default, the whole x-axis horizontally. The x-span can be set using *xmin* (default: 0) and *xmax* (default: 1) which are in axis units; e.g. `xmin = 0.5` always refers to the middle of the x-axis regardless of the limits set by *set_xlim*.

Parameters

ymin

[float] Lower y-coordinate of the span, in data units.

ymax

[float] Upper y-coordinate of the span, in data units.

xmin

[float, default: 0] Lower x-coordinate of the span, in x-axis (0-1) units.

xmax

[float, default: 1] Upper x-coordinate of the span, in x-axis (0-1) units.

Returns

Polygon

Horizontal span (rectangle) from (xmin, ymin) to (xmax, ymax).

Other Parameters

****kwargs**

[*Polygon* properties]

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and retur
alpha	float or None
animated	bool
antialiased or aa	unknown
capstyle	{'butt', 'round', 'projecting'}
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None

Table 69 – continued from previous page

Property	Description
color	color
contains	unknown
edgecolor or ec	color or None or 'auto'
facecolor or fc	color or None
figure	*Figure*
fill	bool
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float or None
path_effects	*AbstractPathEffect*
picker	None or bool or callable
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
visible	bool
zorder	float

See also:

axvspan

Add a vertical span across the axes.

Examples using `matplotlib.axes.Axes.axhspan`

- sphx_glr_gallery_subplots_axes_and_figures_axhspan_demo.py
- *Transformations Tutorial*

matplotlib.axes.Axes.axvline

`Axes.axvline`(*self*, *x=0*, *ymin=0*, *ymax=1*, ***kwargs*)

Add a vertical line across the axes.

Parameters

x

[float, default: 0] x position in data coordinates of the vertical line.

ymin

[float, default: 0] Should be between 0 and 1, 0 being the bottom of the plot, 1 the top of the plot.

ymax

[float, default: 1] Should be between 0 and 1, 0 being the bottom of the plot, 1 the top of the plot.

Returns

Line2D

Other Parameters

****kwargs**

Valid keyword arguments are *Line2D* properties, with the exception of 'transform':

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value
alpha	float or None
animated	bool
antialiased or aa	bool
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color or c	color
contains	unknown
dash_capstyle	{'butt', 'round', 'projecting'}
dash_joinstyle	{'miter', 'round', 'bevel'}
dashes	sequence of floats (on/off ink in points) or (None, None)
data	(2, N) array or two 1D arrays
drawstyle or ds	{'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'd
figure	*Figure*
fillstyle	{'full', 'left', 'right', 'bottom', 'top', 'none'}
gid	str
in_layout	bool
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float
marker	marker style string, *Path* or *MarkerStyle*
markeredgecolor or mec	color

Table 70 – continued from previous page

Property	Description
markeredgewidth or mew	float
markerfacecolor or mfc	color
markerfacecoloralt or mfcalt	color
markersize or ms	float
markevery	None or int or (int, int) or slice or List[int] or float or (float, float)
path_effects	*AbstractPathEffect*
picker	unknown
pickradius	float
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
solid_capstyle	{'butt', 'round', 'projecting'}
solid_joinstyle	{'miter', 'round', 'bevel'}
transform	*matplotlib.transforms.Transform*
url	str
visible	bool
xdata	1D array
ydata	1D array
zorder	float

See also:

vlines

Add vertical lines in data coordinates.

axvspan

Add a vertical span (rectangle) across the axis.

axline

Add a line with an arbitrary slope.

Examples

- draw a thick red vline at $x = 0$ that spans the yrange:

```
>>> axvline(linewidth=4, color='r')
```

- draw a default vline at $x = 1$ that spans the yrange:

```
>>> axvline(x=1)
```

- draw a default vline at $x = .5$ that spans the middle half of the yrange:

```
>>> axvline(x=.5, ymin=0.25, ymax=0.75)
```

Examples using `matplotlib.axes.Axes.axvline`

- sphx_glr_gallery_subplots_axes_and_figures_axhspan_demo.py
- sphx_glr_gallery_statistics_confidence_ellipse.py
- sphx_glr_gallery_text_labels_and_annotations_usetex_baseline_test.py
- sphx_glr_gallery_pyplots_axline.py
- sphx_glr_gallery_color_color_cycle_default.py
- sphx_glr_gallery_misc_cursor_demo.py
- sphx_glr_gallery_specialty_plots_skewt.py
- *The Lifecycle of a Plot*
- *Transformations Tutorial*

matplotlib.axes.Axes.axvspan

Axes.**axvspan**(*self*, *xmin*, *xmax*, *ymin=0*, *ymax=1*, ***kwargs*)

Add a vertical span (rectangle) across the axes.

The rectangle spans from *xmin* to *xmax* horizontally, and, by default, the whole y-axis vertically. The y-span can be set using *ymin* (default: 0) and *ymax* (default: 1) which are in axis units; e.g. `ymin = 0.5` always refers to the middle of the y-axis regardless of the limits set by *set_ylim*.

Parameters

xmin

[float] Lower x-coordinate of the span, in data units.

xmax

[float] Upper x-coordinate of the span, in data units.

ymin

[float, default: 0] Lower y-coordinate of the span, in y-axis units (0-1).

ymax

[float, default: 1] Upper y-coordinate of the span, in y-axis units (0-1).

Returns

Polygon

Vertical span (rectangle) from (xmin, ymin) to (xmax, ymax).

Other Parameters

****kwargs**

[*Polygon* properties]

%(Polygon)s

See also:

axhspan

Add a horizontal span across the axes.

Examples

Draw a vertical, green, translucent rectangle from x = 1.25 to x = 1.55 that spans the yrange of the axes.

```
>>> axvspan(1.25, 1.55, facecolor='g', alpha=0.5)
```

Examples using `matplotlib.axes.Axes.axvspan`

- sphx_glr_gallery_subplots_axes_and_figures_axhspan_demo.py
- *Transformations Tutorial*

matplotlib.axes.Axes.axline

`Axes.axline`(*self*, *xy1*, *xy2=None*, ***, *slope=None*, ***kwargs*)

Add an infinitely long straight line.

The line can be defined either by two points *xy1* and *xy2*, or by one point *xy1* and a *slope*.

This draws a straight line "on the screen", regardless of the x and y scales, and is thus also suitable for drawing exponential decays in semilog plots, power laws in loglog plots, etc. However, *slope* should only be used with linear scales; It has no clear meaning for all other scales, and thus the behavior is undefined. Please specify the line using the points *xy1*, *xy2* for non-linear scales.

Parameters

xy1, xy2

[(float, float)] Points for the line to pass through. Either *xy2* or *slope* has to be given.

slope

[float, optional] The slope of the line. Either *xy2* or *slope* has to be given.

Returns

Line2D

Other Parameters

****kwargs**

Valid kwargs are *Line2D* properties, with the exception of 'transform':

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value
alpha	float or None
animated	bool
antialiased or aa	bool
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color or c	color
contains	unknown
dash_capstyle	{'butt', 'round', 'projecting'}
dash_joinstyle	{'miter', 'round', 'bevel'}
dashes	sequence of floats (on/off ink in points) or (None, None)
data	(2, N) array or two 1D arrays
drawstyle or ds	{'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'd
figure	*Figure*
fillstyle	{'full', 'left', 'right', 'bottom', 'top', 'none'}
gid	str
in_layout	bool
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float
marker	marker style string, *Path* or *MarkerStyle*
markeredgecolor or mec	color
markeredgewidth or mew	float
markerfacecolor or mfc	color

Table 71 – continued from previous page

Property	Description
markerfacecoloralt or mfcalt	color
markersize or ms	float
markevery	None or int or (int, int) or slice or List[int] or float or (float, float)
path_effects	*AbstractPathEffect*
picker	unknown
pickradius	float
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
solid_capstyle	{'butt', 'round', 'projecting'}
solid_joinstyle	{'miter', 'round', 'bevel'}
transform	*matplotlib.transforms.Transform*
url	str
visible	bool
xdata	1D array
ydata	1D array
zorder	float

See also:

axhline

for horizontal lines

axvline

for vertical lines

Examples

Draw a thick red line passing through (0, 0) and (1, 1):

```
>>> axline((0, 0), (1, 1), linewidth=4, color='r')
```

Examples using `matplotlib.axes.Axes.axline`

- sphx_glr_gallery_subplots_axes_and_figures_axhspan_demo.py
- sphx_glr_gallery_pyplots_axline.py

Spectral

Axes.acorr	Plot the autocorrelation of *x*.
Axes.angle_spectrum	Plot the angle spectrum.
Axes.cohere	Plot the coherence between *x* and *y*.
Axes.csd	Plot the cross-spectral density.
Axes.magnitude_spectrum	Plot the magnitude spectrum.
Axes.phase_spectrum	Plot the phase spectrum.
Axes.psd	Plot the power spectral density.
Axes.specgram	Plot a spectrogram.
Axes.xcorr	Plot the cross correlation between *x* and *y*.

matplotlib.axes.Axes.acorr

`Axes.acorr`(*self*, *x*, *, *data=None*, ***kwargs*)

Plot the autocorrelation of *x*.

Parameters

x

[array-like]

detrend

[callable, default: *mlab.detrend_none* (no detrending)] A detrending function applied to *x*. It must have the signature

```
detrend(x: np.ndarray) -> np.ndarray
```

normed

[bool, default: True] If `True`, input vectors are normalised to unit length.

usevlines

[bool, default: True] Determines the plot style.

If `True`, vertical lines are plotted from 0 to the acorr value using *Axes.vlines*. Additionally, a horizontal line is plotted at y=0 using *Axes.axhline*.

If `False`, markers are plotted at the acorr values using *Axes.plot*.

maxlags

[int, default: 10] Number of lags to show. If `None`, will return all `2 * len(x) - 1` lags.

Returns

lags

[array (length `2*maxlags+1`)] The lag vector.

c

[array (length `2*maxlags+1`)] The auto correlation vector.

line

[*LineCollection* or *Line2D*] *Artist* added to the axes of the correlation:

- *LineCollection* if *usevlines* is True.
- *Line2D* if *usevlines* is False.

b

[*Line2D* or None] Horizontal line at 0 if *usevlines* is True None *usevlines* is False.

Other Parameters

linestyle

[*Line2D* property, optional] The linestyle for plotting the data points. Only used if *usevlines* is `False`.

marker

[str, default: 'o'] The marker for plotting the data points. Only used if *usevlines* is `False`.

****kwargs**

Additional parameters are passed to *Axes.vlines* and *Axes.axhline* if *usevlines* is `True`; otherwise they are passed to *Axes.plot*.

Notes

The cross correlation is performed with `numpy.correlate` with `mode = "full"`.

Note: In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, the following arguments can also be string `s`, which is interpreted as `data[s]` (unless this raises an exception): *x*.

Objects passed as **data** must support item access (`data[s]`) and membership test (`s in data`).

Examples using `matplotlib.axes.Axes.acorr`

- sphx_glr_gallery_lines_bars_and_markers_xcorr_acorr_demo.py

matplotlib.axes.Axes.angle_spectrum

Axes.**angle_spectrum**(*self*, *x*, *Fs=None*, *Fc=None*, *window=None*, *pad_to=None*, *sides=None*, *, *data=None*, ***kwargs*)

Plot the angle spectrum.

Compute the angle spectrum (wrapped phase spectrum) of *x*. Data is padded to a length of *pad_to* and the windowing function *window* is applied to the signal.

Parameters

x

[1-D array or sequence] Array or sequence containing the data.

Fs

[float, default: 2] The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, *freqs*, in cycles per time unit.

window

[callable or ndarray, default: *window_hanning*] A function or a vector of length *NFFT*. To create window vectors see *window_hanning*, *window_none*, `numpy.blackman`, `numpy.hamming`, `numpy.bartlett`, `scipy.signal`, `scipy.signal.get_window`, etc. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

sides

[{'default', 'onesided', 'twosided'}, optional] Which sides of the spectrum to return. 'default' is one-sided for real data and two-sided for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

pad_to

[int, optional] The number of points to which the data segment is padded when performing the FFT. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to fft(). The default is None, which sets *pad_to* equal to the length of the input signal (i.e. no padding).

Fc

[int, default: 0] The center frequency of *x*, which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

Returns

spectrum

[1-D array] The values for the angle spectrum in radians (real valued).

freqs

[1-D array] The frequencies corresponding to the elements in *spectrum*.

line

[*Line2D*] The line created by this function.

Other Parameters

****kwargs**

Keyword arguments control the *Line2D* properties:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value
alpha	float or None
animated	bool
antialiased or aa	bool
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color or c	color
contains	unknown
dash_capstyle	{'butt', 'round', 'projecting'}
dash_joinstyle	{'miter', 'round', 'bevel'}
dashes	sequence of floats (on/off ink in points) or (None, None)
data	(2, N) array or two 1D arrays
drawstyle or ds	{'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'd
figure	*Figure*
fillstyle	{'full', 'left', 'right', 'bottom', 'top', 'none'}
gid	str
in_layout	bool
label	object

Table 73 – continued from previous page

Property	Description
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float
marker	marker style string, *Path* or *MarkerStyle*
markeredgecolor or mec	color
markeredgewidth or mew	float
markerfacecolor or mfc	color
markerfacecoloralt or mfcalt	color
markersize or ms	float
markevery	None or int or (int, int) or slice or List[int] or float or (float, float)
path_effects	*AbstractPathEffect*
picker	unknown
pickradius	float
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
solid_capstyle	{'butt', 'round', 'projecting'}
solid_joinstyle	{'miter', 'round', 'bevel'}
transform	*matplotlib.transforms.Transform*
url	str
visible	bool
xdata	1D array
ydata	1D array
zorder	float

See also:

magnitude_spectrum

Plots the magnitudes of the corresponding frequencies.

phase_spectrum

Plots the unwrapped version of this function.

specgram

Can plot the angle spectrum of segments within the signal in a colormap.

Notes

Note: In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, the following arguments can also be string `s`, which is interpreted as `data[s]` (unless this raises an exception): *x*.

Objects passed as **data** must support item access (`data[s]`) and membership test (`s in data`).

Examples using `matplotlib.axes.Axes.angle_spectrum`

matplotlib.axes.Axes.cohere

`Axes.cohere`(*self*, *x*, *y*, *NFFT=256*, *Fs=2*, *Fc=0*, *detrend=<function detrend_none at 0x7f5434697820>*, *window=<function window_hanning at 0x7f5434697160>*, *noverlap=0*, *pad_to=None*, *sides='default'*, *scale_by_freq=None*, ***, *data=None*, ***kwargs*)

Plot the coherence between *x* and *y*.

Plot the coherence between *x* and *y*. Coherence is the normalized cross spectral density:

$$C_{xy} = \frac{|P_{xy}|^2}{P_{xx}P_{yy}}$$

Parameters

Fs

[float, default: 2] The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, *freqs*, in cycles per time unit.

window

[callable or ndarray, default: *window_hanning*] A function or a vector of length *NFFT*. To create window vectors see *window_hanning*, *window_none*, `numpy.blackman`, `numpy.hamming`, `numpy.bartlett`, `scipy.signal`, `scipy.signal.get_window`, etc. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

sides

[{'default', 'onesided', 'twosided'}, optional] Which sides of the spectrum to return. 'default' is one-sided for real data and two-sided for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

pad_to

[int, optional] The number of points to which the data segment is padded when performing the FFT. This can be different from *NFFT*, which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to fft(). The default is None, which sets *pad_to* equal to *NFFT*

NFFT

[int, default: 256] The number of data points used in each block for the FFT. A power 2 is most efficient. This should *NOT* be used to get zero padding, or the scaling of the result will be incorrect; use *pad_to* for this instead.

detrend

[{'none', 'mean', 'linear'} or callable, default 'none'] The function applied to each segment before fft-ing, designed to remove the mean or linear trend. Unlike in MATLAB, where the *detrend* parameter is a vector, in Matplotlib is it a function. The *mlab* module defines *detrend_none*, *detrend_mean*, and *detrend_linear*, but you can use a custom function as well. You can also use a string to choose one of the functions: 'none' calls *detrend_none*. 'mean' calls *detrend_mean*. 'linear' calls *detrend_linear*.

scale_by_freq

[bool, default: True] Whether the resulting density values should be scaled by the scaling frequency, which gives density in units of Hz^-1. This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

noverlap

[int, default: 0 (no overlap)] The number of points of overlap between blocks.

Fc

[int, default: 0] The center frequency of *x*, which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

Returns

Cxy

[1-D array] The coherence vector.

freqs

[1-D array] The frequencies for the elements in *Cxy*.

Other Parameters

****kwargs**

Keyword arguments control the *Line2D* properties:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value
alpha	float or None
animated	bool
antialiased or aa	bool
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color or c	color
contains	unknown
dash_capstyle	{'butt', 'round', 'projecting'}
dash_joinstyle	{'miter', 'round', 'bevel'}
dashes	sequence of floats (on/off ink in points) or (None, None)
data	(2, N) array or two 1D arrays
drawstyle or ds	{'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'd
figure	*Figure*
fillstyle	{'full', 'left', 'right', 'bottom', 'top', 'none'}
gid	str
in_layout	bool
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float
marker	marker style string, *Path* or *MarkerStyle*
markeredgecolor or mec	color
markeredgewidth or mew	float
markerfacecolor or mfc	color
markerfacecoloralt or mfcalt	color
markersize or ms	float
markevery	None or int or (int, int) or slice or List[int] or float or (float, float)
path_effects	*AbstractPathEffect*
picker	unknown
pickradius	float
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
solid_capstyle	{'butt', 'round', 'projecting'}

Table 74 – continued from previous page

Property	Description
solid_joinstyle	{'miter', 'round', 'bevel'}
transform	*matplotlib.transforms.Transform*
url	str
visible	bool
xdata	1D array
ydata	1D array
zorder	float

Notes

Note: In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, the following arguments can also be string `s`, which is interpreted as `data[s]` (unless this raises an exception): *x*, *y*.

Objects passed as **data** must support item access (`data[s]`) and membership test (`s in data`).

References

Bendat & Piersol -- Random Data: Analysis and Measurement Procedures, John Wiley & Sons (1986)

Examples using `matplotlib.axes.Axes.cohere`

matplotlib.axes.Axes.csd

Axes.csd(*self*, *x*, *y*, *NFFT=None*, *Fs=None*, *Fc=None*, *detrend=None*, *window=None*, *noverlap=None*, *pad_to=None*, *sides=None*, *scale_by_freq=None*, *return_line=None*, *, *data=None*, **kwargs*)

Plot the cross-spectral density.

The cross spectral density P_{xy} by Welch's average periodogram method. The vectors *x* and *y* are divided into *NFFT* length segments. Each segment is detrended by function *detrend* and windowed by function *window*. *noverlap* gives the length of the overlap between segments. The product of the direct FFTs of *x* and *y* are averaged over each segment to compute P_{xy}, with a scaling to correct for power loss due to windowing.

If len(*x*) < *NFFT* or len(*y*) < *NFFT*, they will be zero padded to *NFFT*.

Parameters

x, y

[1-D arrays or sequences] Arrays or sequences containing the data.

Fs

[float, default: 2] The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, *freqs*, in cycles per time unit.

window

[callable or ndarray, default: *window_hanning*] A function or a vector of length *NFFT*. To create window vectors see *window_hanning*, *window_none*, numpy.blackman, numpy.hamming, numpy.bartlett, scipy.signal, scipy.signal.get_window, etc. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

sides

[{'default', 'onesided', 'twosided'}, optional] Which sides of the spectrum to return. 'default' is one-sided for real data and two-sided for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

pad_to

[int, optional] The number of points to which the data segment is padded when performing the FFT. This can be different from *NFFT*, which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to fft(). The default is None, which sets *pad_to* equal to *NFFT*

NFFT

[int, default: 256] The number of data points used in each block for the FFT. A power 2 is most efficient. This should *NOT* be used to get zero padding, or the scaling of the result will be incorrect; use *pad_to* for this instead.

detrend

[{'none', 'mean', 'linear'} or callable, default 'none'] The function applied to each segment before fft-ing, designed to remove the mean or linear trend. Unlike in MATLAB, where the *detrend* parameter is a vector, in Matplotlib is it a function. The *mlab* module defines *detrend_none*, *detrend_mean*, and *detrend_linear*, but you can use a custom function as well. You can also use a

string to choose one of the functions: 'none' calls *detrend_none*. 'mean' calls *detrend_mean*. 'linear' calls *detrend_linear*.

scale_by_freq

[bool, default: True] Whether the resulting density values should be scaled by the scaling frequency, which gives density in units of Hz^-1. This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

noverlap

[int, default: 0 (no overlap)] The number of points of overlap between segments.

Fc

[int, default: 0] The center frequency of *x*, which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

return_line

[bool, default: False] Whether to include the line object plotted in the returned values.

Returns

Pxy

[1-D array] The values for the cross spectrum P_{xy} before scaling (complex valued).

freqs

[1-D array] The frequencies corresponding to the elements in *Pxy*.

line

[*Line2D*] The line created by this function. Only returned if *return_line* is True.

Other Parameters

****kwargs**

Keyword arguments control the *Line2D* properties:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value
alpha	float or None

Table 75 – continued from previous page

Property	Description
animated	bool
antialiased or aa	bool
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color or c	color
contains	unknown
dash_capstyle	{'butt', 'round', 'projecting'}
dash_joinstyle	{'miter', 'round', 'bevel'}
dashes	sequence of floats (on/off ink in points) or (None, None)
data	(2, N) array or two 1D arrays
drawstyle or ds	{'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'd
figure	*Figure*
fillstyle	{'full', 'left', 'right', 'bottom', 'top', 'none'}
gid	str
in_layout	bool
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float
marker	marker style string, *Path* or *MarkerStyle*
markeredgecolor or mec	color
markeredgewidth or mew	float
markerfacecolor or mfc	color
markerfacecoloralt or mfcalt	color
markersize or ms	float
markevery	None or int or (int, int) or slice or List[int] or float or (float, float)
path_effects	*AbstractPathEffect*
picker	unknown
pickradius	float
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
solid_capstyle	{'butt', 'round', 'projecting'}
solid_joinstyle	{'miter', 'round', 'bevel'}
transform	*matplotlib.transforms.Transform*
url	str
visible	bool
xdata	1D array
ydata	1D array
zorder	float

See also:

psd

is equivalent to setting `y = x`.

Notes

For plotting, the power is plotted as $10\log_{10}(P_{xy})$ for decibels, though P_{xy} itself is returned.

References

Bendat & Piersol -- Random Data: Analysis and Measurement Procedures, John Wiley & Sons (1986)

Note: In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, the following arguments can also be string `s`, which is interpreted as `data[s]` (unless this raises an exception): *x*, *y*.

Objects passed as **data** must support item access (`data[s]`) and membership test (`s in data`).

Examples using `matplotlib.axes.Axes.csd`

- sphx_glr_gallery_lines_bars_and_markers_csd_demo.py

matplotlib.axes.Axes.magnitude_spectrum

Axes.**magnitude_spectrum**(*self*, *x*, *Fs=None*, *Fc=None*, *window=None*, *pad_to=None*, *sides=None*, *scale=None*, *, *data=None*, ***kwargs*)

Plot the magnitude spectrum.

Compute the magnitude spectrum of *x*. Data is padded to a length of *pad_to* and the windowing function *window* is applied to the signal.

Parameters

x

[1-D array or sequence] Array or sequence containing the data.

Fs

[float, default: 2] The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, *freqs*, in cycles per time unit.

window

[callable or ndarray, default: *window_hanning*] A function or a vector of length *NFFT*. To create window vectors see *window_hanning*, *window_none*, numpy.blackman, numpy.hamming, numpy.bartlett, scipy.signal, scipy.signal.get_window, etc. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

sides

[{'default', 'onesided', 'twosided'}, optional] Which sides of the spectrum to return. 'default' is one-sided for real data and two-sided for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

pad_to

[int, optional] The number of points to which the data segment is padded when performing the FFT. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to fft(). The default is None, which sets *pad_to* equal to the length of the input signal (i.e. no padding).

scale

[{'default', 'linear', 'dB'}] The scaling of the values in the *spec*. 'linear' is no scaling. 'dB' returns the values in dB scale, i.e., the dB amplitude (20 * log10). 'default' is 'linear'.

Fc

[int, default: 0] The center frequency of *x*, which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

Returns

spectrum

[1-D array] The values for the magnitude spectrum before scaling (real valued).

freqs

[1-D array] The frequencies corresponding to the elements in *spectrum*.

line

[*Line2D*] The line created by this function.

Other Parameters

**kwargs

Keyword arguments control the *Line2D* properties:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value
alpha	float or None
animated	bool
antialiased or aa	bool
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color or c	color
contains	unknown
dash_capstyle	{'butt', 'round', 'projecting'}
dash_joinstyle	{'miter', 'round', 'bevel'}
dashes	sequence of floats (on/off ink in points) or (None, None)
data	(2, N) array or two 1D arrays
drawstyle or ds	{'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'd
figure	*Figure*
fillstyle	{'full', 'left', 'right', 'bottom', 'top', 'none'}
gid	str
in_layout	bool
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float
marker	marker style string, *Path* or *MarkerStyle*
markeredgecolor or mec	color
markeredgewidth or mew	float
markerfacecolor or mfc	color
markerfacecoloralt or mfcalt	color
markersize or ms	float
markevery	None or int or (int, int) or slice or List[int] or float or (float, float)
path_effects	*AbstractPathEffect*
picker	unknown
pickradius	float
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
solid_capstyle	{'butt', 'round', 'projecting'}
solid_joinstyle	{'miter', 'round', 'bevel'}
transform	*matplotlib.transforms.Transform*
url	str
visible	bool

Table 76 – continued from previous page

Property	Description
xdata	1D array
ydata	1D array
zorder	float

See also:

psd

Plots the power spectral density.

angle_spectrum

Plots the angles of the corresponding frequencies.

phase_spectrum

Plots the phase (unwrapped angle) of the corresponding frequencies.

specgram

Can plot the magnitude spectrum of segments within the signal in a colormap.

Notes

Note: In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, the following arguments can also be string `s`, which is interpreted as `data[s]` (unless this raises an exception): *x*.

Objects passed as **data** must support item access (`data[s]`) and membership test (`s in data`).

Examples using `matplotlib.axes.Axes.magnitude_spectrum`

matplotlib.axes.Axes.phase_spectrum

Axes.**phase_spectrum**(*self*, *x*, *Fs=None*, *Fc=None*, *window=None*, *pad_to=None*, *sides=None*, *, *data=None*, **kwargs*)

Plot the phase spectrum.

Compute the phase spectrum (unwrapped angle spectrum) of *x*. Data is padded to a length of *pad_to* and the windowing function *window* is applied to the signal.

Parameters

x

[1-D array or sequence] Array or sequence containing the data

Fs

[float, default: 2] The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, *freqs*, in cycles per time unit.

window

[callable or ndarray, default: *window_hanning*] A function or a vector of length *NFFT*. To create window vectors see *window_hanning*, *window_none*, numpy.blackman, numpy.hamming, numpy.bartlett, scipy.signal, scipy.signal.get_window, etc. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

sides

[{'default', 'onesided', 'twosided'}, optional] Which sides of the spectrum to return. 'default' is one-sided for real data and two-sided for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

pad_to

[int, optional] The number of points to which the data segment is padded when performing the FFT. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to fft(). The default is None, which sets *pad_to* equal to the length of the input signal (i.e. no padding).

Fc

[int, default: 0] The center frequency of *x*, which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

Returns

spectrum

[1-D array] The values for the phase spectrum in radians (real valued).

freqs

[1-D array] The frequencies corresponding to the elements in *spectrum*.

line

[*Line2D*] The line created by this function.

Other Parameters

****kwargs**

Keyword arguments control the *Line2D* properties:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value
alpha	float or None
animated	bool
antialiased or aa	bool
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color or c	color
contains	unknown
dash_capstyle	{'butt', 'round', 'projecting'}
dash_joinstyle	{'miter', 'round', 'bevel'}
dashes	sequence of floats (on/off ink in points) or (None, None)
data	(2, N) array or two 1D arrays
drawstyle or ds	{'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'd
figure	*Figure*
fillstyle	{'full', 'left', 'right', 'bottom', 'top', 'none'}
gid	str
in_layout	bool
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float
marker	marker style string, *Path* or *MarkerStyle*
markeredgecolor or mec	color
markeredgewidth or mew	float
markerfacecolor or mfc	color
markerfacecoloralt or mfcalt	color
markersize or ms	float
markevery	None or int or (int, int) or slice or List[int] or float or (float, float)
path_effects	*AbstractPathEffect*
picker	unknown
pickradius	float
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
solid_capstyle	{'butt', 'round', 'projecting'}

Table 77 – continued from previous page

Property	Description
solid_joinstyle	{'miter', 'round', 'bevel'}
transform	*matplotlib.transforms.Transform*
url	str
visible	bool
xdata	1D array
ydata	1D array
zorder	float

See also:

magnitude_spectrum

Plots the magnitudes of the corresponding frequencies.

angle_spectrum

Plots the wrapped version of this function.

specgram

Can plot the phase spectrum of segments within the signal in a colormap.

Notes

Note: In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, the following arguments can also be string `s`, which is interpreted as `data[s]` (unless this raises an exception): *x*.

Objects passed as **data** must support item access (`data[s]`) and membership test (`s in data`).

Examples using `matplotlib.axes.Axes.phase_spectrum`

matplotlib.axes.Axes.psd

`Axes.psd`(*self*, *x*, *NFFT=None*, *Fs=None*, *Fc=None*, *detrend=None*, *window=None*, *noverlap=None*, *pad_to=None*, *sides=None*, *scale_by_freq=None*, *return_line=None*, ***, *data=None*, ***kwargs*)

Plot the power spectral density.

The power spectral density P_{xx} by Welch's average periodogram method. The vector *x* is divided into *NFFT* length segments. Each segment is detrended by function *detrend* and windowed by function *window*. *noverlap* gives the length

of the overlap between segments. The $|\mathrm{fft}(i)|^2$ of each segment i are averaged to compute P_{xx}, with a scaling to correct for power loss due to windowing.

If len(*x*) < *NFFT*, it will be zero padded to *NFFT*.

Parameters

x

[1-D array or sequence] Array or sequence containing the data

Fs

[float, default: 2] The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, *freqs*, in cycles per time unit.

window

[callable or ndarray, default: *window_hanning*] A function or a vector of length *NFFT*. To create window vectors see *window_hanning*, *window_none*, numpy.blackman, numpy.hamming, numpy.bartlett, scipy.signal, scipy.signal.get_window, etc. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

sides

[{'default', 'onesided', 'twosided'}, optional] Which sides of the spectrum to return. 'default' is one-sided for real data and two-sided for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

pad_to

[int, optional] The number of points to which the data segment is padded when performing the FFT. This can be different from *NFFT*, which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to fft(). The default is None, which sets *pad_to* equal to *NFFT*

NFFT

[int, default: 256] The number of data points used in each block for the FFT. A power 2 is most efficient. This should *NOT* be used to get zero padding, or the scaling of the result will be incorrect; use *pad_to* for this instead.

detrend

[{'none', 'mean', 'linear'} or callable, default 'none'] The function applied to each segment before fft-ing, designed to remove the

mean or linear trend. Unlike in MATLAB, where the *detrend* parameter is a vector, in Matplotlib is it a function. The *mlab* module defines *detrend_none*, *detrend_mean*, and *detrend_linear*, but you can use a custom function as well. You can also use a string to choose one of the functions: 'none' calls *detrend_none*. 'mean' calls *detrend_mean*. 'linear' calls *detrend_linear*.

scale_by_freq

[bool, default: True] Whether the resulting density values should be scaled by the scaling frequency, which gives density in units of Hz^-1. This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

noverlap

[int, default: 0 (no overlap)] The number of points of overlap between segments.

Fc

[int, default: 0] The center frequency of *x*, which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

return_line

[bool, default: False] Whether to include the line object plotted in the returned values.

Returns

Pxx

[1-D array] The values for the power spectrum P_{xx} before scaling (real valued).

freqs

[1-D array] The frequencies corresponding to the elements in *Pxx*.

line

[*Line2D*] The line created by this function. Only returned if *return_line* is True.

Other Parameters

****kwargs**

Keyword arguments control the *Line2D* properties:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value
alpha	float or None
animated	bool
antialiased or aa	bool
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color or c	color
contains	unknown
dash_capstyle	{'butt', 'round', 'projecting'}
dash_joinstyle	{'miter', 'round', 'bevel'}
dashes	sequence of floats (on/off ink in points) or (None, None)
data	(2, N) array or two 1D arrays
drawstyle or ds	{'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'd
figure	*Figure*
fillstyle	{'full', 'left', 'right', 'bottom', 'top', 'none'}
gid	str
in_layout	bool
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float
marker	marker style string, *Path* or *MarkerStyle*
markeredgecolor or mec	color
markeredgewidth or mew	float
markerfacecolor or mfc	color
markerfacecoloralt or mfcalt	color
markersize or ms	float
markevery	None or int or (int, int) or slice or List[int] or float or (float, float)
path_effects	*AbstractPathEffect*
picker	unknown
pickradius	float
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
solid_capstyle	{'butt', 'round', 'projecting'}
solid_joinstyle	{'miter', 'round', 'bevel'}
transform	*matplotlib.transforms.Transform*
url	str
visible	bool
xdata	1D array
ydata	1D array
zorder	float

See also:

specgram

Differs in the default overlap; in not returning the mean of the segment periodograms; in returning the times of the segments; and in plotting a colormap instead of a line.

magnitude_spectrum

Plots the magnitude spectrum.

csd

Plots the spectral density between two signals.

Notes

For plotting, the power is plotted as $10 \log_{10}(P_{xx})$ for decibels, though *Pxx* itself is returned.

References

Bendat & Piersol -- Random Data: Analysis and Measurement Procedures, John Wiley & Sons (1986)

Note: In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, the following arguments can also be string `s`, which is interpreted as `data[s]` (unless this raises an exception): *x*.

Objects passed as **data** must support item access (`data[s]`) and membership test (`s in data`).

Examples using `matplotlib.axes.Axes.psd`

- sphx_glr_gallery_lines_bars_and_markers_psd_demo.py

matplotlib.axes.Axes.specgram

Axes.**specgram**(*self*, *x*, *NFFT=None*, *Fs=None*, *Fc=None*, *detrend=None*, *window=None*, *noverlap=None*, *cmap=None*, *xextent=None*, *pad_to=None*, *sides=None*, *scale_by_freq=None*, *mode=None*, *scale=None*, *vmin=None*, *vmax=None*, ***, *data=None*, ***kwargs*)

Plot a spectrogram.

Compute and plot a spectrogram of data in *x*. Data are split into *NFFT* length segments and the spectrum of each section is computed. The windowing function *window* is applied to each segment, and the amount of overlap of each segment is specified with *noverlap*. The spectrogram is plotted as a colormap (using imshow).

Parameters

x

[1-D array or sequence] Array or sequence containing the data.

Fs

[float, default: 2] The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, *freqs*, in cycles per time unit.

window

[callable or ndarray, default: *window_hanning*] A function or a vector of length *NFFT*. To create window vectors see *window_hanning*, *window_none*, numpy.blackman, numpy.hamming, numpy.bartlett, scipy.signal, scipy.signal.get_window, etc. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

sides

[{'default', 'onesided', 'twosided'}, optional] Which sides of the spectrum to return. 'default' is one-sided for real data and two-sided for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

pad_to

[int, optional] The number of points to which the data segment is padded when performing the FFT. This can be different from *NFFT*, which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to fft(). The default is None, which sets *pad_to* equal to *NFFT*

NFFT

[int, default: 256] The number of data points used in each block for the FFT. A power 2 is most efficient. This should *NOT* be used to get zero padding, or the scaling of the result will be incorrect; use *pad_to* for this instead.

detrend

[{'none', 'mean', 'linear'} or callable, default 'none'] The function applied to each segment before fft-ing, designed to remove the mean or linear trend. Unlike in MATLAB, where the *detrend* parameter is a vector, in Matplotlib is it a function. The *mlab* module defines *detrend_none*, *detrend_mean*, and *detrend_linear*, but you can use a custom function as well. You can also use a string to choose one of the functions: 'none' calls *detrend_none*. 'mean' calls *detrend_mean*. 'linear' calls *detrend_linear*.

scale_by_freq

[bool, default: True] Whether the resulting density values should be scaled by the scaling frequency, which gives density in units of Hz^-1. This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

mode

[{'default', 'psd', 'magnitude', 'angle', 'phase'}] What sort of spectrum to use. Default is 'psd', which takes the power spectral density. 'magnitude' returns the magnitude spectrum. 'angle' returns the phase spectrum without unwrapping. 'phase' returns the phase spectrum with unwrapping.

noverlap

[int] The number of points of overlap between blocks. The default value is 128.

scale

[{'default', 'linear', 'dB'}] The scaling of the values in the *spec*. 'linear' is no scaling. 'dB' returns the values in dB scale. When *mode* is 'psd', this is dB power (10 * log10). Otherwise this is dB amplitude (20 * log10). 'default' is 'dB' if *mode* is 'psd' or 'magnitude' and 'linear' otherwise. This must be 'linear' if *mode* is 'angle' or 'phase'.

Fc

[int, default: 0] The center frequency of *x*, which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

cmap

[*Colormap*, default: `rcParams["image.cmap"]` (default: `'viridis'`)]

xextent

[*None* or (xmin, xmax)] The image extent along the x-axis. The default sets *xmin* to the left border of the first bin (*spectrum* column) and *xmax* to the right border of the last bin. Note that

for *noverlap>0* the width of the bins is smaller than those of the segments.

****kwargs**

Additional keyword arguments are passed on to *imshow* which makes the specgram image.

Returns

spectrum

[2-D array] Columns are the periodograms of successive segments.

freqs

[1-D array] The frequencies corresponding to the rows in *spectrum*.

t

[1-D array] The times corresponding to midpoints of segments (i.e., the columns in *spectrum*).

im

[*AxesImage*] The image created by imshow containing the spectrogram.

See also:

psd

Differs in the default overlap; in returning the mean of the segment periodograms; in not returning times; and in generating a line plot instead of colormap.

magnitude_spectrum

A single spectrum, similar to having a single segment when *mode* is 'magnitude'. Plots a line instead of a colormap.

angle_spectrum

A single spectrum, similar to having a single segment when *mode* is 'angle'. Plots a line instead of a colormap.

phase_spectrum

A single spectrum, similar to having a single segment when *mode* is 'phase'. Plots a line instead of a colormap.

Notes

The parameters *detrend* and *scale_by_freq* do only apply when *mode* is set to 'psd'.

Note: In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, the following arguments can also be string `s`, which is interpreted as `data[s]` (unless this raises an exception): *x*.

Objects passed as **data** must support item access (`data[s]`) and membership test (`s in data`).

Examples using `matplotlib.axes.Axes.specgram`

- sphx_glr_gallery_images_contours_and_fields_specgram_demo.py

matplotlib.axes.Axes.xcorr

Axes.xcorr(*self*, *x*, *y*, *normed=True*, *detrend=<function detrend_none at 0x7f5434697820>*, *usevlines=True*, *maxlags=10*, ***, *data=None*, ***kwargs*)

Plot the cross correlation between *x* and *y*.

The correlation with lag k is defined as $\sum_n x[n+k] \cdot y^*[n]$, where y^* is the complex conjugate of y.

Parameters

x, y

[array-like of length n]

detrend

[callable, default: *mlab.detrend_none* (no detrending)] A detrending function applied to *x* and *y*. It must have the signature

```
detrend(x: np.ndarray) -> np.ndarray
```

normed

[bool, default: True] If `True`, input vectors are normalised to unit length.

usevlines

[bool, default: True] Determines the plot style.

If `True`, vertical lines are plotted from 0 to the xcorr value using *Axes.vlines*. Additionally, a horizontal line is plotted at y=0 using *Axes.axhline*.

If `False`, markers are plotted at the xcorr values using *Axes.plot*.

maxlags

[int, default: 10] Number of lags to show. If None, will return all `2 * len(x) - 1` lags.

Returns

lags

[array (length `2*maxlags+1`)] The lag vector.

c

[array (length `2*maxlags+1`)] The auto correlation vector.

line

[*LineCollection* or *Line2D*] *Artist* added to the axes of the correlation:

- *LineCollection* if *usevlines* is True.
- *Line2D* if *usevlines* is False.

b

[*Line2D* or None] Horizontal line at 0 if *usevlines* is True None *usevlines* is False.

Other Parameters

linestyle

[*Line2D* property, optional] The linestyle for plotting the data points. Only used if *usevlines* is `False`.

marker

[str, default: 'o'] The marker for plotting the data points. Only used if *usevlines* is `False`.

****kwargs**

Additional parameters are passed to *Axes.vlines* and *Axes.axhline* if *usevlines* is `True`; otherwise they are passed to *Axes.plot*.

Notes

The cross correlation is performed with `numpy.correlate` with `mode = "full"`.

Note: In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, the following arguments can also be string `s`, which is interpreted as `data[s]` (unless this raises an exception): *x*, *y*.

Objects passed as **data** must support item access (`data[s]`) and membership test (`s in data`).

Examples using `matplotlib.axes.Axes.xcorr`

- sphx_glr_gallery_lines_bars_and_markers_xcorr_acorr_demo.py

Statistics

Axes.boxplot	Make a box and whisker plot.
Axes.violinplot	Make a violin plot.
Axes.violin	Drawing function for violin plots.
Axes.bxp	Drawing function for box and whisker plots.

matplotlib.axes.Axes.boxplot

`Axes.boxplot`(*self*, *x*, *notch=None*, *sym=None*, *vert=None*, *whis=None*, *positions=None*, *widths=None*, *patch_artist=None*, *bootstrap=None*, *usermedians=None*, *conf_intervals=None*, *meanline=None*, *showmeans=None*, *showcaps=None*, *showbox=None*, *showfliers=None*, *boxprops=None*, *labels=None*, *flierprops=None*, *medianprops=None*, *meanprops=None*, *capprops=None*, *whiskerprops=None*, *manage_ticks=True*, *autorange=False*, *zorder=None*, *, *data=None*)

Make a box and whisker plot.

Make a box and whisker plot for each column of *x* or each vector in sequence *x*. The box extends from the lower to upper quartile values of the data, with a line at the median. The whiskers extend from the box to show the range of the data. Flier points are those past the end of the whiskers.

Parameters

x

[Array or a sequence of vectors.] The input data.

notch

[bool, default: False] Whether to draw a noteched box plot (`True`), or a rectangular box plot (`False`). The notches represent the confidence interval (CI) around the median. The documentation for *bootstrap* describes how the locations of the notches are computed.

Note: In cases where the values of the CI are less than the lower quartile or greater than the upper quartile, the notches will extend beyond the box, giving it a distinctive "flipped" appearance. This is expected behavior and consistent with other statistical visualization packages.

---

sym

[str, optional] The default symbol for flier points. An empty string ('') hides the fliers. If `None`, then the fliers default to 'b+'. More control is provided by the *flierprops* parameter.

vert

[bool, default: True] If `True`, draws vertical boxes. If `False`, draw horizontal boxes.

whis

[float or (float, float), default: 1.5] The position of the whiskers.

If a float, the lower whisker is at the lowest datum above `Q1 - whis*(Q3-Q1)`, and the upper whisker at the highest datum below `Q3 + whis*(Q3-Q1)`, where Q1 and Q3 are the first and third quartiles. The default value of `whis = 1.5` corresponds to Tukey's original definition of boxplots.

If a pair of floats, they indicate the percentiles at which to draw the whiskers (e.g., (5, 95)). In particular, setting this to (0, 100) results in whiskers covering the whole range of the data. "range" is a deprecated synonym for (0, 100).

In the edge case where `Q1 == Q3`, *whis* is automatically set to (0, 100) (cover the whole range of the data) if *autorange* is True.

Beyond the whiskers, data are considered outliers and are plotted as individual points.

bootstrap

[int, optional] Specifies whether to bootstrap the confidence intervals around the median for notched boxplots. If *bootstrap* is

None, no bootstrapping is performed, and notches are calculated using a Gaussian-based asymptotic approximation (see McGill, R., Tukey, J.W., and Larsen, W.A., 1978, and Kendall and Stuart, 1967). Otherwise, bootstrap specifies the number of times to bootstrap the median to determine its 95% confidence intervals. Values between 1000 and 10000 are recommended.

usermedians

[array-like, optional] A 1D array-like of length `len(x)`. Each entry that is not `None` forces the value of the median for the corresponding dataset. For entries that are `None`, the medians are computed by Matplotlib as normal.

conf_intervals

[array-like, optional] A 2D array-like of shape `(len(x), 2)`. Each entry that is not None forces the location of the corresponding notch (which is only drawn if *notch* is `True`). For entries that are `None`, the notches are computed by the method specified by the other parameters (e.g., *bootstrap*).

positions

[array-like, optional] Sets the positions of the boxes. The ticks and limits are automatically set to match the positions. Defaults to `range(1, N+1)` where N is the number of boxes to be drawn.

widths

[float or array-like] Sets the width of each box either with a scalar or a sequence. The default is 0.5, or `0.15*(distance between extreme positions)`, if that is smaller.

patch_artist

[bool, default: False] If `False` produces boxes with the Line2D artist. Otherwise, boxes and drawn with Patch artists.

labels

[sequence, optional] Labels for each dataset (one per dataset).

manage_ticks

[bool, default: True] If True, the tick locations and labels will be adjusted to match the boxplot positions.

autorange

[bool, default: False] When `True` and the data are distributed such that the 25th and 75th percentiles are equal, *whis* is set to (0, 100) such that the whisker ends are at the minimum and maximum of the data.

meanline

[bool, default: False] If `True` (and *showmeans* is `True`), will try to render the mean as a line spanning the full width of the box according to *meanprops* (see below). Not recommended if *shownotches* is also True. Otherwise, means will be shown as points.

zorder

[float, default: `Line2D.zorder = 2`] Sets the zorder of the boxplot.

Returns

dict

A dictionary mapping each component of the boxplot to a list of the *Line2D* instances created. That dictionary has the following keys (assuming vertical boxplots):

- `boxes`: the main body of the boxplot showing the quartiles and the median's confidence intervals if enabled.
- `medians`: horizontal lines at the median of each box.
- `whiskers`: the vertical lines extending to the most extreme, non-outlier data points.
- `caps`: the horizontal lines at the ends of the whiskers.
- `fliers`: points representing data that extend beyond the whiskers (fliers).
- `means`: points or lines representing the means.

Other Parameters

showcaps

[bool, default: True] Show the caps on the ends of whiskers.

showbox

[bool, default: True] Show the central box.

showfliers

[bool, default: True] Show the outliers beyond the caps.

showmeans

[bool, default: False] Show the arithmetic means.

capprops

[dict, default: None] The style of the caps.

boxprops

[dict, default: None] The style of the box.

whiskerprops

[dict, default: None] The style of the whiskers.

flierprops

[dict, default: None] The style of the fliers.

medianprops

[dict, default: None] The style of the median.

meanprops

[dict, default: None] The style of the mean.

Notes

Note: In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, every other argument can also be string `s`, which is interpreted as `data[s]` (unless this raises an exception).

Objects passed as **data** must support item access (`data[s]`) and membership test (`s in data`).

Examples using `matplotlib.axes.Axes.boxplot`

- sphx_glr_gallery_statistics_boxplot_color.py
- sphx_glr_gallery_statistics_boxplot_demo.py
- sphx_glr_gallery_pyplots_boxplot_demo_pyplot.py

matplotlib.axes.Axes.violinplot

`Axes.violinplot`(*self*, *dataset*, *positions=None*, *vert=True*, *widths=0.5*, *showmeans=False*, *showextrema=True*, *showmedians=False*, *quantiles=None*, *points=100*, *bw_method=None*, ***, *data=None*)

Make a violin plot.

Make a violin plot for each column of *dataset* or each vector in sequence *dataset*. Each filled area extends to represent the entire data range, with optional lines at the mean, the median, the minimum, the maximum, and user-specified quantiles.

Parameters

dataset

[Array or a sequence of vectors.] The input data.

positions

[array-like, default: [1, 2, ..., n]] Sets the positions of the violins. The ticks and limits are automatically set to match the positions.

vert

[bool, default: True.] If true, creates a vertical violin plot. Otherwise, creates a horizontal violin plot.

widths

[array-like, default: 0.5] Either a scalar or a vector that sets the maximal width of each violin. The default is 0.5, which uses about half of the available horizontal space.

showmeans

[bool, default: False] If `True`, will toggle rendering of the means.

showextrema

[bool, default: True] If `True`, will toggle rendering of the extrema.

showmedians

[bool, default: False] If `True`, will toggle rendering of the medians.

quantiles

[array-like, default: None] If not None, set a list of floats in interval [0, 1] for each violin, which stands for the quantiles that will be rendered for that violin.

points

[int, default: 100] Defines the number of points to evaluate each of the gaussian kernel density estimations at.

bw_method

[str, scalar or callable, optional] The method used to calculate the estimator bandwidth. This can be 'scott', 'silverman', a scalar constant or a callable. If a scalar, this will be used directly as `kde.factor`. If a callable, it should take a `GaussianKDE` instance as its only parameter and return a scalar. If None (default), 'scott' is used.

Returns

dict

A dictionary mapping each component of the violinplot to a list of the corresponding collection instances created. The dictionary has the following keys:

- `bodies`: A list of the *PolyCollection* instances containing the filled area of each violin.
- `cmeans`: A *LineCollection* instance that marks the mean values of each of the violin's distribution.
- `cmins`: A *LineCollection* instance that marks the bottom of each violin's distribution.
- `cmaxes`: A *LineCollection* instance that marks the top of each violin's distribution.
- `cbars`: A *LineCollection* instance that marks the centers of each violin's distribution.
- `cmedians`: A *LineCollection* instance that marks the median values of each of the violin's distribution.
- `cquantiles`: A *LineCollection* instance created to identify the quantile values of each of the violin's distribution.

Notes

Note: In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, the following arguments can also be string `s`, which is interpreted as `data[s]` (unless this raises an exception): *dataset*.

Objects passed as **data** must support item access (`data[s]`) and membership test (`s in data`).

Examples using `matplotlib.axes.Axes.violinplot`

- sphx_glr_gallery_statistics_customized_violin.py

matplotlib.axes.Axes.violin

Axes.violin(*self*, *vpstats*, *positions=None*, *vert=True*, *widths=0.5*, *showmeans=False*, *showextrema=True*, *showmedians=False*)

Drawing function for violin plots.

Draw a violin plot for each column of *vpstats*. Each filled area extends to represent the entire data range, with optional lines at the mean, the median, the minimum, the maximum, and the quantiles values.

Parameters

vpstats

[list of dicts] A list of dictionaries containing stats for each violin plot. Required keys are:

- `coords`: A list of scalars containing the coordinates that the violin's kernel density estimate were evaluated at.
- `vals`: A list of scalars containing the values of the kernel density estimate at each of the coordinates given in *coords*.
- `mean`: The mean value for this violin's dataset.
- `median`: The median value for this violin's dataset.
- `min`: The minimum value for this violin's dataset.
- `max`: The maximum value for this violin's dataset.

Optional keys are:

- `quantiles`: A list of scalars containing the quantile values for this violin's dataset.

positions

[array-like, default: [1, 2, ..., n]] Sets the positions of the violins. The ticks and limits are automatically set to match the positions.

vert

[bool, default: True.] If true, plots the violins vertically. Otherwise, plots the violins horizontally.

widths

[array-like, default: 0.5] Either a scalar or a vector that sets the maximal width of each violin. The default is 0.5, which uses about half of the available horizontal space.

showmeans

[bool, default: False] If true, will toggle rendering of the means.

showextrema

[bool, default: True] If true, will toggle rendering of the extrema.

showmedians

[bool, default: False] If true, will toggle rendering of the medians.

Returns

dict

A dictionary mapping each component of the violinplot to a list of the corresponding collection instances created. The dictionary has the following keys:

- `bodies`: A list of the *PolyCollection* instances containing the filled area of each violin.
- `cmeans`: A *LineCollection* instance that marks the mean values of each of the violin's distribution.
- `cmins`: A *LineCollection* instance that marks the bottom of each violin's distribution.
- `cmaxes`: A *LineCollection* instance that marks the top of each violin's distribution.
- `cbars`: A *LineCollection* instance that marks the centers of each violin's distribution.
- `cmedians`: A *LineCollection* instance that marks the median values of each of the violin's distribution.
- `cquantiles`: A *LineCollection* instance created to identify the quantiles values of each of the violin's distribution.

Examples using `matplotlib.axes.Axes.violin`

matplotlib.axes.Axes.bxp

`Axes.bxp`(*self*, *bxpstats*, *positions=None*, *widths=None*, *vert=True*, *patch_artist=False*, *shownotches=False*, *showmeans=False*, *showcaps=True*, *showbox=True*, *showfliers=True*, *boxprops=None*, *whiskerprops=None*, *flierprops=None*, *medianprops=None*, *capprops=None*, *meanprops=None*, *meanline=False*, *manage_ticks=True*, *zorder=None*)

Drawing function for box and whisker plots.

Make a box and whisker plot for each column of *x* or each vector in sequence *x*. The box extends from the lower to upper quartile values of the data, with a line

at the median. The whiskers extend from the box to show the range of the data. Flier points are those past the end of the whiskers.

Parameters

bxpstats

[list of dicts] A list of dictionaries containing stats for each boxplot. Required keys are:

- `med`: The median (scalar float).
- `q1`: The first quartile (25th percentile) (scalar float).
- `q3`: The third quartile (75th percentile) (scalar float).
- `whislo`: Lower bound of the lower whisker (scalar float).
- `whishi`: Upper bound of the upper whisker (scalar float).

Optional keys are:

- `mean`: The mean (scalar float). Needed if `showmeans=True`.
- `fliers`: Data beyond the whiskers (sequence of floats). Needed if `showfliers=True`.
- `cilo` & `cihi`: Lower and upper confidence intervals about the median. Needed if `shownotches=True`.
- `label`: Name of the dataset (string). If available, this will be used a tick label for the boxplot

positions

[array-like, default: [1, 2, ..., n]] Sets the positions of the boxes. The ticks and limits are automatically set to match the positions.

widths

[array-like, default: None] Either a scalar or a vector and sets the width of each box. The default is `0.15*(distance between extreme positions)`, clipped to no less than 0.15 and no more than 0.5.

vert

[bool, default: True] If `True` (default), makes the boxes vertical. If `False`, makes horizontal boxes.

patch_artist

[bool, default: False] If `False` produces boxes with the *Line2D* artist. If `True` produces boxes with the *Patch* artist.

shownotches

[bool, default: False] If `False` (default), produces a rectangular box plot. If `True`, will produce a notched box plot

showmeans

[bool, default: False] If `True`, will toggle on the rendering of the means

showcaps

[bool, default: True] If `True`, will toggle on the rendering of the caps

showbox

[bool, default: True] If `True`, will toggle on the rendering of the box

showfliers

[bool, default: True] If `True`, will toggle on the rendering of the fliers

boxprops

[dict or None (default)] If provided, will set the plotting style of the boxes

whiskerprops

[dict or None (default)] If provided, will set the plotting style of the whiskers

capprops

[dict or None (default)] If provided, will set the plotting style of the caps

flierprops

[dict or None (default)] If provided will set the plotting style of the fliers

medianprops

[dict or None (default)] If provided, will set the plotting style of the medians

meanprops

[dict or None (default)] If provided, will set the plotting style of the means

meanline

[bool, default: False] If `True` (and *showmeans* is `True`), will try to render the mean as a line spanning the full width of the box according to *meanprops*. Not recommended if *shownotches* is also True. Otherwise, means will be shown as points.

manage_ticks

[bool, default: True] If True, the tick locations and labels will be adjusted to match the boxplot positions.

zorder

[float, default: `Line2D.zorder = 2`] The zorder of the resulting boxplot.

Returns

dict

A dictionary mapping each component of the boxplot to a list of the *Line2D* instances created. That dictionary has the following keys (assuming vertical boxplots):

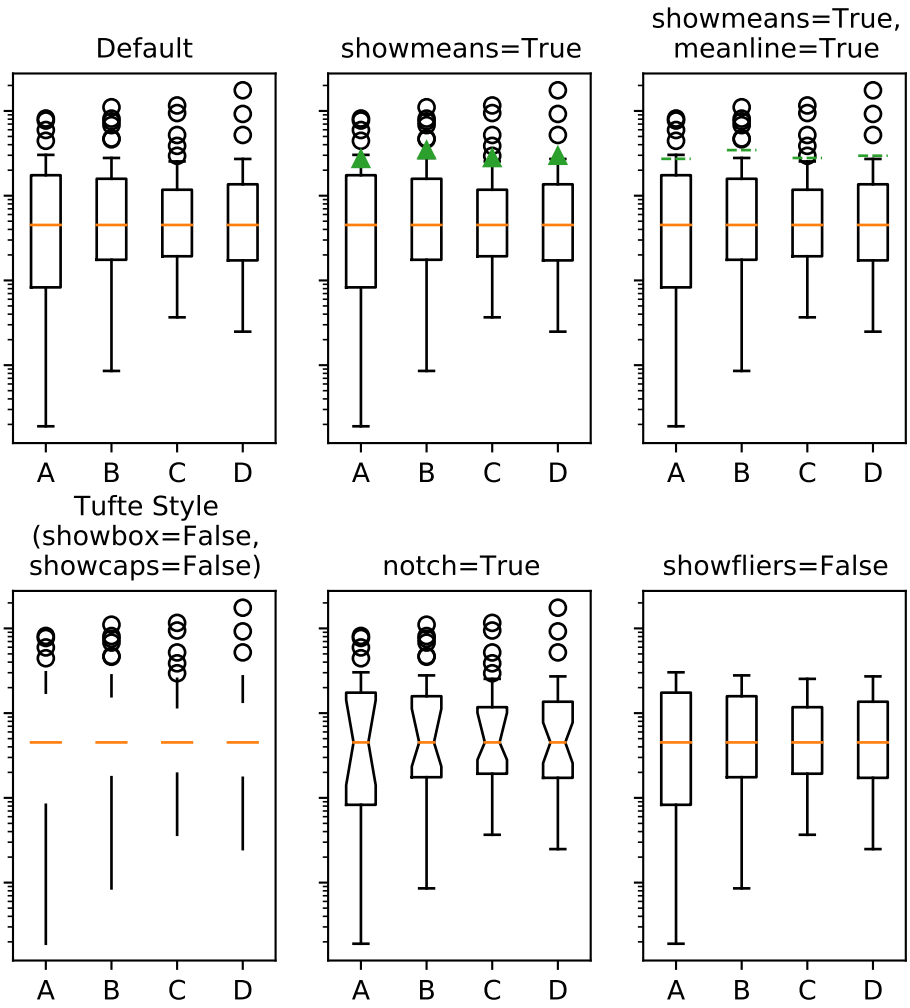
- `boxes`: the main body of the boxplot showing the quartiles and the median's confidence intervals if enabled.
- `medians`: horizontal lines at the median of each box.
- `whiskers`: the vertical lines extending to the most extreme, non-outlier data points.
- `caps`: the horizontal lines at the ends of the whiskers.
- `fliers`: points representing data that extend beyond the whiskers (fliers).
- `means`: points or lines representing the means.

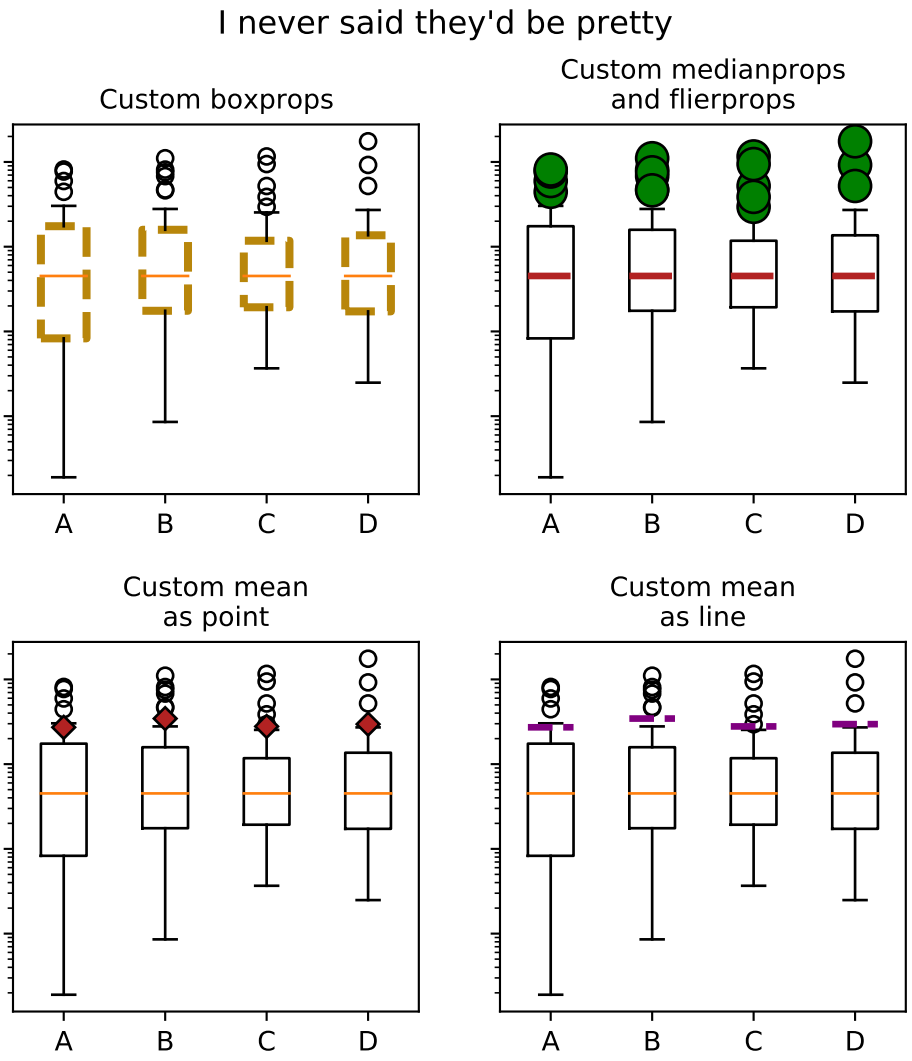
Examples

Examples using `matplotlib.axes.Axes.bxp`

Binned

Axes.hexbin	Make a 2D hexagonal binning plot of points x, y.
Axes.hist	Plot a histogram.
Axes.hist2d	Make a 2D histogram plot.


Default
showmeans=True
showmeans=True,
meanline=True
Tufte Style
(showbox=False,
showcaps=False)
notch=True
showfliers=False
A
B
C
D


I never said they'd be pretty
Custom boxprops
Custom medianprops
and flierprops
Custom mean
as point
Custom mean
as line
A
B
C
D

matplotlib.axes.Axes.hexbin

Axes.hexbin(*self*, *x*, *y*, *C=None*, *gridsize=100*, *bins=None*, *xscale='linear'*, *yscale='linear'*, *extent=None*, *cmap=None*, *norm=None*, *vmin=None*, *vmax=None*, *alpha=None*, *linewidths=None*, *edgecolors='face'*, *reduce_C_function=<function mean at 0x7f5439899310>*, *mincnt=None*, *marginals=False*, ***, *data=None*, ***kwargs*)

Make a 2D hexagonal binning plot of points *x*, *y*.

If *C* is *None*, the value of the hexagon is determined by the number of points in the hexagon. Otherwise, *C* specifies values at the coordinate (x[i], y[i]). For each hexagon, these values are reduced using *reduce_C_function*.

Parameters

x, y

[array-like] The data positions. *x* and *y* must be of the same length.

C

[array-like, optional] If given, these values are accumulated in the bins. Otherwise, every point has a value of 1. Must be of the same length as *x* and *y*.

gridsize

[int or (int, int), default: 100] If a single int, the number of hexagons in the *x*-direction. The number of hexagons in the *y*-direction is chosen such that the hexagons are approximately regular.

Alternatively, if a tuple (*nx*, *ny*), the number of hexagons in the *x*-direction and the *y*-direction.

bins

['log' or int or sequence, default: None] Discretization of the hexagon values.

- If *None*, no binning is applied; the color of each hexagon directly corresponds to its count value.
- If 'log', use a logarithmic scale for the color map. Internally, $log_{10}(i+1)$ is used to determine the hexagon color. This is equivalent to `norm=LogNorm()`.
- If an integer, divide the counts in the specified number of bins, and color the hexagons accordingly.
- If a sequence of values, the values of the lower bound of the bins to be used.

xscale

[{'linear', 'log'}, default: 'linear'] Use a linear or log10 scale on the horizontal axis.

yscale

[{'linear', 'log'}, default: 'linear'] Use a linear or log10 scale on the vertical axis.

mincnt

[int > 0, default: *None*] If not *None*, only display cells with more than *mincnt* number of points in the cell.

marginals

[bool, default: *False*] If marginals is *True*, plot the marginal density as colormapped rectangles along the bottom of the x-axis and left of the y-axis.

extent

[float, default: *None*] The limits of the bins. The default assigns the limits based on *gridsize*, *x*, *y*, *xscale* and *yscale*.

If *xscale* or *yscale* is set to 'log', the limits are expected to be the exponent for a power of 10. E.g. for x-limits of 1 and 50 in 'linear' scale and y-limits of 10 and 1000 in 'log' scale, enter (1, 50, 1, 3).

Order of scalars is (left, right, bottom, top).

Returns

PolyCollection

A *PolyCollection* defining the hexagonal bins.

- *PolyCollection.get_offsets* contains a Mx2 array containing the x, y positions of the M hexagon centers.
- *PolyCollection.get_array* contains the values of the M hexagons.

If *marginals* is *True*, horizontal bar and vertical bar (both PolyCollections) will be attached to the return collection as attributes *hbar* and *vbar*.

Other Parameters

cmap

[str or *Colormap*, default: `rcParams["image.cmap"]` (default: `'viridis'`)] The Colormap instance or registered colormap name used to map the bin values to colors.

norm

[*Normalize*, optional] The Normalize instance scales the bin values to the canonical colormap range [0, 1] for mapping to colors. By default, the data range is mapped to the colorbar range using linear scaling.

vmin, vmax

[float, default: None] The colorbar range. If *None*, suitable min/max values are automatically chosen by the *Normalize* instance (defaults to the respective min/max values of the bins in case of the default linear scaling). It is deprecated to use *vmin*/*vmax* when *norm* is given.

alpha

[float between 0 and 1, optional] The alpha blending value, between 0 (transparent) and 1 (opaque).

linewidths

[float, default: *None*] If *None*, defaults to 1.0.

edgecolors

[{'face', 'none', *None*} or color, default: 'face'] The color of the hexagon edges. Possible values are:

- 'face': Draw the edges in the same color as the fill color.
- 'none': No edges are drawn. This can sometimes lead to unsightly unpainted pixels between the hexagons.
- *None*: Draw outlines in the default color.
- An explicit color.

reduce_C_function

[callable, default: `numpy.mean`] The function to aggregate *C* within the bins. It is ignored if *C* is not given. This must have the signature:

```
def reduce_C_function(C: array) -> float
```

Commonly used functions are:

- `numpy.mean`: average of the points
- `numpy.sum`: integral of the point values
- `numpy.max`: value taken from the largest point

****kwargs**

[*PolyCollection* properties] All other keyword arguments are passed on to *PolyCollection*:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a
alpha	float or None
animated	bool
antialiased or aa or antialiaseds	bool or list of bools
array	ndarray
capstyle	{'butt', 'round', 'projecting'}
clim	(vmin: float, vmax: float)
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
cmap	*Colormap* or str or None
color	color or list of rgba tuples
contains	unknown
edgecolor or ec or edgecolors	color or list of colors or 'face'
facecolor or facecolors or fc	color or list of colors
figure	*Figure*
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or dashes or linestyles or ls	str or tuple or list thereof
linewidth or linewidths or lw	float or list of floats
norm	*Normalize* or None
offset_position	unknown
offsets	array-like (N, 2) or (2,)
path_effects	*AbstractPathEffect*
picker	None or bool or callable
pickradius	unknown
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
urls	list of str or None
visible	bool
zorder	float

Notes

Note: In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, the following arguments can also be string `s`, which is interpreted as `data[s]` (unless this raises an exception): *x*, *y*.

Objects passed as **data** must support item access (`data[s]`) and membership test (`s in data`).

Examples using `matplotlib.axes.Axes.hexbin`

- sphx_glr_gallery_statistics_hexbin_demo.py

matplotlib.axes.Axes.hist

`Axes.hist`(*self*, *x*, *bins=None*, *range=None*, *density=False*, *weights=None*, *cumulative=False*, *bottom=None*, *histtype='bar'*, *align='mid'*, *orientation='vertical'*, *rwidth=None*, *log=False*, *color=None*, *label=None*, *stacked=False*, *, *data=None*, **kwargs*)

Plot a histogram.

Compute and draw the histogram of *x*. The return value is a tuple (*n*, *bins*, *patches*) or ([*n0*, *n1*, ...], *bins*, [*patches0*, *patches1*, ...]) if the input contains multiple data. See the documentation of the *weights* parameter to draw a histogram of already-binned data.

Multiple data can be provided via *x* as a list of datasets of potentially different length ([*x0*, *x1*, ...]), or as a 2-D ndarray in which each column is a dataset. Note that the ndarray form is transposed relative to the list form.

Masked arrays are not supported.

The *bins*, *range*, *weights*, and *density* parameters behave as in `numpy.histogram`.

Parameters

x

[(n,) array or sequence of (n,) arrays] Input values, this takes either a single array or a sequence of arrays which are not required to be of the same length.

bins

[int or sequence or str, default: `rcParams["hist.bins"]` (default: `10`)] If *bins* is an integer, it defines the number of equal-width bins in the range.

If *bins* is a sequence, it defines the bin edges, including the left edge of the first bin and the right edge of the last bin; in this case, bins may be unequally spaced. All but the last (righthand-most) bin is half-open. In other words, if *bins* is:

```
[1, 2, 3, 4]
```

then the first bin is `[1, 2)` (including 1, but excluding 2) and the second `[2, 3)`. The last bin, however, is `[3, 4]`, which *includes* 4.

If *bins* is a string, it is one of the binning strategies supported by `numpy.histogram_bin_edges`: 'auto', 'fd', 'doane', 'scott', 'stone', 'rice', 'sturges', or 'sqrt'.

range

[tuple or None, default: None] The lower and upper range of the bins. Lower and upper outliers are ignored. If not provided, *range* is `(x.min(), x.max())`. Range has no effect if *bins* is a sequence.

If *bins* is a sequence or *range* is specified, autoscaling is based on the specified bin range instead of the range of x.

density

[bool, default: False] If `True`, draw and return a probability density: each bin will display the bin's raw count divided by the total number of counts *and the bin width* (`density = counts / (sum(counts) * np.diff(bins))`), so that the area under the histogram integrates to 1 (`np.sum(density * np.diff(bins)) == 1`).

If *stacked* is also `True`, the sum of the histograms is normalized to 1.

weights

[(n,) array-like or None, default: None] An array of weights, of the same shape as *x*. Each value in *x* only contributes its associated weight towards the bin count (instead of 1). If *density* is `True`, the weights are normalized, so that the integral of the density over the range remains 1.

This parameter can be used to draw a histogram of data that has already been binned, e.g. using `numpy.histogram` (by treating each bin as a single point with a weight equal to its count)

```
counts, bins = np.histogram(data)
plt.hist(bins[:-1], bins, weights=counts)
```

(or you may alternatively use *bar()*).

cumulative

[bool or -1, default: False] If `True`, then a histogram is computed where each bin gives the counts in that bin plus all bins for smaller values. The last bin gives the total number of datapoints.

If *density* is also `True` then the histogram is normalized such that the last bin equals 1.

If *cumulative* is a number less than 0 (e.g., -1), the direction of accumulation is reversed. In this case, if *density* is also `True`, then the histogram is normalized such that the first bin equals 1.

bottom

[array-like, scalar, or None, default: None] Location of the bottom of each bin, ie. bins are drawn from `bottom` to `bottom + hist(x, bins)` If a scalar, the bottom of each bin is shifted by the same amount. If an array, each bin is shifted independently and the length of bottom must match the number of bins. If None, defaults to 0.

histtype

[{'bar', 'barstacked', 'step', 'stepfilled'}, default: 'bar'] The type of histogram to draw.

- 'bar' is a traditional bar-type histogram. If multiple data are given the bars are arranged side by side.
- 'barstacked' is a bar-type histogram where multiple data are stacked on top of each other.
- 'step' generates a lineplot that is by default unfilled.
- 'stepfilled' generates a lineplot that is by default filled.

align

[{'left', 'mid', 'right'}, default: 'mid'] The horizontal alignment of the histogram bars.

- 'left': bars are centered on the left bin edges.
- 'mid': bars are centered between the bin edges.
- 'right': bars are centered on the right bin edges.

orientation

[{'vertical', 'horizontal'}, default: 'vertical'] If 'horizontal', *barh* will be used for bar-type histograms and the *bottom* kwarg will be the left edges.

rwidth

[float or None, default: None] The relative width of the bars as a fraction of the bin width. If `None`, automatically compute the width.

Ignored if *histtype* is 'step' or 'stepfilled'.

log

[bool, default: False] If `True`, the histogram axis will be set to a log scale. If *log* is `True` and *x* is a 1D array, empty bins will be filtered out and only the non-empty `(n, bins, patches)` will be returned.

color

[color or array-like of colors or None, default: None] Color or sequence of colors, one per dataset. Default (`None`) uses the standard line color sequence.

label

[str or None, default: None] String, or sequence of strings to match multiple datasets. Bar charts yield multiple patches per dataset, but only the first gets the label, so that *legend* will work as expected.

stacked

[bool, default: False] If `True`, multiple data are stacked on top of each other If `False` multiple data are arranged side by side if histtype is 'bar' or on top of each other if histtype is 'step'

Returns

n

[array or list of arrays] The values of the histogram bins. See *density* and *weights* for a description of the possible semantics. If input *x* is an array, then this is an array of length *nbins*. If input is a sequence of arrays `[data1, data2, ...]`, then this is a list of arrays with the values of the histograms for each of the arrays in the same order. The dtype of the array *n* (or of its element arrays) will always be float even if no weighting or normalization is used.

bins

[array] The edges of the bins. Length nbins + 1 (nbins left edges and right edge of last bin). Always a single array even when multiple data sets are passed in.

patches

[*BarContainer* or list of a single *Polygon* or list of such objects] Container of individual artists used to create the histogram or list of such containers if there are multiple input datasets.

Other Parameters

****kwargs**

Patch properties

See also:

hist2d

2D histograms

Notes

For large numbers of bins (>1000), 'step' and 'stepfilled' can be significantly faster than 'bar' and 'barstacked'.

Note: In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, the following arguments can also be string `s`, which is interpreted as `data[s]` (unless this raises an exception): *x*, *weights*.

Objects passed as **data** must support item access (`data[s]`) and membership test (`s in data`).

Examples using `matplotlib.axes.Axes.hist`

- sphx_glr_gallery_lines_bars_and_markers_scatter_hist.py
- sphx_glr_gallery_subplots_axes_and_figures_axes_demo.py
- sphx_glr_gallery_statistics_histogram_cumulative.py
- sphx_glr_gallery_statistics_histogram_features.py
- sphx_glr_gallery_statistics_histogram_multihist.py
- sphx_glr_gallery_pyplots_fig_axes_labels_simple.py
- sphx_glr_gallery_style_sheets_bmh.py
- sphx_glr_gallery_axes_grid1_scatter_hist_locatable_axes.py
- sphx_glr_gallery_frontpage_histogram.py
- sphx_glr_gallery_misc_histogram_path.py
- sphx_glr_gallery_recipes_placing_text_boxes.py
- sphx_glr_gallery_specialty_plots_mri_with_eeg.py

- *Artist tutorial*
- *Path Tutorial*
- *Transformations Tutorial*

matplotlib.axes.Axes.hist2d

`Axes.hist2d`(*self*, *x*, *y*, *bins=10*, *range=None*, *density=False*, *weights=None*, *cmin=None*, *cmax=None*, *, *data=None*, **kwargs*)

Make a 2D histogram plot.

Parameters

x, y

[array-like, shape (n,)] Input values

bins

[None or int or [int, int] or array-like or [array, array]] The bin specification:

- If int, the number of bins for the two dimensions (nx=ny=bins).
- If `[int, int]`, the number of bins in each dimension (nx, ny = bins).
- If array-like, the bin edges for the two dimensions (x_edges=y_edges=bins).
- If `[array, array]`, the bin edges in each dimension (x_edges, y_edges = bins).

The default value is 10.

range

[array-like shape(2, 2), optional] The leftmost and rightmost edges of the bins along each dimension (if not specified explicitly in the bins parameters): `[[xmin, xmax], [ymin, ymax]]`. All values outside of this range will be considered outliers and not tallied in the histogram.

density

[bool, default: False] Normalize histogram. See the documentation for the *density* parameter of *hist* for more details.

weights

[array-like, shape (n,), optional] An array of values w_i weighing each sample (x_i, y_i).

cmin, cmax

[float, default: None] All bins that has count less than *cmin* or more than *cmax* will not be displayed (set to NaN before passing to imshow) and these count values in the return value count histogram will also be set to nan upon return.

Returns

h

[2D array] The bi-dimensional histogram of samples x and y. Values in x are histogrammed along the first dimension and values in y are histogrammed along the second dimension.

xedges

[1D array] The bin edges along the x axis.

yedges

[1D array] The bin edges along the y axis.

image

[*QuadMesh*]

Other Parameters

cmap

[Colormap or str, optional] A *colors.Colormap* instance. If not set, use rc settings.

norm

[Normalize, optional] A *colors.Normalize* instance is used to scale luminance data to `[0, 1]`. If not set, defaults to *colors.Normalize()*.

vmin/vmax

[None or scalar, optional] Arguments passed to the *Normalize* instance.

alpha

[`0 <= scalar <= 1` or `None`, optional] The alpha blending value.

****kwargs**

Additional parameters are passed along to the *pcolormesh* method and *QuadMesh* constructor.

See also:

hist

1D histogram plotting

Notes

- Currently `hist2d` calculates its own axis limits, and any limits previously set are ignored.
- Rendering the histogram with a logarithmic color scale is accomplished by passing a *colors.LogNorm* instance to the *norm* keyword argument. Likewise, power-law normalization (similar in effect to gamma correction) can be accomplished with *colors.PowerNorm*.

Note: In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, the following arguments can also be string `s`, which is interpreted as `data[s]` (unless this raises an exception): *x*, *y*, *weights*.

Objects passed as **data** must support item access (`data[s]`) and membership test (`s in data`).

Examples using `matplotlib.axes.Axes.hist2d`

- sphx_glr_gallery_statistics_hist.py
- sphx_glr_gallery_scales_power_norm.py

Contours

Axes.clabel	Label a contour plot.
Axes.contour	Plot contours.
Axes.contourf	Plot contours.

matplotlib.axes.Axes.clabel

Axes.clabel(*self*, *CS*, *levels=None*, ***kwargs*)

Label a contour plot.

Adds labels to line contours in given *ContourSet*.

Parameters

CS

[*ContourSet* instance] Line contours to label.

levels

[array-like, optional] A list of level values, that should be labeled. The list must be a subset of `CS.levels`. If not given, all levels are labeled.

****kwargs**

All other parameters are documented in *clabel*.

Examples using matplotlib.axes.Axes.clabel

- sphx_glr_gallery_images_contours_and_fields_contour_demo.py
- sphx_glr_gallery_images_contours_and_fields_contour_label_demo.py
- sphx_glr_gallery_images_contours_and_fields_contourf_demo.py
- sphx_glr_gallery_misc_patheffect_demo.py
- sphx_glr_gallery_mplot3d_contour3d.py
- sphx_glr_gallery_mplot3d_contour3d_2.py
- sphx_glr_gallery_mplot3d_contourf3d.py

matplotlib.axes.Axes.contour

Axes.contour(*self*, **args*, *data=None*, ***kwargs*)

Plot contours.

Call signature:

```
contour([X, Y,] Z, [levels], **kwargs)
```

contour and *contourf* draw contour lines and filled contours, respectively. Except as noted, function signatures and return values are the same for both versions.

Parameters

X, Y

[array-like, optional] The coordinates of the values in *Z*.

X and *Y* must both be 2-D with the same shape as *Z* (e.g. created via `numpy.meshgrid`), or they must both be 1-D such that `len(X) == M` is the number of columns in *Z* and `len(Y) == N` is the number of rows in *Z*.

If not given, they are assumed to be integer indices, i.e. `X = range(M)`, `Y = range(N)`.

Z

[array-like(N, M)] The height values over which the contour is drawn.

levels

[int or array-like, optional] Determines the number and positions of the contour lines / regions.

If an int *n*, use *MaxNLocator*, which tries to automatically choose no more than *n+1* "nice" contour levels between *vmin* and *vmax*.

If array-like, draw contour lines at the specified levels. The values must be in increasing order.

Returns

QuadContourSet

Other Parameters

corner_mask

[bool, default: `rcParams["contour.corner_mask"]` (default: `True`)] Enable/disable corner masking, which only has an effect if *Z* is a masked array. If `False`, any quad touching a masked point is masked out. If `True`, only the triangular corners of quads nearest those points are always masked out, other triangular corners comprising three unmasked points are contoured as usual.

colors

[color string or sequence of colors, optional] The colors of the levels, i.e. the lines for *contour* and the areas for *contourf*.

The sequence is cycled for the levels in ascending order. If the sequence is shorter than the number of levels, it's repeated.

As a shortcut, single color strings may be used in place of one-element lists, i.e. `'red'` instead of `['red']` to color all levels with the same color. This shortcut does only work for color strings, not for other ways of specifying colors.

By default (value *None*), the colormap specified by *cmap* will be used.

alpha

[float, default: 1] The alpha blending value, between 0 (transparent) and 1 (opaque).

cmap

[str or *Colormap*, default: `rcParams["image.cmap"]` (default: `'viridis'`)] A *Colormap* instance or registered colormap name. The colormap maps the level values to colors.

If both *colors* and *cmap* are given, an error is raised.

norm

[*Normalize*, optional] If a colormap is used, the *Normalize* instance scales the level values to the canonical colormap range [0, 1] for mapping to colors. If not given, the default linear scaling is used.

vmin, vmax

[float, optional] If not *None*, either or both of these values will be supplied to the *Normalize* instance, overriding the default color scaling based on *levels*.

origin

[{*None*, 'upper', 'lower', 'image'}, default: None] Determines the orientation and exact position of *Z* by specifying the position of `Z[0, 0]`. This is only relevant, if *X*, *Y* are not given.

- *None*: `Z[0, 0]` is at X=0, Y=0 in the lower left corner.
- 'lower': `Z[0, 0]` is at X=0.5, Y=0.5 in the lower left corner.
- 'upper': `Z[0, 0]` is at X=N+0.5, Y=0.5 in the upper left corner.
- 'image': Use the value from `rcParams["image.origin"]` (default: `'upper'`).

extent

[(x0, x1, y0, y1), optional] If *origin* is not *None*, then *extent* is interpreted as in *imshow*: it gives the outer pixel boundaries. In this case, the position of Z[0, 0] is the center of the pixel, not a corner. If *origin* is *None*, then (*x0*, *y0*) is the position of Z[0, 0], and (*x1*, *y1*) is the position of Z[-1, -1].

This argument is ignored if *X* and *Y* are specified in the call to contour.

locator

[ticker.Locator subclass, optional] The locator is used to determine the contour levels if they are not given explicitly via *levels*. Defaults to *MaxNLocator*.

extend

[{'neither', 'both', 'min', 'max'}, default: 'neither'] Determines the `contourf`-coloring of values that are outside the *levels* range.

If 'neither', values outside the *levels* range are not colored. If 'min', 'max' or 'both', color the values below, above or below and above the *levels* range.

Values below `min(levels)` and above `max(levels)` are mapped to the under/over values of the *Colormap*. Note that most colormaps do not have dedicated colors for these by default, so that the over and under values are the edge values of the colormap. You may want to set these values explicitly using *Colormap.set_under* and *Colormap.set_over*.

Note: An existing *QuadContourSet* does not get notified if properties of its colormap are changed. Therefore, an explicit call `QuadContourSet.changed()` is needed after modifying the colormap. The explicit call can be left out, if a colorbar is assigned to the *QuadContourSet* because it internally calls `QuadContourSet.changed()`.

Example:

```
x = np.arange(1, 10)
y = x.reshape(-1, 1)
h = x * y

cs = plt.contourf(h, levels=[10, 30, 50],
    colors=['#808080', '#A0A0A0', '#C0C0C0'], extend='both')
cs.cmap.set_over('red')
cs.cmap.set_under('blue')
cs.changed()
```

xunits, yunits

[registered units, optional] Override axis units by specifying an instance of a *matplotlib.units.ConversionInterface*.

antialiased

[bool, optional] Enable antialiasing, overriding the defaults. For filled contours, the default is *True*. For line contours, it is taken from `rcParams["lines.antialiased"]` (default: `True`).

nchunk

[int >= 0, optional] If 0, no subdivision of the domain. Specify a positive integer to divide the domain into subdomains of *nchunk* by *nchunk* quads. Chunking reduces the maximum length of polygons generated by the contouring algorithm which reduces the rendering workload passed on to the backend and also requires slightly less RAM. It can however introduce rendering artifacts at chunk boundaries depending on the backend, the *antialiased* flag and value of *alpha*.

linewidths

[float or array-like, default: `rcParams["contour.linewidth"]` (default: `None`)] *Only applies to* *`contour`*.

The line width of the contour lines.

If a number, all levels will be plotted with this linewidth.

If a sequence, the levels in ascending order will be plotted with the linewidths in the order specified.

If None, this falls back to `rcParams["lines.linewidth"]` (default: `1.5`).

linestyles

[{*None*, 'solid', 'dashed', 'dashdot', 'dotted'}, optional] *Only applies to* *`contour`*.

If *linestyles* is *None*, the default is 'solid' unless the lines are monochrome. In that case, negative contours will take their linestyle from `rcParams["contour.negative_linestyle"]` (default: `'dashed'`) setting.

linestyles can also be an iterable of the above strings specifying a set of linestyles to be used. If this iterable is shorter than the number of contour levels it will be repeated as necessary.

hatches

[List[str], optional] *Only applies to* *`contourf`*.

A list of cross hatch patterns to use on the filled areas. If None, no hatching will be added to the contour. Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Notes

1. *contourf* differs from the MATLAB version in that it does not draw the polygon edges. To draw edges, add line contours with calls to *contour*.
2. *contourf* fills intervals that are closed at the top; that is, for boundaries *z1* and *z2*, the filled region is:

```
z1 < Z <= z2
```

   except for the lowest interval, which is closed on both sides (i.e. it includes the lowest value).

Examples using `matplotlib.axes.Axes.contour`

- sphx_glr_gallery_images_contours_and_fields_contour_corner_mask.py
- sphx_glr_gallery_images_contours_and_fields_contour_demo.py
- sphx_glr_gallery_images_contours_and_fields_contour_image.py
- sphx_glr_gallery_images_contours_and_fields_contour_label_demo.py
- sphx_glr_gallery_images_contours_and_fields_contourf_demo.py
- sphx_glr_gallery_images_contours_and_fields_contourf_hatching.py
- sphx_glr_gallery_images_contours_and_fields_image_transparency_blend.py
- sphx_glr_gallery_images_contours_and_fields_irregulardatagrid.py
- sphx_glr_gallery_misc_patheffect_demo.py
- sphx_glr_gallery_mplot3d_contour3d.py
- sphx_glr_gallery_mplot3d_contour3d_2.py
- sphx_glr_gallery_mplot3d_contour3d_3.py

matplotlib.axes.Axes.contourf

Axes.**contourf**(*self*, **args*, *data=None*, ***kwargs*)

Plot contours.

Call signature:

```
contour([X, Y,] Z, [levels], **kwargs)
```

contour and *contourf* draw contour lines and filled contours, respectively. Except as noted, function signatures and return values are the same for both versions.

Parameters

X, Y

[array-like, optional] The coordinates of the values in *Z*.

X and *Y* must both be 2-D with the same shape as *Z* (e.g. created via `numpy.meshgrid`), or they must both be 1-D such that `len(X) == M` is the number of columns in *Z* and `len(Y) == N` is the number of rows in *Z*.

If not given, they are assumed to be integer indices, i.e. `X = range(M), Y = range(N)`.

Z

[array-like(N, M)] The height values over which the contour is drawn.

levels

[int or array-like, optional] Determines the number and positions of the contour lines / regions.

If an int *n*, use *MaxNLocator*, which tries to automatically choose no more than *n+1* "nice" contour levels between *vmin* and *vmax*.

If array-like, draw contour lines at the specified levels. The values must be in increasing order.

Returns

QuadContourSet

Other Parameters

corner_mask

[bool, default: `rcParams["contour.corner_mask"]` (default: `True`)] Enable/disable corner masking, which only has an effect if *Z* is a masked array. If `False`, any quad touching a masked point is masked out. If `True`, only the triangular corners of quads nearest those points are always masked out, other triangular corners comprising three unmasked points are contoured as usual.

colors

[color string or sequence of colors, optional] The colors of the levels, i.e. the lines for *contour* and the areas for *contourf*.

The sequence is cycled for the levels in ascending order. If the sequence is shorter than the number of levels, it's repeated.

As a shortcut, single color strings may be used in place of one-element lists, i.e. `'red'` instead of `['red']` to color all levels with the same color. This shortcut does only work for color strings, not for other ways of specifying colors.

By default (value *None*), the colormap specified by *cmap* will be used.

alpha

[float, default: 1] The alpha blending value, between 0 (transparent) and 1 (opaque).

cmap

[str or *Colormap*, default: `rcParams["image.cmap"]` (default: `'viridis'`)] A *Colormap* instance or registered colormap name. The colormap maps the level values to colors.

If both *colors* and *cmap* are given, an error is raised.

norm

[*Normalize*, optional] If a colormap is used, the *Normalize* instance scales the level values to the canonical colormap range [0, 1] for mapping to colors. If not given, the default linear scaling is used.

vmin, vmax

[float, optional] If not *None*, either or both of these values will be supplied to the *Normalize* instance, overriding the default color scaling based on *levels*.

origin

[{*None*, 'upper', 'lower', 'image'}, default: None] Determines the orientation and exact position of *Z* by specifying the position of `Z[0, 0]`. This is only relevant, if *X*, *Y* are not given.

- *None*: `Z[0, 0]` is at X=0, Y=0 in the lower left corner.
- 'lower': `Z[0, 0]` is at X=0.5, Y=0.5 in the lower left corner.
- 'upper': `Z[0, 0]` is at X=N+0.5, Y=0.5 in the upper left corner.
- 'image': Use the value from `rcParams["image.origin"]` (default: `'upper'`).

extent

[(x0, x1, y0, y1), optional] If *origin* is not *None*, then *extent* is interpreted as in *imshow*: it gives the outer pixel boundaries. In this case, the position of Z[0, 0] is the center of the pixel, not a corner. If *origin* is *None*, then (*x0*, *y0*) is the position of Z[0, 0], and (*x1*, *y1*) is the position of Z[-1, -1].

This argument is ignored if *X* and *Y* are specified in the call to contour.

locator

[ticker.Locator subclass, optional] The locator is used to determine the contour levels if they are not given explicitly via *levels*. Defaults to *MaxNLocator*.

extend

[{'neither', 'both', 'min', 'max'}, default: 'neither'] Determines the `contourf`-coloring of values that are outside the *levels* range.

If 'neither', values outside the *levels* range are not colored. If 'min', 'max' or 'both', color the values below, above or below and above the *levels* range.

Values below `min(levels)` and above `max(levels)` are mapped to the under/over values of the *Colormap*. Note that most colormaps do not have dedicated colors for these by default, so that the over and under values are the edge values of the colormap. You may want to set these values explicitly using *Colormap.set_under* and *Colormap.set_over*.

Note: An existing *QuadContourSet* does not get notified if properties of its colormap are changed. Therefore, an explicit call `QuadContourSet.changed()` is needed after modifying the colormap. The explicit call can be left out, if a colorbar is assigned to the *QuadContourSet* because it internally calls `QuadContourSet.changed()`.

Example:

```
x = np.arange(1, 10)
y = x.reshape(-1, 1)
h = x * y

cs = plt.contourf(h, levels=[10, 30, 50],
    colors=['#808080', '#A0A0A0', '#C0C0C0'], extend='both')
cs.cmap.set_over('red')
cs.cmap.set_under('blue')
cs.changed()
```

xunits, yunits

[registered units, optional] Override axis units by specifying an instance of a *matplotlib.units.ConversionInterface*.

antialiased

[bool, optional] Enable antialiasing, overriding the defaults. For filled contours, the default is *True*. For line contours, it is taken from `rcParams["lines.antialiased"]` (default: `True`).

nchunk

[int >= 0, optional] If 0, no subdivision of the domain. Specify a positive integer to divide the domain into subdomains of *nchunk* by *nchunk* quads. Chunking reduces the maximum length of polygons generated by the contouring algorithm which reduces the rendering workload passed on to the backend and also requires slightly less RAM. It can however introduce rendering artifacts at chunk boundaries depending on the backend, the *antialiased* flag and value of *alpha*.

linewidths

[float or array-like, default: `rcParams["contour.linewidth"]` (default: `None`)] *Only applies to* *`contour`*.

The line width of the contour lines.

If a number, all levels will be plotted with this linewidth.

If a sequence, the levels in ascending order will be plotted with the linewidths in the order specified.

If None, this falls back to `rcParams["lines.linewidth"]` (default: `1.5`).

linestyles

[{*None*, 'solid', 'dashed', 'dashdot', 'dotted'}, optional] *Only applies to* *`contour`*.

If *linestyles* is *None*, the default is 'solid' unless the lines are monochrome. In that case, negative contours will take their linestyle from `rcParams["contour.negative_linestyle"]` (default: `'dashed'`) setting.

linestyles can also be an iterable of the above strings specifying a set of linestyles to be used. If this iterable is shorter than the number of contour levels it will be repeated as necessary.

hatches

[List[str], optional] *Only applies to* *`contourf`*.

A list of cross hatch patterns to use on the filled areas. If None, no hatching will be added to the contour. Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Notes

1. *contourf* differs from the MATLAB version in that it does not draw the polygon edges. To draw edges, add line contours with calls to *contour*.
2. *contourf* fills intervals that are closed at the top; that is, for boundaries *z1* and *z2*, the filled region is:

```
z1 < Z <= z2
```

   except for the lowest interval, which is closed on both sides (i.e. it includes the lowest value).

Examples using `matplotlib.axes.Axes.contourf`

- sphx_glr_gallery_images_contours_and_fields_contour_corner_mask.py
- sphx_glr_gallery_images_contours_and_fields_contourf_demo.py
- sphx_glr_gallery_images_contours_and_fields_contourf_hatching.py
- sphx_glr_gallery_images_contours_and_fields_contourf_log.py
- sphx_glr_gallery_images_contours_and_fields_irregulardatagrid.py
- sphx_glr_gallery_images_contours_and_fields_pcolormesh_levels.py
- sphx_glr_gallery_images_contours_and_fields_triinterp_demo.py
- sphx_glr_gallery_frontpage_contour.py
- sphx_glr_gallery_mplot3d_contourf3d.py
- sphx_glr_gallery_mplot3d_contourf3d_2.py

Array

Axes.imshow	Display data as an image, i.e., on a 2D regular raster.
Axes.matshow	Plot the values of a 2D matrix or array as color-coded image.
Axes.pcolor	Create a pseudocolor plot with a non-regular rectangular grid.
Axes.pcolorfast	Create a pseudocolor plot with a non-regular rectangular grid.
Axes.pcolormesh	Create a pseudocolor plot with a non-regular rectangular grid.
Axes.spy	Plot the sparsity pattern of a 2D array.

matplotlib.axes.Axes.imshow

Axes.**imshow**(*self*, *X*, *cmap=None*, *norm=None*, *aspect=None*, *interpolation=None*, *alpha=None*, *vmin=None*, *vmax=None*, *origin=None*, *extent=None*, ***, *filternorm=True*, *filterrad=4.0*, *resample=None*, *url=None*, *data=None*, ***kwargs*)

Display data as an image, i.e., on a 2D regular raster.

The input may either be actual RGB(A) data, or 2D scalar data, which will be rendered as a pseudocolor image. For displaying a grayscale image set up the color mapping using the parameters `cmap='gray', vmin=0, vmax=255`.

The number of pixels used to render an image is set by the axes size and the *dpi* of the figure. This can lead to aliasing artifacts when the image is resampled because the displayed image size will usually not match the size of *X* (see /gallery/images_contours_and_fields/image_antialiasing). The resampling can be controlled via the *interpolation* parameter and/or `rcParams["image.interpolation"]` (default: `'antialiased'`).

Parameters

X

[array-like or PIL image] The image data. Supported array shapes are:

- (M, N): an image with scalar data. The values are mapped to colors using normalization and a colormap. See parameters *norm*, *cmap*, *vmin*, *vmax*.
- (M, N, 3): an image with RGB values (0-1 float or 0-255 int).
- (M, N, 4): an image with RGBA values (0-1 float or 0-255 int), i.e. including transparency.

The first two dimensions (M, N) define the rows and columns of the image.

Out-of-range RGB(A) values are clipped.

cmap

[str or *Colormap*, default: `rcParams["image.cmap"]` (default: `'viridis'`)] The Colormap instance or registered colormap name used to map scalar data to colors. This parameter is ignored for RGB(A) data.

norm

[*Normalize*, optional] The *Normalize* instance used to scale scalar data to the [0, 1] range before mapping to colors using *cmap*. By default, a linear scaling mapping the lowest value to 0 and the highest to 1 is used. This parameter is ignored for RGB(A) data.

aspect

[{'equal', 'auto'} or float, default: `rcParams["image.aspect"]` (default: `'equal'`)] The aspect ratio of the axes. This parameter is particularly relevant for images since it determines whether data pixels are square.

This parameter is a shortcut for explicitly calling *Axes.set_aspect*. See there for further details.

- 'equal': Ensures an aspect ratio of 1. Pixels will be square (unless pixel sizes are explicitly made non-square in data coordinates using *extent*).
- 'auto': The axes is kept fixed and the aspect is adjusted so that the data fit in the axes. In general, this will result in non-square pixels.

interpolation

[str, default: `rcParams["image.interpolation"]` (default: `'antialiased'`)] The interpolation method used.

Supported values are 'none', 'antialiased', 'nearest', 'bilinear', 'bicubic', 'spline16', 'spline36', 'hanning', 'hamming', 'hermite', 'kaiser', 'quadric', 'catrom', 'gaussian', 'bessel', 'mitchell', 'sinc', 'lanczos'.

If *interpolation* is 'none', then no interpolation is performed on the Agg, ps, pdf and svg backends. Other backends will fall back to 'nearest'. Note that most SVG renderers perform interpolation at rendering and that the default interpolation method they implement may differ.

If *interpolation* is the default 'antialiased', then 'nearest' interpolation is used if the image is upsampled by more than a factor of three (i.e. the number of display pixels is at least three times the size of the data array). If the upsampling rate is smaller than 3, or the image is downsampled, then 'hanning' interpolation is used to act as an anti-aliasing filter, unless the image happens to be upsampled by exactly a factor of two or one.

See /gallery/images_contours_and_fields/interpolation_methods for an overview of the supported interpolation methods, and /gallery/images_contours_and_fields/image_antialiasing for a discussion of image antialiasing.

Some interpolation methods require an additional radius parameter, which can be set by *filterrad*. Additionally, the antigrain image resize filter is controlled by the parameter *filternorm*.

alpha

[float or array-like, optional] The alpha blending value, between 0 (transparent) and 1 (opaque). If *alpha* is an array, the alpha blending values are applied pixel by pixel, and *alpha* must have the same shape as *X*.

vmin, vmax

[float, optional] When using scalar data and no explicit *norm*, *vmin* and *vmax* define the data range that the colormap covers. By default, the colormap covers the complete value range of the supplied data. It is deprecated to use *vmin*/*vmax* when *norm* is given.

origin

[{'upper', 'lower'}, default: `rcParams["image.origin"]` (default: `'upper'`)] Place the [0, 0] index of the array in the upper left or lower left corner of the axes. The convention (the default) 'upper' is typically used for matrices and images.

Note that the vertical axes points upward for 'lower' but downward for 'upper'.

See the *origin and extent in imshow* tutorial for examples and a more detailed description.

extent

[floats (left, right, bottom, top), optional] The bounding box in data coordinates that the image will fill. The image is stretched individually along x and y to fill the box.

The default extent is determined by the following conditions. Pixels have unit size in data coordinates. Their centers are on integer coordinates, and their center coordinates range from 0 to columns-1 horizontally and from 0 to rows-1 vertically.

Note that the direction of the vertical axis and thus the default values for top and bottom depend on *origin*:

- For `origin == 'upper'` the default is `(-0.5, numcols-0.5, numrows-0.5, -0.5)`.
- For `origin == 'lower'` the default is `(-0.5, numcols-0.5, -0.5, numrows-0.5)`.

See the *origin and extent in imshow* tutorial for examples and a more detailed description.

filternorm

[bool, default: True] A parameter for the antigrain image resize filter (see the antigrain documentation). If *filternorm* is set, the filter normalizes integer values and corrects the rounding errors. It doesn't do anything with the source floating point values, it

corrects only integers according to the rule of 1.0 which means that any sum of pixel weights must be equal to 1.0. So, the filter function must produce a graph of the proper shape.

filterrad

[float > 0, default: 4.0] The filter radius for filters that have a radius parameter, i.e. when interpolation is one of: 'sinc', 'lanczos' or 'blackman'.

resample

[bool, default: `rcParams["image.resample"]` (default: `True`)] When *True*, use a full resampling method. When *False*, only resample when the output image is larger than the input image.

url

[str, optional] Set the url of the created *AxesImage*. See *Artist.set_url*.

Returns

AxesImage

Other Parameters

****kwargs**

[*Artist* properties] These parameters are passed on to the constructor of the *AxesImage* artist.

See also:

matshow

Plot a matrix or an array as an image.

Notes

Unless *extent* is used, pixel centers will be located at integer coordinates. In other words: the origin will coincide with the center of pixel (0, 0).

There are two common representations for RGB images with an alpha channel:

- Straight (unassociated) alpha: R, G, and B channels represent the color of the pixel, disregarding its opacity.
- Premultiplied (associated) alpha: R, G, and B channels represent the color of the pixel, adjusted for its opacity by multiplication.

imshow expects RGB images adopting the straight (unassociated) alpha representation.

Note: In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, every other argument can also be string `s`, which is interpreted as `data[s]` (unless this raises an exception).

Objects passed as **data** must support item access (`data[s]`) and membership test (`s in data`).

Examples using `matplotlib.axes.Axes.imshow`

- sphx_glr_gallery_lines_bars_and_markers_gradient_bar.py
- sphx_glr_gallery_images_contours_and_fields_affine_image.py
- sphx_glr_gallery_images_contours_and_fields_barcode_demo.py
- sphx_glr_gallery_images_contours_and_fields_contour_demo.py
- sphx_glr_gallery_images_contours_and_fields_contour_image.py
- sphx_glr_gallery_images_contours_and_fields_image_annotated_heatmap.py
- sphx_glr_gallery_images_contours_and_fields_image_antialiasing.py
- sphx_glr_gallery_images_contours_and_fields_image_clip_path.py
- sphx_glr_gallery_images_contours_and_fields_image_demo.py
- sphx_glr_gallery_images_contours_and_fields_image_masked.py
- sphx_glr_gallery_images_contours_and_fields_image_transparency_blend.py
- sphx_glr_gallery_images_contours_and_fields_image_zcoord.py
- sphx_glr_gallery_images_contours_and_fields_interpolation_methods.py
- sphx_glr_gallery_images_contours_and_fields_layer_images.py
- sphx_glr_gallery_images_contours_and_fields_multi_image.py
- sphx_glr_gallery_images_contours_and_fields_pcolor_demo.py
- sphx_glr_gallery_images_contours_and_fields_plot_streamplot.py
- sphx_glr_gallery_images_contours_and_fields_shading_example.py
- sphx_glr_gallery_subplots_axes_and_figures_axes_box_aspect.py
- sphx_glr_gallery_subplots_axes_and_figures_zoom_inset_axes.py
- sphx_glr_gallery_text_labels_and_annotations_demo_text_path.py
- sphx_glr_gallery_color_colorbar_basics.py

- sphx_glr_gallery_color_colormap_reference.py
- sphx_glr_gallery_color_custom_cmap.py
- sphx_glr_gallery_axes_grid1_demo_anchored_direction_arrows.py
- sphx_glr_gallery_axes_grid1_demo_axes_grid2.py
- sphx_glr_gallery_axes_grid1_demo_axes_hbox_divider.py
- sphx_glr_gallery_axes_grid1_demo_colorbar_of_inset_axes.py
- sphx_glr_gallery_axes_grid1_demo_colorbar_with_axes_divider.py
- sphx_glr_gallery_axes_grid1_demo_colorbar_with_inset_locator.py
- sphx_glr_gallery_axes_grid1_inset_locator_demo2.py
- sphx_glr_gallery_axes_grid1_simple_axesgrid.py
- sphx_glr_gallery_axes_grid1_simple_axesgrid2.py
- sphx_glr_gallery_axes_grid1_simple_colorbar.py
- sphx_glr_gallery_showcase_mandelbrot.py
- sphx_glr_gallery_animation_animation_demo.py
- *Animated image using a precomputed list of images*
- sphx_glr_gallery_event_handling_image_slices_viewer.py
- sphx_glr_gallery_event_handling_viewlims.py
- sphx_glr_gallery_misc_agg_buffer_to_array.py
- sphx_glr_gallery_misc_patheffect_demo.py
- sphx_glr_gallery_specialty_plots_mri_demo.py
- sphx_glr_gallery_specialty_plots_mri_with_eeg.py
- sphx_glr_gallery_specialty_plots_topographic_hillshading.py
- sphx_glr_gallery_ticks_and_spines_colorbar_tick_labelling_demo.py
- sphx_glr_gallery_ticks_and_spines_spines_dropped.py
- *Artist tutorial*
- *Tight Layout guide*
- *Choosing Colormaps in Matplotlib*

matplotlib.axes.Axes.matshow

Axes.matshow(*self*, *Z*, ***kwargs*)

Plot the values of a 2D matrix or array as color-coded image.

The matrix will be shown the way it would be printed, with the first row at the top. Row and column numbering is zero-based.

Parameters

Z

[array-like(M, N)] The matrix to be displayed.

Returns

AxesImage

Other Parameters

****kwargs**

[*imshow* arguments]

See also:

imshow

More general function to plot data on a 2D regular raster.

Notes

This is just a convenience function wrapping *imshow* to set useful defaults for displaying a matrix. In particular:

- Set `origin='upper'`.
- Set `interpolation='nearest'`.
- Set `aspect='equal'`.
- Ticks are placed to the left and above.
- Ticks are formatted to show integer indices.

Examples using `matplotlib.axes.Axes.matshow`

- sphx_glr_gallery_images_contours_and_fields_matshow.py

matplotlib.axes.Axes.pcolor

`Axes.pcolor`(*self*, **args*, *shading=None*, *alpha=None*, *norm=None*, *cmap=None*, *vmin=None*, *vmax=None*, *data=None*, ***kwargs*)

Create a pseudocolor plot with a non-regular rectangular grid.

Call signature:

```
pcolor([X, Y,] C, **kwargs)
```

X and *Y* can be used to specify the corners of the quadrilaterals.

Hint: `pcolor()` can be very slow for large arrays. In most cases you should use the similar but much faster *`pcolormesh`* instead. See *Differences between pcolor() and pcolormesh()* for a discussion of the differences.

Parameters

C

[array-like] A scalar 2-D array. The values will be color-mapped.

X, Y

[array-like, optional] The coordinates of the corners of quadrilaterals of a pcolormesh:

```
(X[i+1, j], Y[i+1, j])       (X[i+1, j+1], Y[i+1, j+1])
                      +-----+
                      |     |
                      +-----+
   (X[i, j], Y[i, j])       (X[i, j+1], Y[i, j+1])
```

Note that the column index corresponds to the x-coordinate, and the row index corresponds to y. For details, see the *Notes* section below.

If `shading='flat'` the dimensions of *X* and *Y* should be one greater than those of *C*, and the quadrilateral is colored due to the value at `C[i, j]`. If *X*, *Y* and *C* have equal dimensions, a warning will be raised and the last row and column of *C* will be ignored.

If `shading='nearest'`, the dimensions of *X* and *Y* should be the same as those of *C* (if not, a ValueError will be raised). The color `C[i, j]` will be centered on `(X[i, j], Y[i, j])`.

If *X* and/or *Y* are 1-D arrays or column vectors they will be expanded as needed into the appropriate 2-D arrays, making a rectangular grid.

shading

[{'flat', 'nearest', 'auto'}, optional] The fill style for the quadrilateral; defaults to 'flat' or `rcParams["pcolor.shading"]` (default: `'flat'`). Possible values:

- 'flat': A solid color is used for each quad. The color of the quad (i, j), (i+1, j), (i, j+1), (i+1, j+1) is given by `C[i, j]`. The dimensions of *X* and *Y* should be one greater than those of *C*; if they are the same as *C*, then a deprecation warning is raised, and the last row and column of *C* are dropped.
- 'nearest': Each grid point will have a color centered on it, extending halfway between the adjacent grid centers. The dimensions of *X* and *Y* must be the same as *C*.
- 'auto': Choose 'flat' if dimensions of *X* and *Y* are one larger than *C*. Choose 'nearest' if dimensions are the same.

See /gallery/images_contours_and_fields/pcolormesh_grids for more description.

cmap

[str or *Colormap*, default: `rcParams["image.cmap"]` (default: `'viridis'`)] A Colormap instance or registered colormap name. The colormap maps the *C* values to colors.

norm

[*Normalize*, optional] The Normalize instance scales the data values to the canonical colormap range [0, 1] for mapping to colors. By default, the data range is mapped to the colorbar range using linear scaling.

vmin, vmax

[float, default: None] The colorbar range. If *None*, suitable min/max values are automatically chosen by the *Normalize* instance (defaults to the respective min/max values of *C* in case of the default linear scaling). It is deprecated to use *vmin*/*vmax* when *norm* is given.

edgecolors

[{'none', None, 'face', color, color sequence}, optional] The color of the edges. Defaults to 'none'. Possible values:

- 'none' or '': No edge.

- *None*: `rcParams["patch.edgecolor"]` (default: `'black'`) will be used. Note that currently `rcParams["patch.force_edgecolor"]` (default: `False`) has to be True for this to work.
- 'face': Use the adjacent face color.
- A color or sequence of colors will set the edge color.

The singular form *edgecolor* works as an alias.

alpha

[float, default: None] The alpha blending value of the face color, between 0 (transparent) and 1 (opaque). Note: The edgecolor is currently not affected by this.

snap

[bool, default: False] Whether to snap the mesh to pixel boundaries.

Returns

matplotlib.collections.Collection

Other Parameters

antialiaseds

[bool, default: False] The default *antialiaseds* is False if the default *edgecolors*="none" is used. This eliminates artificial lines at patch boundaries, and works regardless of the value of alpha. If *edgecolors* is not "none", then the default *antialiaseds* is taken from `rcParams["patch.antialiased"]` (default: `True`). Stroking the edges may be preferred if *alpha* is 1, but will cause artifacts otherwise.

****kwargs**

Additionally, the following arguments are allowed. They are passed along to the *PolyCollection* constructor:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a
alpha	float or None
animated	bool
antialiased or aa or antialiaseds	bool or list of bools
array	ndarray
capstyle	{'butt', 'round', 'projecting'}
clim	(vmin: float, vmax: float)
clip_box	*Bbox*

Table 84 – continued from previous page

Property	Description
clip_on	bool
clip_path	Patch or (Path, Transform) or None
cmap	*Colormap* or str or None
color	color or list of rgba tuples
contains	unknown
edgecolor or ec or edgecolors	color or list of colors or 'face'
facecolor or facecolors or fc	color or list of colors
figure	*Figure*
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or dashes or linestyles or ls	str or tuple or list thereof
linewidth or linewidths or lw	float or list of floats
norm	*Normalize* or None
offset_position	unknown
offsets	array-like (N, 2) or (2,)
path_effects	*AbstractPathEffect*
picker	None or bool or callable
pickradius	unknown
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
urls	list of str or None
visible	bool
zorder	float

See also:

pcolormesh

for an explanation of the differences between pcolor and pcolormesh.

imshow

If X and Y are each equidistant, *imshow* can be a faster alternative.

Notes

Masked arrays

X, *Y* and *C* may be masked arrays. If either `C[i, j]`, or one of the vertices surrounding `C[i, j]` (*X* or *Y* at `[i, j]`, `[i+1, j]`, `[i, j+1]`, `[i+1, j+1]`) is masked, nothing is plotted.

Grid orientation

The grid orientation follows the standard matrix convention: An array *C* with shape (nrows, ncolumns) is plotted with the column number as *X* and the row number as *Y*.

Note: In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, every other argument can also be string `s`, which is interpreted as `data[s]` (unless this raises an exception).

Objects passed as **data** must support item access (`data[s]`) and membership test (`s in data`).

Examples using `matplotlib.axes.Axes.pcolor`

- sphx_glr_gallery_images_contours_and_fields_pcolor_demo.py
- sphx_glr_gallery_subplots_axes_and_figures_axes_margins.py

matplotlib.axes.Axes.pcolorfast

`Axes.pcolorfast`(*self*, **args*, *alpha=None*, *norm=None*, *cmap=None*, *vmin=None*, *vmax=None*, *data=None*, ***kwargs*)

Create a pseudocolor plot with a non-regular rectangular grid.

Call signature:

```
ax.pcolorfast([X, Y], C, /, **kwargs)
```

This method is similar to *pcolor* and *pcolormesh*. It's designed to provide the fastest pcolor-type plotting with the Agg backend. To achieve this, it uses different algorithms internally depending on the complexity of the input grid (regular rectangular, non-regular rectangular or arbitrary quadrilateral).

Warning: This method is experimental. Compared to *pcolor* or *pcolormesh* it has some limitations:

- It supports only flat shading (no outlines)
- It lacks support for log scaling of the axes.
- It does not have a have a pyplot wrapper.

Parameters

C

[array-like(M, N)] The image data. Supported array shapes are:

- (M, N): an image with scalar data. The data is visualized using a colormap.
- (M, N, 3): an image with RGB values (0-1 float or 0-255 int).
- (M, N, 4): an image with RGBA values (0-1 float or 0-255 int), i.e. including transparency.

The first two dimensions (M, N) define the rows and columns of the image.

This parameter can only be passed positionally.

X, Y

[tuple or array-like, default: `(0, N), (0, M)`] *X* and *Y* are used to specify the coordinates of the quadrilaterals. There are different ways to do this:

- Use tuples `X=(xmin, xmax)` and `Y=(ymin, ymax)` to define a *uniform rectangular grid*.

  The tuples define the outer edges of the grid. All individual quadrilaterals will be of the same size. This is the fastest version.

- Use 1D arrays *X*, *Y* to specify a *non-uniform rectangular grid*.

  In this case *X* and *Y* have to be monotonic 1D arrays of length *N+1* and *M+1*, specifying the x and y boundaries of the cells.

  The speed is intermediate. Note: The grid is checked, and if found to be uniform the fast version is used.

- Use 2D arrays *X*, *Y* if you need an *arbitrary quadrilateral grid* (i.e. if the quadrilaterals are not rectangular).

  In this case *X* and *Y* are 2D arrays with shape (M + 1, N + 1), specifying the x and y coordinates of the corners of the colored quadrilaterals.

This is the most general, but the slowest to render. It may produce faster and more compact output using ps, pdf, and svg backends, however.

These arguments can only be passed positionally.

cmap

[str or *Colormap*, default: `rcParams["image.cmap"]` (default: `'viridis'`)] A Colormap instance or registered colormap name. The colormap maps the *C* values to colors.

norm

[*Normalize*, optional] The Normalize instance scales the data values to the canonical colormap range [0, 1] for mapping to colors. By default, the data range is mapped to the colorbar range using linear scaling.

vmin, vmax

[float, default: None] The colorbar range. If *None*, suitable min/max values are automatically chosen by the *Normalize* instance (defaults to the respective min/max values of *C* in case of the default linear scaling). It is deprecated to use *vmin*/*vmax* when *norm* is given.

alpha

[float, default: None] The alpha blending value, between 0 (transparent) and 1 (opaque).

snap

[bool, default: False] Whether to snap the mesh to pixel boundaries.

Returns

AxesImage **or** *PcolorImage* **or** *QuadMesh*

The return type depends on the type of grid:

- *AxesImage* for a regular rectangular grid.
- *PcolorImage* for a non-regular rectangular grid.
- *QuadMesh* for a non-rectangular grid.

Other Parameters

****kwargs**

Supported additional parameters depend on the type of grid. See return types of *image* for further description.

Notes

Note: In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, every other argument can also be string `s`, which is interpreted as `data[s]` (unless this raises an exception).

Objects passed as **data** must support item access (`data[s]`) and membership test (`s in data`).

Examples using `matplotlib.axes.Axes.pcolorfast`

- sphx_glr_gallery_images_contours_and_fields_pcolor_demo.py

matplotlib.axes.Axes.pcolormesh

Axes.**pcolormesh**(*self*, **args*, *alpha=None*, *norm=None*, *cmap=None*, *vmin=None*, *vmax=None*, *shading=None*, *antialiased=False*, *data=None*, ***kwargs*)

Create a pseudocolor plot with a non-regular rectangular grid.

Call signature:

```
pcolormesh([X, Y,] C, **kwargs)
```

X and *Y* can be used to specify the corners of the quadrilaterals.

Hint: *pcolormesh* is similar to *pcolor*. It is much faster and preferred in most cases. For a detailed discussion on the differences see *Differences between pcolor() and pcolormesh()*.

Parameters

C

[array-like] A scalar 2-D array. The values will be color-mapped.

X, Y

[array-like, optional] The coordinates of the corners of quadrilaterals of a pcolormesh:

```
(X[i+1, j], Y[i+1, j])       (X[i+1, j+1], Y[i+1, j+1])
                      +------+
                      |      |
                      +------+
   (X[i, j], Y[i, j])       (X[i, j+1], Y[i, j+1])
```

Note that the column index corresponds to the x-coordinate, and the row index corresponds to y. For details, see the *Notes* section below.

If `shading='flat'` the dimensions of *X* and *Y* should be one greater than those of *C*, and the quadrilateral is colored due to the value at `C[i, j]`. If *X*, *Y* and *C* have equal dimensions, a warning will be raised and the last row and column of *C* will be ignored.

If `shading='nearest'` or `'gouraud'`, the dimensions of *X* and *Y* should be the same as those of *C* (if not, a ValueError will be raised). For `'nearest'` the color `C[i, j]` is centered on `(X[i, j], Y[i, j])`. For `'gouraud'`, a smooth interpolation is caried out between the quadrilateral corners.

If *X* and/or *Y* are 1-D arrays or column vectors they will be expanded as needed into the appropriate 2-D arrays, making a rectangular grid.

cmap

[str or *Colormap*, default: `rcParams["image.cmap"]` (default: `'viridis'`)] A Colormap instance or registered colormap name. The colormap maps the *C* values to colors.

norm

[*Normalize*, optional] The Normalize instance scales the data values to the canonical colormap range [0, 1] for mapping to colors. By default, the data range is mapped to the colorbar range using linear scaling.

vmin, vmax

[float, default: None] The colorbar range. If *None*, suitable min/max values are automatically chosen by the *Normalize* instance (defaults to the respective min/max values of *C* in case of the default linear scaling). It is deprecated to use *vmin*/*vmax* when *norm* is given.

edgecolors

[{'none', None, 'face', color, color sequence}, optional] The color of the edges. Defaults to 'none'. Possible values:

- 'none' or '': No edge.

- *None*: `rcParams["patch.edgecolor"]` (default: `'black'`) will be used. Note that currently `rcParams["patch.force_edgecolor"]` (default: `False`) has to be True for this to work.
- 'face': Use the adjacent face color.
- A color or sequence of colors will set the edge color.

The singular form *edgecolor* works as an alias.

alpha

[float, default: None] The alpha blending value, between 0 (transparent) and 1 (opaque).

shading

[{'flat', 'nearest', 'gouraud', 'auto'}, optional] The fill style for the quadrilateral; defaults to 'flat' or `rcParams["pcolor.shading"]` (default: `'flat'`). Possible values:

- 'flat': A solid color is used for each quad. The color of the quad (i, j), (i+1, j), (i, j+1), (i+1, j+1) is given by `C[i, j]`. The dimensions of *X* and *Y* should be one greater than those of *C*; if they are the same as *C*, then a deprecation warning is raised, and the last row and column of *C* are dropped.
- 'nearest': Each grid point will have a color centered on it, extending halfway between the adjacent grid centers. The dimensions of *X* and *Y* must be the same as *C*.
- 'gouraud': Each quad will be Gouraud shaded: The color of the corners (i', j') are given by `C[i', j']`. The color values of the area in between is interpolated from the corner values. The dimensions of *X* and *Y* must be the same as *C*. When Gouraud shading is used, *edgecolors* is ignored.
- 'auto': Choose 'flat' if dimensions of *X* and *Y* are one larger than *C*. Choose 'nearest' if dimensions are the same.

See /gallery/images_contours_and_fields/pcolormesh_grids for more description.

snap

[bool, default: False] Whether to snap the mesh to pixel boundaries.

Returns

matplotlib.collections.QuadMesh

Other Parameters

**kwargs

Additionally, the following arguments are allowed. They are passed along to the *QuadMesh* constructor:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a
alpha	float or None
animated	bool
antialiased or aa or antialiaseds	bool or list of bools
array	ndarray
capstyle	{'butt', 'round', 'projecting'}
clim	(vmin: float, vmax: float)
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
cmap	*Colormap* or str or None
color	color or list of rgba tuples
contains	unknown
edgecolor or ec or edgecolors	color or list of colors or 'face'
facecolor or facecolors or fc	color or list of colors
figure	*Figure*
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or dashes or linestyles or ls	str or tuple or list thereof
linewidth or linewidths or lw	float or list of floats
norm	*Normalize* or None
offset_position	unknown
offsets	array-like (N, 2) or (2,)
path_effects	*AbstractPathEffect*
picker	None or bool or callable
pickradius	unknown
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
urls	list of str or None
visible	bool
zorder	float

See also:

pcolor

An alternative implementation with slightly different features. For a detailed discussion on the differences see *Differences between pcolor() and pcolormesh()*.

imshow

If X and Y are each equidistant, *imshow* can be a faster alternative.

Notes

Masked arrays

C may be a masked array. If `C[i, j]` is masked, the corresponding quadrilateral will be transparent. Masking of X and Y is not supported. Use *pcolor* if you need this functionality.

Grid orientation

The grid orientation follows the standard matrix convention: An array C with shape (nrows, ncolumns) is plotted with the column number as X and the row number as Y.

Differences between pcolor() and pcolormesh()

Both methods are used to create a pseudocolor plot of a 2-D array using quadrilaterals.

The main difference lies in the created object and internal data handling: While *pcolor* returns a *PolyCollection*, *pcolormesh* returns a *QuadMesh*. The latter is more specialized for the given purpose and thus is faster. It should almost always be preferred.

There is also a slight difference in the handling of masked arrays. Both *pcolor* and *pcolormesh* support masked arrays for C. However, only *pcolor* supports masked arrays for X and Y. The reason lies in the internal handling of the masked values. *pcolor* leaves out the respective polygons from the PolyCollection. *pcolormesh* sets the facecolor of the masked elements to transparent. You can see the difference when using edgecolors. While all edges are drawn irrespective of masking in a QuadMesh, the edge between two adjacent masked quadrilaterals in *pcolor* is not drawn as the corresponding polygons do not exist in the PolyCollection.

Another difference is the support of Gouraud shading in *pcolormesh*, which is not available with *pcolor*.

Note: In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, every other argument can also be string `s`, which is interpreted as `data[s]` (unless this raises an exception).

Objects passed as **data** must support item access (`data[s]`) and membership test (`s in data`).

Examples using `matplotlib.axes.Axes.pcolormesh`

- sphx_glr_gallery_images_contours_and_fields_pcolor_demo.py
- sphx_glr_gallery_images_contours_and_fields_pcolormesh_grids.py
- sphx_glr_gallery_images_contours_and_fields_pcolormesh_levels.py
- sphx_glr_gallery_images_contours_and_fields_quadmesh_demo.py
- sphx_glr_gallery_subplots_axes_and_figures_colorbar_placement.py
- sphx_glr_gallery_misc_rasterization_demo.py
- *Constrained Layout Guide*
- *Creating Colormaps in Matplotlib*
- *Colormap Normalization*

matplotlib.axes.Axes.spy

Axes.spy(*self*, *Z*, *precision=0*, *marker=None*, *markersize=None*, *aspect='equal'*, *origin='upper'*, ***kwargs*)

Plot the sparsity pattern of a 2D array.

This visualizes the non-zero values of the array.

Two plotting styles are available: image and marker. Both are available for full arrays, but only the marker style works for `scipy.sparse.spmatrix` instances.

Image style

If *marker* and *markersize* are *None*, *imshow* is used. Any extra remaining keyword arguments are passed to this method.

Marker style

If *Z* is a `scipy.sparse.spmatrix` or *marker* or *markersize* are *None*, a *Line2D* object will be returned with the value of marker determining the marker type, and any remaining keyword arguments passed to *plot*.

Parameters

Z

[array-like (M, N)] The array to be plotted.

precision

[float or 'present', default: 0] If *precision* is 0, any non-zero value will be plotted. Otherwise, values of $|Z| > precision$ will be plotted.

For `scipy.sparse.spmatrix` instances, you can also pass 'present'. In this case any value present in the array will be plotted, even if it is identically zero.

aspect

[{'equal', 'auto', None} or float, default: 'equal'] The aspect ratio of the axes. This parameter is particularly relevant for images since it determines whether data pixels are square.

This parameter is a shortcut for explicitly calling *Axes.set_aspect*. See there for further details.

- 'equal': Ensures an aspect ratio of 1. Pixels will be square.
- 'auto': The axes is kept fixed and the aspect is adjusted so that the data fit in the axes. In general, this will result in non-square pixels.
- *None*: Use `rcParams["image.aspect"]` (default: `'equal'`).

origin

[{'upper', 'lower'}, default: `rcParams["image.origin"]` (default: `'upper'`)] Place the [0, 0] index of the array in the upper left or lower left corner of the axes. The convention 'upper' is typically used for matrices and images.

Returns

AxesImage **or** *Line2D*

The return type depends on the plotting style (see above).

Other Parameters

****kwargs**

The supported additional parameters depend on the plotting style.

For the image style, you can pass the following additional parameters of *imshow*:

- *cmap*
- *alpha*
- *url*

- any *Artist* properties (passed on to the *AxesImage*)

For the marker style, you can pass any *Line2D* property except for *linestyle*:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value
alpha	float or None
animated	bool
antialiased or aa	bool
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color or c	color
contains	unknown
dash_capstyle	{'butt', 'round', 'projecting'}
dash_joinstyle	{'miter', 'round', 'bevel'}
dashes	sequence of floats (on/off ink in points) or (None, None)
data	(2, N) array or two 1D arrays
drawstyle or ds	{'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'd
figure	*Figure*
fillstyle	{'full', 'left', 'right', 'bottom', 'top', 'none'}
gid	str
in_layout	bool
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float
marker	marker style string, *Path* or *MarkerStyle*
markeredgecolor or mec	color
markeredgewidth or mew	float
markerfacecolor or mfc	color
markerfacecoloralt or mfcalt	color
markersize or ms	float
markevery	None or int or (int, int) or slice or List[int] or float or (float, float)
path_effects	*AbstractPathEffect*
picker	unknown
pickradius	float
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
solid_capstyle	{'butt', 'round', 'projecting'}
solid_joinstyle	{'miter', 'round', 'bevel'}
transform	*matplotlib.transforms.Transform*
url	str
visible	bool
xdata	1D array

Table 86 – continued from previous page

Property	Description
ydata	1D array
zorder	float

Examples using `matplotlib.axes.Axes.spy`

- sphx_glr_gallery_images_contours_and_fields_spy_demos.py

Unstructured Triangles

Axes.tripcolor	Create a pseudocolor plot of an unstructured triangular grid.
Axes.triplot	Draw a unstructured triangular grid as lines and/or markers.
Axes.tricontour	Draw contour lines on an unstructured triangular grid.
Axes.tricontourf	Draw contour regions on an unstructured triangular grid.

matplotlib.axes.Axes.tripcolor

Axes.**tripcolor**(*ax*, **args*, *alpha=1.0*, *norm=None*, *cmap=None*, *vmin=None*, *vmax=None*, *shading='flat'*, *facecolors=None*, ***kwargs*)

Create a pseudocolor plot of an unstructured triangular grid.

The triangulation can be specified in one of two ways; either:

```
tripcolor(triangulation, ...)
```

where triangulation is a *Triangulation* object, or

```
tripcolor(x, y, ...)
tripcolor(x, y, triangles, ...)
tripcolor(x, y, triangles=triangles, ...)
tripcolor(x, y, mask=mask, ...)
tripcolor(x, y, triangles, mask=mask, ...)
```

in which case a Triangulation object will be created. See *Triangulation* for a explanation of these possibilities.

The next argument must be *C*, the array of color values, either one per point in the triangulation if color values are defined at points, or one per triangle in the triangulation if color values are defined at triangles. If there are the same number of points and triangles in the triangulation it is assumed that color values

are defined at points; to force the use of color values at triangles use the kwarg `facecolors=C` instead of just `C`.

shading may be 'flat' (the default) or 'gouraud'. If *shading* is 'flat' and C values are defined at points, the color values used for each triangle are from the mean C of the triangle's three points. If *shading* is 'gouraud' then color values must be defined at points.

The remaining kwargs are the same as for *pcolor*.

Examples using `matplotlib.axes.Axes.tripcolor`

- sphx_glr_gallery_images_contours_and_fields_tripcolor_demo.py

matplotlib.axes.Axes.triplot

Axes.**triplot**(*ax*, **args*, ***kwargs*)

Draw a unstructured triangular grid as lines and/or markers.

The triangulation to plot can be specified in one of two ways; either:

```
triplot(triangulation, ...)
```

where triangulation is a *Triangulation* object, or

```
triplot(x, y, ...)
triplot(x, y, triangles, ...)
triplot(x, y, triangles=triangles, ...)
triplot(x, y, mask=mask, ...)
triplot(x, y, triangles, mask=mask, ...)
```

in which case a Triangulation object will be created. See *Triangulation* for a explanation of these possibilities.

The remaining args and kwargs are the same as for *plot*.

Returns

lines

[*Line2D*] The drawn triangles edges.

markers

[*Line2D*] The drawn marker nodes.

Examples using `matplotlib.axes.Axes.triplot`

- sphx_glr_gallery_images_contours_and_fields_tricontour_smooth_delaunay.py
- sphx_glr_gallery_images_contours_and_fields_tricontour_smooth_user.py
- sphx_glr_gallery_images_contours_and_fields_trigradient_demo.py
- sphx_glr_gallery_images_contours_and_fields_triinterp_demo.py
- sphx_glr_gallery_images_contours_and_fields_triplot_demo.py

matplotlib.axes.Axes.tricontour

Axes.**tricontour**(*ax*, **args*, ***kwargs*)

Draw contour lines on an unstructured triangular grid.

The triangulation can be specified in one of two ways; either

```
tricontour(triangulation, ...)
```

where *triangulation* is a *Triangulation* object, or

```
tricontour(x, y, ...)
tricontour(x, y, triangles, ...)
tricontour(x, y, triangles=triangles, ...)
tricontour(x, y, mask=mask, ...)
tricontour(x, y, triangles, mask=mask, ...)
```

in which case a *Triangulation* object will be created. See that class' docstring for an explanation of these cases.

The remaining arguments may be:

```
tricontour(..., Z)
```

where *Z* is the array of values to contour, one per point in the triangulation. The level values are chosen automatically.

```
tricontour(..., Z, levels)
```

contour up to *levels+1* automatically chosen contour levels (*levels* intervals).

```
tricontour(..., Z, levels)
```

draw contour lines at the values specified in sequence *levels*, which must be in increasing order.

```
tricontour(Z, **kwargs)
```

Use keyword arguments to control colors, linewidth, origin, cmap ... see below for more details.

Parameters

triangulation

[*Triangulation*, optional] The unstructured triangular grid.

If specified, then *x*, *y*, *triangles*, and *mask* are not accepted.

x, y

[array-like, optional] The coordinates of the values in *Z*.

triangles

[int array-like of shape (ntri, 3), optional] For each triangle, the indices of the three points that make up the triangle, ordered in an anticlockwise manner. If not specified, the Delaunay triangulation is calculated.

mask

[bool array-like of shape (ntri), optional] Which triangles are masked out.

Z

[array-like(N, M)] The height values over which the contour is drawn.

levels

[int or array-like, optional] Determines the number and positions of the contour lines / regions.

If an int *n*, use *MaxNLocator*, which tries to automatically choose no more than *n+1* "nice" contour levels between *vmin* and *vmax*.

If array-like, draw contour lines at the specified levels. The values must be in increasing order.

Returns

TriContourSet

Other Parameters

colors

[color string or sequence of colors, optional] The colors of the levels, i.e., the contour lines.

The sequence is cycled for the levels in ascending order. If the sequence is shorter than the number of levels, it's repeated.

As a shortcut, single color strings may be used in place of one-element lists, i.e. `'red'` instead of `['red']` to color all levels with

the same color. This shortcut does only work for color strings, not for other ways of specifying colors.

By default (value *None*), the colormap specified by *cmap* will be used.

alpha

[float, default: 1] The alpha blending value, between 0 (transparent) and 1 (opaque).

cmap

[str or *Colormap*, default: `rcParams["image.cmap"]` (default: `'viridis'`)] A *Colormap* instance or registered colormap name. The colormap maps the level values to colors.

If both *colors* and *cmap* are given, an error is raised.

norm

[*Normalize*, optional] If a colormap is used, the *Normalize* instance scales the level values to the canonical colormap range [0, 1] for mapping to colors. If not given, the default linear scaling is used.

origin

[{*None*, 'upper', 'lower', 'image'}, default: None] Determines the orientation and exact position of *Z* by specifying the position of `Z[0, 0]`. This is only relevant, if *X*, *Y* are not given.

- *None*: `Z[0, 0]` is at X=0, Y=0 in the lower left corner.
- 'lower': `Z[0, 0]` is at X=0.5, Y=0.5 in the lower left corner.
- 'upper': `Z[0, 0]` is at X=N+0.5, Y=0.5 in the upper left corner.
- 'image': Use the value from `rcParams["image.origin"]` (default: `'upper'`).

extent

[(x0, x1, y0, y1), optional] If *origin* is not *None*, then *extent* is interpreted as in *imshow*: it gives the outer pixel boundaries. In this case, the position of Z[0, 0] is the center of the pixel, not a corner. If *origin* is *None*, then (*x0*, *y0*) is the position of Z[0, 0], and (*x1*, *y1*) is the position of Z[-1, -1].

This argument is ignored if *X* and *Y* are specified in the call to contour.

locator

[ticker.Locator subclass, optional] The locator is used to determine the contour levels if they are not given explicitly via *levels*. Defaults to *MaxNLocator*.

extend

[{'neither', 'both', 'min', 'max'}, default: 'neither'] Determines the `tricontour`-coloring of values that are outside the *levels* range.

If 'neither', values outside the *levels* range are not colored. If 'min', 'max' or 'both', color the values below, above or below and above the *levels* range.

Values below `min(levels)` and above `max(levels)` are mapped to the under/over values of the *Colormap*. Note that most colormaps do not have dedicated colors for these by default, so that the over and under values are the edge values of the colormap. You may want to set these values explicitly using *Colormap.set_under* and *Colormap.set_over*.

Note: An existing *TriContourSet* does not get notified if properties of its colormap are changed. Therefore, an explicit call to *ContourSet.changed()* is needed after modifying the colormap. The explicit call can be left out, if a colorbar is assigned to the *TriContourSet* because it internally calls *ContourSet.changed()*.

---

xunits, yunits

[registered units, optional] Override axis units by specifying an instance of a *matplotlib.units.ConversionInterface*.

linewidths

[float or array-like, default: `rcParams["contour.linewidth"]` (default: `None`)] The line width of the contour lines.

If a number, all levels will be plotted with this linewidth.

If a sequence, the levels in ascending order will be plotted with the linewidths in the order specified.

If None, this falls back to `rcParams["lines.linewidth"]` (default: `1.5`).

linestyles

[{*None*, 'solid', 'dashed', 'dashdot', 'dotted'}, optional] If *linestyles* is *None*, the default is 'solid' unless the lines are monochrome. In that case, negative contours will take their linestyle from `rcParams["contour.negative_linestyle"]` (default: `'dashed'`) setting.

linestyles can also be an iterable of the above strings specifying a set of linestyles to be used. If this iterable is shorter than the number of contour levels it will be repeated as necessary.

Examples using `matplotlib.axes.Axes.tricontour`

- sphx_glr_gallery_images_contours_and_fields_irregulardatagrid.py
- sphx_glr_gallery_images_contours_and_fields_tricontour_demo.py
- sphx_glr_gallery_images_contours_and_fields_tricontour_smooth_delaunay.py
- sphx_glr_gallery_images_contours_and_fields_tricontour_smooth_user.py
- sphx_glr_gallery_images_contours_and_fields_trigradient_demo.py
- sphx_glr_gallery_mplot3d_tricontour3d.py

matplotlib.axes.Axes.tricontourf

Axes.**tricontourf**(*ax*, **args*, ***kwargs*)

Draw contour regions on an unstructured triangular grid.

The triangulation can be specified in one of two ways; either

```
tricontourf(triangulation, ...)
```

where *triangulation* is a *Triangulation* object, or

```
tricontourf(x, y, ...)
tricontourf(x, y, triangles, ...)
tricontourf(x, y, triangles=triangles, ...)
tricontourf(x, y, mask=mask, ...)
tricontourf(x, y, triangles, mask=mask, ...)
```

in which case a *Triangulation* object will be created. See that class' docstring for an explanation of these cases.

The remaining arguments may be:

```
tricontourf(..., Z)
```

where *Z* is the array of values to contour, one per point in the triangulation. The level values are chosen automatically.

```
tricontourf(..., Z, levels)
```

contour up to *levels+1* automatically chosen contour levels (*levels* intervals).

```
tricontourf(..., Z, levels)
```

draw contour regions at the values specified in sequence *levels*, which must be in increasing order.

```
tricontourf(Z, **kwargs)
```

Use keyword arguments to control colors, linewidth, origin, cmap ... see below for more details.

Parameters

triangulation

[*Triangulation*, optional] The unstructured triangular grid.

If specified, then *x*, *y*, *triangles*, and *mask* are not accepted.

x, y

[array-like, optional] The coordinates of the values in *Z*.

triangles

[int array-like of shape (ntri, 3), optional] For each triangle, the indices of the three points that make up the triangle, ordered in an anticlockwise manner. If not specified, the Delaunay triangulation is calculated.

mask

[bool array-like of shape (ntri), optional] Which triangles are masked out.

Z

[array-like(N, M)] The height values over which the contour is drawn.

levels

[int or array-like, optional] Determines the number and positions of the contour lines / regions.

If an int *n*, use *MaxNLocator*, which tries to automatically choose no more than *n+1* "nice" contour levels between *vmin* and *vmax*.

If array-like, draw contour lines at the specified levels. The values must be in increasing order.

Returns

TriContourSet

Other Parameters

colors

[color string or sequence of colors, optional] The colors of the levels, i.e., the contour regions.

The sequence is cycled for the levels in ascending order. If the sequence is shorter than the number of levels, it's repeated.

As a shortcut, single color strings may be used in place of one-element lists, i.e. `'red'` instead of `['red']` to color all levels with the same color. This shortcut does only work for color strings, not for other ways of specifying colors.

By default (value *None*), the colormap specified by *cmap* will be used.

alpha

[float, default: 1] The alpha blending value, between 0 (transparent) and 1 (opaque).

cmap

[str or *Colormap*, default: `rcParams["image.cmap"]` (default: `'viridis'`)] A *Colormap* instance or registered colormap name. The colormap maps the level values to colors.

If both *colors* and *cmap* are given, an error is raised.

norm

[*Normalize*, optional] If a colormap is used, the *Normalize* instance scales the level values to the canonical colormap range [0, 1] for mapping to colors. If not given, the default linear scaling is used.

origin

[{*None*, 'upper', 'lower', 'image'}, default: None] Determines the orientation and exact position of *Z* by specifying the position of `Z[0, 0]`. This is only relevant, if *X*, *Y* are not given.

- *None*: `Z[0, 0]` is at X=0, Y=0 in the lower left corner.
- 'lower': `Z[0, 0]` is at X=0.5, Y=0.5 in the lower left corner.
- 'upper': `Z[0, 0]` is at X=N+0.5, Y=0.5 in the upper left corner.
- 'image': Use the value from `rcParams["image.origin"]` (default: `'upper'`).

extent

[(x0, x1, y0, y1), optional] If *origin* is not *None*, then *extent* is interpreted as in *imshow*: it gives the outer pixel boundaries. In this case, the position of Z[0, 0] is the center of the pixel, not a corner. If *origin* is *None*, then (*x0*, *y0*) is the position of Z[0, 0], and (*x1*, *y1*) is the position of Z[-1, -1].

This argument is ignored if *X* and *Y* are specified in the call to contour.

locator

[ticker.Locator subclass, optional] The locator is used to determine the contour levels if they are not given explicitly via *levels*. Defaults to *MaxNLocator*.

extend

[{'neither', 'both', 'min', 'max'}, default: 'neither'] Determines the `tricontourf`-coloring of values that are outside the *levels* range.

If 'neither', values outside the *levels* range are not colored. If 'min', 'max' or 'both', color the values below, above or below and above the *levels* range.

Values below `min(levels)` and above `max(levels)` are mapped to the under/over values of the *Colormap*. Note that most colormaps do not have dedicated colors for these by default, so that the over and under values are the edge values of the colormap. You may want to set these values explicitly using *Colormap.set_under* and *Colormap.set_over*.

Note: An existing *TriContourSet* does not get notified if properties of its colormap are changed. Therefore, an explicit call to *ContourSet.changed()* is needed after modifying the colormap. The explicit call can be left out, if a colorbar is assigned to the *TriContourSet* because it internally calls *ContourSet.changed()*.

---

xunits, yunits

[registered units, optional] Override axis units by specifying an instance of a *matplotlib.units.ConversionInterface*.

antialiased

[bool, default: True] Whether to use antialiasing.

Notes

tricontourf fills intervals that are closed at the top; that is, for boundaries *z1* and *z2*, the filled region is:

```
z1 < Z <= z2
```

except for the lowest interval, which is closed on both sides (i.e. it includes the lowest value).

Examples using `matplotlib.axes.Axes.tricontourf`

- sphx_glr_gallery_images_contours_and_fields_irregulardatagrid.py
- sphx_glr_gallery_images_contours_and_fields_tricontour_demo.py
- sphx_glr_gallery_images_contours_and_fields_tricontour_smooth_delaunay.py
- sphx_glr_gallery_images_contours_and_fields_tricontour_smooth_user.py
- sphx_glr_gallery_images_contours_and_fields_triinterp_demo.py
- sphx_glr_gallery_mplot3d_tricontourf3d.py

Text and Annotations

Axes.annotate	Annotate the point *xy* with text *text*.
Axes.text	Add text to the axes.
Axes.table	Add a table to an *Axes*.
Axes.arrow	Add an arrow to the axes.
Axes.inset_axes	Add a child inset axes to this existing axes.
Axes.indicate_inset	Add an inset indicator to the axes.
Axes.indicate_inset_zoom	Add an inset indicator rectangle to the axes based on the axis limits for an *inset_ax* and draw connectors between *inset_ax* and the rectangle.
Axes.secondary_xaxis	Add a second x-axis to this axes.
Axes.secondary_yaxis	Add a second y-axis to this axes.

matplotlib.axes.Axes.annotate

Axes.**annotate**(*self*, *text*, *xy*, **args*, ***kwargs*)

Annotate the point *xy* with text *text*.

In the simplest form, the text is placed at *xy*.

Optionally, the text can be displayed in another position *xytext*. An arrow pointing from the text to the annotated point *xy* can then be added by defining *arrowprops*.

Parameters

text

[str] The text of the annotation. *s* is a deprecated synonym for this parameter.

xy

[(float, float)] The point *(x, y)* to annotate. The coordinate system is determined by *xycoords*.

xytext

[(float, float), default: *xy*] The position *(x, y)* to place the text at. The coordinate system is determined by *textcoords*.

xycoords

[str or *Artist* or *Transform* or callable or (float, float), default: 'data'] The coordinate system that *xy* is given in. The following types of values are supported:

- One of the following strings:

Value	Description
'figure points'	Points from the lower left of the figure
'figure pixels'	Pixels from the lower left of the figure
'figure fraction'	Fraction of figure from lower left
'axes points'	Points from lower left corner of axes
'axes pixels'	Pixels from lower left corner of axes
'axes fraction'	Fraction of axes from lower left
'data'	Use the coordinate system of the object being annotated (default)
'polar'	*(theta, r)* if not native 'data' coordinates

- An *Artist*: *xy* is interpreted as a fraction of the artist's *Bbox*. E.g. *(0, 0)* would be the lower left corner of the bounding box and *(0.5, 1)* would be the center top of the bounding box.
- A *Transform* to transform *xy* to screen coordinates.
- A function with one of the following signatures:

```
def transform(renderer) -> Bbox
def transform(renderer) -> Transform
```

  where *renderer* is a *RendererBase* subclass.

  The result of the function is interpreted like the *Artist* and *Transform* cases above.

- A tuple *(xcoords, ycoords)* specifying separate coordinate systems for *x* and *y*. *xcoords* and *ycoords* must each be of one of the above described types.

See *Advanced Annotations* for more details.

textcoords

[str or *Artist* or *Transform* or callable or (float, float), default: value of *xycoords*] The coordinate system that *xytext* is given in.

All *xycoords* values are valid as well as the following strings:

Value	Description
'offset points'	Offset (in points) from the *xy* value
'offset pixels'	Offset (in pixels) from the *xy* value

arrowprops

[dict, optional] The properties used to draw a *FancyArrowPatch* arrow between the positions *xy* and *xytext*.

If *arrowprops* does not contain the key 'arrowstyle' the allowed keys are:

Key	Description
width	The width of the arrow in points
headwidth	The width of the base of the arrow head in points
headlength	The length of the arrow head in points
shrink	Fraction of total length to shrink from both ends
?	Any key to *matplotlib.patches.FancyArrowPatch*

If *arrowprops* contains the key 'arrowstyle' the above keys are forbidden. The allowed values of `'arrowstyle'` are:

Name	Attrs
`'-'`	None
`'->'`	head_length=0.4,head_width=0.2
`'-['`	widthB=1.0,lengthB=0.2,angleB=None
`'|-|'`	widthA=1.0,widthB=1.0
`'-|>'`	head_length=0.4,head_width=0.2
`'<-'`	head_length=0.4,head_width=0.2
`'<->'`	head_length=0.4,head_width=0.2
`'<|-'`	head_length=0.4,head_width=0.2
`'<|-|>'`	head_length=0.4,head_width=0.2
`'fancy'`	head_length=0.4,head_width=0.4,tail_width=0.4
`'simple'`	head_length=0.5,head_width=0.5,tail_width=0.2
`'wedge'`	tail_width=0.3,shrink_factor=0.5

Valid keys for *FancyArrowPatch* are:

Key	Description
arrowstyle	the arrow style
connectionstyle	the connection style
relpos	default is (0.5, 0.5)
patchA	default is bounding box of the text
patchB	default is None
shrinkA	default is 2 points
shrinkB	default is 2 points
mutation_scale	default is text size (in points)
mutation_aspect	default is 1.
?	any key for *matplotlib.patches.PathPatch*

Defaults to None, i.e. no arrow is drawn.

annotation_clip

[bool or None, default: None] Whether to draw the annotation when the annotation point *xy* is outside the axes area.

- If *True*, the annotation will only be drawn when *xy* is within the axes.
- If *False*, the annotation will always be drawn.
- If *None*, the annotation will only be drawn when *xy* is within the axes and *xycoords* is 'data'.

****kwargs**

Additional kwargs are passed to *Text*.

Returns

Annotation

See also:

Advanced Annotations

Examples using `matplotlib.axes.Axes.annotate`

- sphx_glr_gallery_lines_bars_and_markers_barchart.py
- sphx_glr_gallery_lines_bars_and_markers_broken_barh.py
- sphx_glr_gallery_lines_bars_and_markers_timeline.py
- sphx_glr_gallery_subplots_axes_and_figures_gridspec_and_subplots.py
- sphx_glr_gallery_statistics_barchart_demo.py

- sphx_glr_gallery_pie_and_polar_charts_pie_and_donut_labels.py
- sphx_glr_gallery_text_labels_and_annotations_annotation_demo.py
- sphx_glr_gallery_text_labels_and_annotations_fancyarrow_demo.py
- sphx_glr_gallery_text_labels_and_annotations_tex_demo.py
- sphx_glr_gallery_pyplots_annotate_transform.py
- sphx_glr_gallery_pyplots_annotation_basic.py
- sphx_glr_gallery_pyplots_annotation_polar.py
- sphx_glr_gallery_pyplots_text_commands.py
- sphx_glr_gallery_shapes_and_collections_donut.py
- sphx_glr_gallery_axisartist_axis_direction_demo_step02.py
- sphx_glr_gallery_axisartist_axis_direction_demo_step03.py
- sphx_glr_gallery_axisartist_axis_direction_demo_step04.py
- sphx_glr_gallery_axisartist_demo_axis_direction.py
- sphx_glr_gallery_axisartist_simple_axis_pad.py
- sphx_glr_gallery_showcase_anatomy.py
- sphx_glr_gallery_showcase_xkcd.py
- sphx_glr_gallery_misc_patheffect_demo.py
- sphx_glr_gallery_units_annotate_with_units.py
- sphx_glr_gallery_userdemo_annotate_explain.py
- sphx_glr_gallery_userdemo_annotate_simple01.py
- sphx_glr_gallery_userdemo_annotate_simple02.py
- sphx_glr_gallery_userdemo_annotate_simple03.py
- sphx_glr_gallery_userdemo_annotate_simple04.py
- sphx_glr_gallery_userdemo_annotate_simple_coord01.py
- sphx_glr_gallery_userdemo_annotate_simple_coord02.py
- sphx_glr_gallery_userdemo_annotate_simple_coord03.py
- sphx_glr_gallery_userdemo_connectionstyle_demo.py
- sphx_glr_gallery_userdemo_simple_annotate01.py
- *Customizing Figure Layouts Using GridSpec and Other Functions*
- *Blitting tutorial*
- *Transformations Tutorial*
- *Text in Matplotlib Plots*

- *Annotations*

matplotlib.axes.Axes.text

`Axes.text`(*self*, *x*, *y*, *s*, *fontdict=None*, ***kwargs*)

Add text to the axes.

Add the text *s* to the axes at location *x*, *y* in data coordinates.

Parameters

x, y

[float] The position to place the text. By default, this is in data coordinates. The coordinate system can be changed using the *transform* parameter.

s

[str] The text.

fontdict

[dict, default: None] A dictionary to override the default text properties. If fontdict is None, the defaults are determined by *rcParams*.

Returns

Text

The created *Text* instance.

Other Parameters

****kwargs**

[*Text* properties.] Other miscellaneous text parameters.

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array an
alpha	float or None
animated	bool
backgroundcolor	color
bbox	dict with properties for *patches.FancyBboxPatch*
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color or c	color

Tab

Property	Description
contains	unknown
figure	*Figure*
fontfamily or family	{FONTNAME, 'serif', 'sans-serif', 'cursive', 'fantasy', '
fontproperties or font or font_properties	*font_manager.FontProperties* or `str` or `pathlib.Path`
fontsize or size	float or {'xx-small', 'x-small', 'small', 'medium', 'large',
fontstretch or stretch	{a numeric value in range 0-1000, 'ultra-condensed',
fontstyle or style	{'normal', 'italic', 'oblique'}
fontvariant or variant	{'normal', 'small-caps'}
fontweight or weight	{a numeric value in range 0-1000, 'ultralight', 'light',
gid	str
horizontalalignment or ha	{'center', 'right', 'left'}
in_layout	bool
label	object
linespacing	float (multiple of font size)
multialignment or ma	{'left', 'right', 'center'}
path_effects	*AbstractPathEffect*
picker	None or bool or callable
position	(float, float)
rasterized	bool or None
rotation	float or {'vertical', 'horizontal'}
rotation_mode	{None, 'default', 'anchor'}
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
text	object
transform	*Transform*
url	str
usetex	bool or None
verticalalignment or va	{'center', 'top', 'bottom', 'baseline', 'center_baseline'}
visible	bool
wrap	bool
x	float
y	float
zorder	float

Examples

Individual keyword arguments can be used to override any given parameter:

```
>>> text(x, y, s, fontsize=12)
```

The default transform specifies that text is in data coords, alternatively, you can specify text in axis coords ((0, 0) is lower-left and (1, 1) is upper-right). The example below places text in the center of the axes:

```
>>> text(0.5, 0.5, 'matplotlib', horizontalalignment='center',
...      verticalalignment='center', transform=ax.transAxes)
```

You can put a rectangular box around the text instance (e.g., to set a background color) by using the keyword *bbox*. *bbox* is a dictionary of *Rectangle* properties. For example:

```
>>> text(x, y, s, bbox=dict(facecolor='red', alpha=0.5))
```

Examples using `matplotlib.axes.Axes.text`

- sphx_glr_gallery_lines_bars_and_markers_horizontal_barchart_distribution.py
- sphx_glr_gallery_lines_bars_and_markers_joinstyle.py
- sphx_glr_gallery_lines_bars_and_markers_marker_reference.py
- sphx_glr_gallery_images_contours_and_fields_demo_bboximage.py
- sphx_glr_gallery_images_contours_and_fields_image_annotated_heatmap.py
- sphx_glr_gallery_statistics_boxplot_demo.py
- sphx_glr_gallery_pie_and_polar_charts_bar_of_pie.py
- sphx_glr_gallery_text_labels_and_annotations_accented_text.py
- sphx_glr_gallery_text_labels_and_annotations_demo_text_rotation_mode.py
- sphx_glr_gallery_text_labels_and_annotations_mathtext_demo.py
- sphx_glr_gallery_text_labels_and_annotations_tex_demo.py
- sphx_glr_gallery_text_labels_and_annotations_text_alignment.py
- sphx_glr_gallery_text_labels_and_annotations_text_rotation_relative_to_line.py
- sphx_glr_gallery_text_labels_and_annotations_usetex_baseline_test.py
- sphx_glr_gallery_text_labels_and_annotations_watermark_text.py
- sphx_glr_gallery_pyplots_pyplot_mathtext.py
- sphx_glr_gallery_pyplots_text_commands.py
- sphx_glr_gallery_color_named_colors.py
- sphx_glr_gallery_shapes_and_collections_fancybox_demo.py
- sphx_glr_gallery_showcase_anatomy.py
- sphx_glr_gallery_showcase_bachelors_degrees_by_gender.py
- sphx_glr_gallery_showcase_integral.py
- sphx_glr_gallery_showcase_mandelbrot.py

- *The double pendulum problem*
- *MATPLOTLIB UNCHAINED*
- sphx_glr_gallery_event_handling_data_browser.py
- sphx_glr_gallery_event_handling_pick_event_demo2.py
- sphx_glr_gallery_misc_cursor_demo.py
- sphx_glr_gallery_misc_rasterization_demo.py
- sphx_glr_gallery_mplot3d_text3d.py
- sphx_glr_gallery_recipes_placing_text_boxes.py
- sphx_glr_gallery_specialty_plots_anscombe.py
- sphx_glr_gallery_ticks_and_spines_tick-formatters.py
- sphx_glr_gallery_userdemo_annotate_explain.py
- sphx_glr_gallery_userdemo_annotate_text_arrow.py
- sphx_glr_gallery_userdemo_connectionstyle_demo.py
- sphx_glr_gallery_userdemo_custom_boxstyle01.py
- sphx_glr_gallery_userdemo_simple_annotate01.py
- *The Lifecycle of a Plot*
- *Artist tutorial*
- *Path Tutorial*
- *Transformations Tutorial*
- *Specifying Colors*
- *Text in Matplotlib Plots*
- *Text properties and layout*

matplotlib.axes.Axes.table

`Axes.table`(*ax*, *cellText=None*, *cellColours=None*, *cellLoc='right'*, *colWidths=None*, *rowLabels=None*, *rowColours=None*, *rowLoc='left'*, *colLabels=None*, *colColours=None*, *colLoc='center'*, *loc='bottom'*, *bbox=None*, *edges='closed'*, ***kwargs*)

Add a table to an *Axes*.

At least one of *cellText* or *cellColours* must be specified. These parameters must be 2D lists, in which the outer lists define the rows and the inner list define the column values per row. Each row must have the same number of elements.

The table can optionally have row and column headers, which are configured using *rowLabels*, *rowColours*, *rowLoc* and *colLabels*, *colColours*, *colLoc* respectively.

For finer grained control over tables, use the *Table* class and add it to the axes with *Axes.add_table*.

Parameters

cellText

[2D list of str, optional] The texts to place into the table cells.

Note: Line breaks in the strings are currently not accounted for and will result in the text exceeding the cell boundaries.

cellColours

[2D list of colors, optional] The background colors of the cells.

cellLoc

[{'left', 'center', 'right'}, default: 'right'] The alignment of the text within the cells.

colWidths

[list of float, optional] The column widths in units of the axes. If not given, all columns will have a width of *1 / ncols*.

rowLabels

[list of str, optional] The text of the row header cells.

rowColours

[list of colors, optional] The colors of the row header cells.

rowLoc

[{'left', 'center', 'right'}, default: 'left'] The text alignment of the row header cells.

colLabels

[list of str, optional] The text of the column header cells.

colColours

[list of colors, optional] The colors of the column header cells.

colLoc

[{'left', 'center', 'right'}, default: 'left'] The text alignment of the column header cells.

loc

[str, optional] The position of the cell with respect to *ax*. This must be one of the *codes*.

bbox

[*Bbox*, optional] A bounding box to draw the table into. If this is not *None*, this overrides *loc*.

edges

[substring of 'BRTL' or {'open', 'closed', 'horizontal', 'vertical'}] The cell edges to be drawn with a line. See also *visible_edges*.

Returns

Table

The created table.

Other Parameters

****kwargs**

Table properties.

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array
alpha	float or None
animated	bool
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
contains	unknown
figure	*Figure*
fontsize	float
gid	str
in_layout	bool
label	object
path_effects	*AbstractPathEffect*
picker	None or bool or callable
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
visible	bool
zorder	float

Examples using `matplotlib.axes.Axes.table`

matplotlib.axes.Axes.arrow

`Axes.`**`arrow`**(*self*, *x*, *y*, *dx*, *dy*, ***kwargs*)

Add an arrow to the axes.

This draws an arrow from `(x, y)` to `(x+dx, y+dy)`.

Parameters

x, y

[float] The x and y coordinates of the arrow base.

dx, dy

[float] The length of the arrow along x and y direction.

width: float, default: 0.001

Width of full arrow tail.

length_includes_head: bool, default: False

True if head is to be counted in calculating the length.

head_width: float or None, default: 3*width

Total width of the full arrow head.

head_length: float or None, default: 1.5*head_width

Length of arrow head.

shape: ['full', 'left', 'right'], default: 'full'

Draw the left-half, right-half, or full arrow.

overhang: float, default: 0

Fraction that the arrow is swept back (0 overhang means triangular shape). Can be negative or greater than one.

head_starts_at_zero: bool, default: False

If True, the head starts being drawn at coordinate 0 instead of ending at coordinate 0.

****kwargs**

Patch properties:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and retur
alpha	float or None

Table 90 – continued from previous page

Property	Description
animated	bool
antialiased or aa	unknown
capstyle	{'butt', 'round', 'projecting'}
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color	color
contains	unknown
edgecolor or ec	color or None or 'auto'
facecolor or fc	color or None
figure	*Figure*
fill	bool
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float or None
path_effects	*AbstractPathEffect*
picker	None or bool or callable
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
visible	bool
zorder	float

Returns

FancyArrow

The created *FancyArrow* object.

Notes

The resulting arrow is affected by the axes aspect ratio and limits. This may produce an arrow whose head is not square with its stem. To create an arrow whose head is square with its stem, use *annotate()* for example:

```
>>> ax.annotate("", xy=(0.5, 0.5), xytext=(0, 0),
...             arrowprops=dict(arrowstyle="->"))
```

Examples using `matplotlib.axes.Axes.arrow`

- sphx_glr_gallery_text_labels_and_annotations_arrow_simple_demo.py

matplotlib.axes.Axes.inset_axes

Axes.**inset_axes**(*self*, *bounds*, *, *transform=None*, *zorder=5*, ***kwargs*)

Add a child inset axes to this existing axes.

Parameters

bounds

[[x0, y0, width, height]] Lower-left corner of inset axes, and its width and height.

transform

[*Transform*] Defaults to `ax.transAxes`, i.e. the units of *rect* are in axes-relative coordinates.

zorder

[number] Defaults to 5 (same as *Axes.legend*). Adjust higher or lower to change whether it is above or below data plotted on the parent axes.

****kwargs**

Other keyword arguments are passed on to the child *Axes*.

Returns

ax

The created *Axes* instance.

Warning: This method is experimental as of 3.0, and the API may change.

Examples

This example makes two inset axes, the first is in axes-relative coordinates, and the second in data-coordinates:

```
fig, ax = plt.subplots()
ax.plot(range(10))
axin1 = ax.inset_axes([0.8, 0.1, 0.15, 0.15])
```

(continues on next page)

(continued from previous page)

```
axin2 = ax.inset_axes(
        [5, 7, 2.3, 2.3], transform=ax.transData)
```

Examples using `matplotlib.axes.Axes.inset_axes`

- sphx_glr_gallery_subplots_axes_and_figures_zoom_inset_axes.py

matplotlib.axes.Axes.indicate_inset

Axes.**indicate_inset**(*self*, *bounds*, *inset_ax=None*, ***, *transform=None*, *facecolor='none'*, *edgecolor='0.5'*, *alpha=0.5*, *zorder=4.99*, ***kwargs*)

Add an inset indicator to the axes. This is a rectangle on the plot at the position indicated by *bounds* that optionally has lines that connect the rectangle to an inset axes (*Axes.inset_axes*).

Parameters

bounds

[[x0, y0, width, height]] Lower-left corner of rectangle to be marked, and its width and height.

inset_ax

[*Axes*] An optional inset axes to draw connecting lines to. Two lines are drawn connecting the indicator box to the inset axes on corners chosen so as to not overlap with the indicator box.

transform

[*Transform*] Transform for the rectangle coordinates. Defaults to `ax.transAxes`, i.e. the units of *rect* are in axes-relative coordinates.

facecolor

[color, default: 'none'] Facecolor of the rectangle.

edgecolor

[color, default: '0.5'] Color of the rectangle and color of the connecting lines.

alpha

[float, default: 0.5] Transparency of the rectangle and connector lines.

zorder

[float, default: 4.99] Drawing order of the rectangle and connector lines. The default, 4.99, is just below the default level of inset axes.

****kwargs**

Other keyword arguments are passed on to the *Rectangle* patch:

%(Rectangle)s

Returns

rectangle_patch

[*patches.Rectangle*] The indicator frame.

connector_lines

[4-tuple of *patches.ConnectionPatch*] The four connector lines connecting to (lower_left, upper_left, lower_right upper_right) corners of *inset_ax*. Two lines are set with visibility to *False*, but the user can set the visibility to True if the automatic choice is not deemed correct.

Warning: This method is experimental as of 3.0, and the API may change.

Examples using `matplotlib.axes.Axes.indicate_inset`

matplotlib.axes.Axes.indicate_inset_zoom

`Axes.indicate_inset_zoom`(*self*, *inset_ax*, ***kwargs*)

Add an inset indicator rectangle to the axes based on the axis limits for an *inset_ax* and draw connectors between *inset_ax* and the rectangle.

Parameters

inset_ax

[*Axes*] Inset axes to draw connecting lines to. Two lines are drawn connecting the indicator box to the inset axes on corners chosen so as to not overlap with the indicator box.

****kwargs**

Other keyword arguments are passed on to *Axes.indicate_inset*

Returns

rectangle_patch

[*patches.Rectangle*] Rectangle artist.

connector_lines

[4-tuple of *patches.ConnectionPatch*] Each of four connector lines coming from the rectangle drawn on this axis, in the order lower left, upper left, lower right, upper right. Two are set with visibility to *False*, but the user can set the visibility to *True* if the automatic choice is not deemed correct.

Warning: This method is experimental as of 3.0, and the API may change.

Examples using `matplotlib.axes.Axes.indicate_inset_zoom`

- sphx_glr_gallery_subplots_axes_and_figures_zoom_inset_axes.py

matplotlib.axes.Axes.secondary_xaxis

`Axes.secondary_xaxis`(*self*, *location*, *, *functions=None*, ***kwargs*)

Add a second x-axis to this axes.

For example if we want to have a second scale for the data plotted on the xaxis.

Parameters

location

[{'top', 'bottom', 'left', 'right'} or float] The position to put the secondary axis. Strings can be 'top' or 'bottom' for orientation='x' and 'right' or 'left' for orientation='y'. A float indicates the relative position on the parent axes to put the new axes, 0.0 being the bottom (or left) and 1.0 being the top (or right).

functions

[2-tuple of func, or Transform with an inverse] If a 2-tuple of functions, the user specifies the transform function and its inverse. i.e. `functions=(lambda x: 2 / x, lambda x: 2 / x)` would be an reciprocal transform with a factor of 2.

The user can also directly supply a subclass of *transforms.Transform* so long as it has an inverse.

See /gallery/subplots_axes_and_figures/secondary_axis for examples of making these conversions.

Returns

ax

[axes._secondary_axes.SecondaryAxis]

Other Parameters

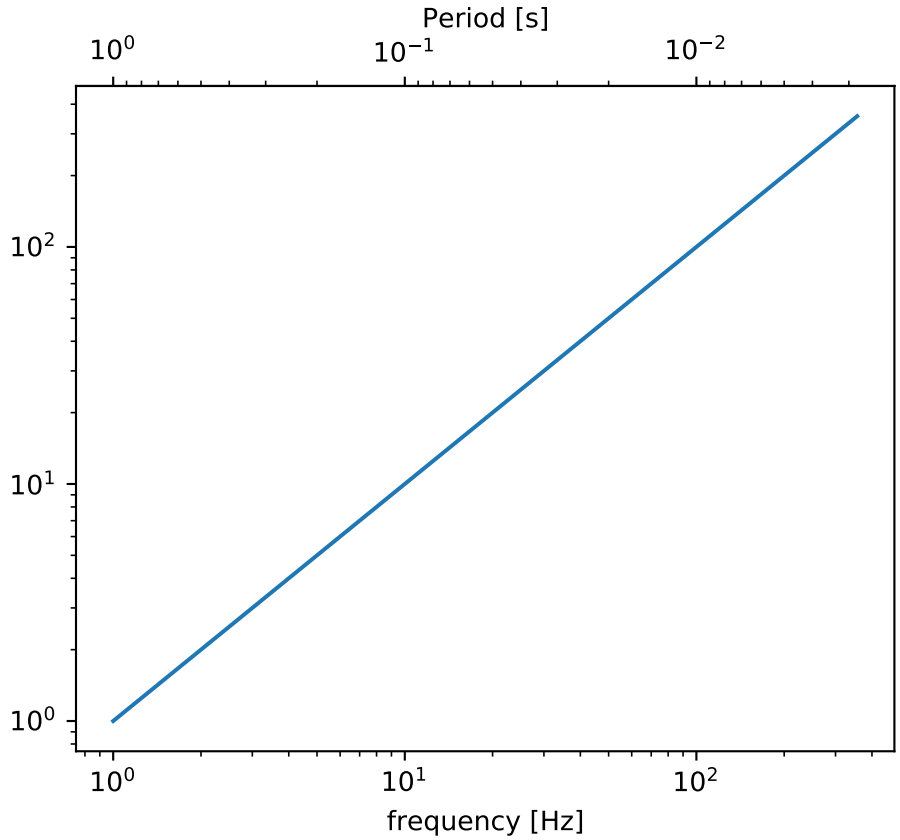
****kwargs**

[*Axes* properties.] Other miscellaneous axes parameters.

Warning: This method is experimental as of 3.1, and the API may change.

Examples

The main axis shows frequency, and the secondary axis shows period.



Examples using `matplotlib.axes.Axes.secondary_xaxis`

- sphx_glr_gallery_subplots_axes_and_figures_secondary_axis.py

matplotlib.axes.Axes.secondary_yaxis

`Axes.secondary_yaxis`(*self*, *location*, *, *functions=None*, ***kwargs*)

Add a second y-axis to this axes.

For example if we want to have a second scale for the data plotted on the yaxis.

Parameters

location

[{'top', 'bottom', 'left', 'right'} or float] The position to put the secondary axis. Strings can be 'top' or 'bottom' for orientation='x' and 'right' or 'left' for orientation='y'. A float indicates the relative position on the parent axes to put the new axes, 0.0 being the bottom (or left) and 1.0 being the top (or right).

functions

[2-tuple of func, or Transform with an inverse] If a 2-tuple of functions, the user specifies the transform function and its inverse. i.e. `functions=(lambda x: 2 / x, lambda x: 2 / x)` would be an reciprocal transform with a factor of 2.

The user can also directly supply a subclass of *transforms.Transform* so long as it has an inverse.

See /gallery/subplots_axes_and_figures/secondary_axis for examples of making these conversions.

Returns

ax

[axes._secondary_axes.SecondaryAxis]

Other Parameters

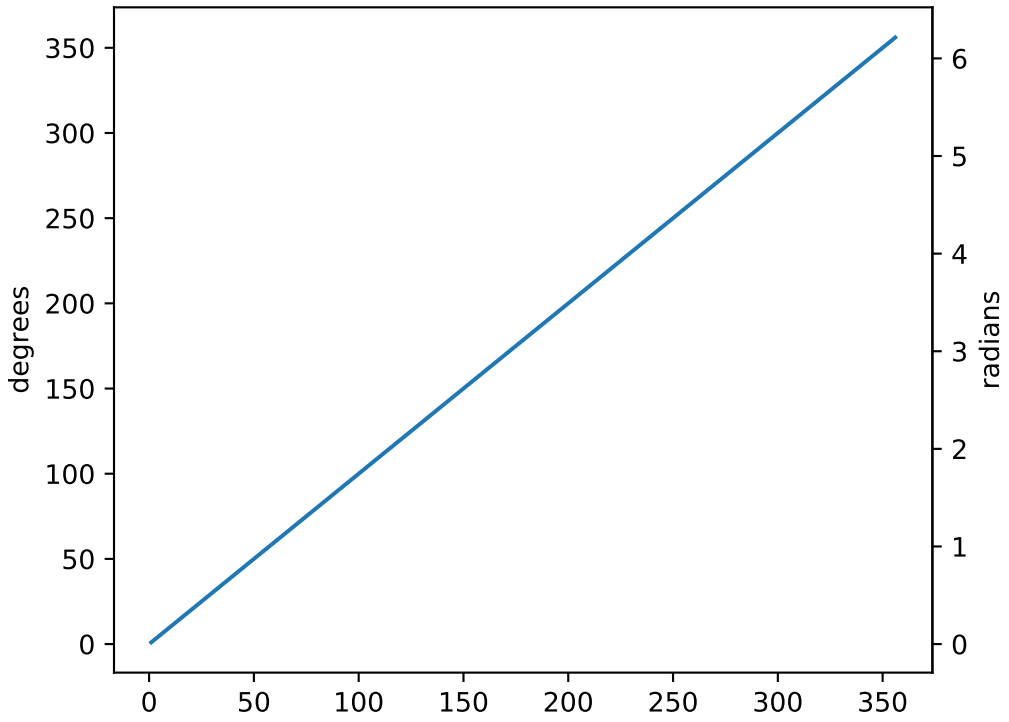
****kwargs**

[*Axes* properties.] Other miscellaneous axes parameters.

Warning: This method is experimental as of 3.1, and the API may change.

Examples

Add a secondary axes that converts from radians to degrees



Examples using `matplotlib.axes.Axes.secondary_yaxis`

- sphx_glr_gallery_subplots_axes_and_figures_secondary_axis.py

Fields

Axes.barbs	Plot a 2D field of barbs.
Axes.quiver	Plot a 2D field of arrows.
Axes.quiverkey	Add a key to a quiver plot.
Axes.streamplot	Draw streamlines of a vector flow.

matplotlib.axes.Axes.barbs

Axes.**barbs**(*self*, **args*, *data=None*, ***kw*)

Plot a 2D field of barbs.

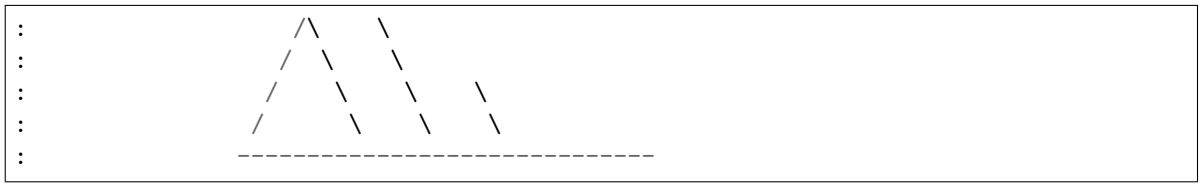
Call signature:

```
barbs([X, Y], U, V, [C], **kw)
```

Where *X*, *Y* define the barb locations, *U*, *V* define the barb directions, and *C* optionally sets the color.

All arguments may be 1D or 2D. *U*, *V*, *C* may be masked arrays, but masked *X*, *Y* are not supported at present.

Barbs are traditionally used in meteorology as a way to plot the speed and direction of wind observations, but can technically be used to plot any two dimensional vector quantity. As opposed to arrows, which give vector magnitude by the length of the arrow, the barbs give more quantitative information about the vector magnitude by putting slanted lines or a triangle for various increments in magnitude, as show schematically below:



The largest increment is given by a triangle (or "flag"). After those come full lines (barbs). The smallest increment is a half line. There is only, of course, ever at most 1 half line. If the magnitude is small and only needs a single half-line and no full lines or triangles, the half-line is offset from the end of the barb so that it can be easily distinguished from barbs with a single full line. The magnitude for the barb shown above would nominally be 65, using the standard increments of 50, 10, and 5.

See also https://en.wikipedia.org/wiki/Wind_barb.

Parameters

X, Y

[1D or 2D array-like, optional] The x and y coordinates of the barb locations. See *pivot* for how the barbs are drawn to the x, y positions.

If not given, they will be generated as a uniform integer meshgrid based on the dimensions of *U* and *V*.

If *X* and *Y* are 1D but *U*, *V* are 2D, *X*, *Y* are expanded to 2D using `X, Y = np.meshgrid(X, Y)`. In this case `len(X)` and `len(Y)` must match the column and row dimensions of *U* and *V*.

U, V

[1D or 2D array-like] The x and y components of the barb shaft.

C

[1D or 2D array-like, optional] Numeric data that defines the barb colors by colormapping via *norm* and *cmap*.

This does not support explicit colors. If you want to set colors directly, use *barbcolor* instead.

length

[float, default: 7] Length of the barb in points; the other parts of the barb are scaled against this.

pivot

[{'tip', 'middle'} or float, default: 'tip'] The part of the arrow that is anchored to the *X*, *Y* grid. The barb rotates about this point. This can also be a number, which shifts the start of the barb that many points away from grid point.

barbcolor

[color or color sequence] The color of all parts of the barb except for the flags. This parameter is analogous to the *edgecolor* parameter for polygons, which can be used instead. However this parameter will override facecolor.

flagcolor

[color or color sequence] The color of any flags on the barb. This parameter is analogous to the *facecolor* parameter for polygons, which can be used instead. However, this parameter will override facecolor. If this is not set (and *C* has not either) then *flagcolor* will be set to match *barbcolor* so that the barb has a uniform color. If *C* has been set, *flagcolor* has no effect.

sizes

[dict, optional] A dictionary of coefficients specifying the ratio of a given feature to the length of the barb. Only those values one wishes to override need to be included. These features include:

- 'spacing' - space between features (flags, full/half barbs)
- 'height' - height (distance from shaft to top) of a flag or full barb
- 'width' - width of a flag, twice the width of a full barb
- 'emptybarb' - radius of the circle used for low magnitudes

fill_empty

[bool, default: False] Whether the empty barbs (circles) that are drawn should be filled with the flag color. If they are not filled, the center is transparent.

rounding

[bool, default: True] Whether the vector magnitude should be rounded when allocating barb components. If True, the magnitude is rounded to the nearest multiple of the half-barb increment. If False, the magnitude is simply truncated to the next lowest multiple.

barb_increments

[dict, optional] A dictionary of increments specifying values to associate with different parts of the barb. Only those values one wishes to override need to be included.

- 'half' - half barbs (Default is 5)
- 'full' - full barbs (Default is 10)
- 'flag' - flags (default is 50)

flip_barb

[bool or array-like of bool, default: False] Whether the lines and flags should point opposite to normal. Normal behavior is for the barbs and lines to point right (comes from wind barbs having these features point towards low pressure in the Northern Hemisphere).

A single value is applied to all barbs. Individual barbs can be flipped by passing a bool array of the same size as *U* and *V*.

Returns

barbs

[*Barbs*]

Other Parameters

****kwargs**

The barbs can further be customized using *PolyCollection* keyword arguments:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a
alpha	float or None
animated	bool

Table 92 – continued from previous page

Property	Description
antialiased or aa or antialiaseds	bool or list of bools
array	ndarray
capstyle	{'butt', 'round', 'projecting'}
clim	(vmin: float, vmax: float)
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
cmap	*Colormap* or str or None
color	color or list of rgba tuples
contains	unknown
edgecolor or ec or edgecolors	color or list of colors or 'face'
facecolor or facecolors or fc	color or list of colors
figure	*Figure*
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or dashes or linestyles or ls	str or tuple or list thereof
linewidth or linewidths or lw	float or list of floats
norm	*Normalize* or None
offset_position	unknown
offsets	array-like (N, 2) or (2,)
path_effects	*AbstractPathEffect*
picker	None or bool or callable
pickradius	unknown
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
urls	list of str or None
visible	bool
zorder	float

Notes

Note: In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, every other argument can also be string `s`, which is interpreted as `data[s]` (unless this raises an exception).

Objects passed as **data** must support item access (`data[s]`) and membership test (`s in data`).

Examples using `matplotlib.axes.Axes.barbs`

- sphx_glr_gallery_images_contours_and_fields_barb_demo.py

matplotlib.axes.Axes.quiver

`Axes.quiver`(*self*, **args*, *data=None*, ***kw*)

Plot a 2D field of arrows.

Call signature:

```
quiver([X, Y], U, V, [C], **kw)
```

X, *Y* define the arrow locations, *U*, *V* define the arrow directions, and *C* optionally sets the color.

Arrow size

The default settings auto-scales the length of the arrows to a reasonable size. To change this behavior see the *scale* and *scale_units* parameters.

Arrow shape

The defaults give a slightly swept-back arrow; to make the head a triangle, make *headaxislength* the same as *headlength*. To make the arrow more pointed, reduce *headwidth* or increase *headlength* and *headaxislength*. To make the head smaller relative to the shaft, scale down all the head parameters. You will probably do best to leave minshaft alone.

Arrow outline

linewidths and *edgecolors* can be used to customize the arrow outlines.

Parameters

X, Y

[1D or 2D array-like, optional] The x and y coordinates of the arrow locations.

If not given, they will be generated as a uniform integer meshgrid based on the dimensions of *U* and *V*.

If *X* and *Y* are 1D but *U*, *V* are 2D, *X*, *Y* are expanded to 2D using `X, Y = np.meshgrid(X, Y)`. In this case `len(X)` and `len(Y)` must match the column and row dimensions of *U* and *V*.

U, V

[1D or 2D array-like] The x and y direction components of the arrow vectors.

They must have the same number of elements, matching the number of arrow locations. *U* and *V* may be masked. Only locations unmasked in *U*, *V*, and *C* will be drawn.

C

[1D or 2D array-like, optional] Numeric data that defines the arrow colors by colormapping via *norm* and *cmap*.

This does not support explicit colors. If you want to set colors directly, use *color* instead. The size of *C* must match the number of arrow locations.

units

[{'width', 'height', 'dots', 'inches', 'x', 'y' 'xy'}, default: 'width'] The arrow dimensions (except for *length*) are measured in multiples of this unit.

The following values are supported:

- 'width', 'height': The width or height of the axis.
- 'dots', 'inches': Pixels or inches based on the figure dpi.
- 'x', 'y', 'xy': *X*, *Y* or $\sqrt{X^2 + Y^2}$ in data units.

The arrows scale differently depending on the units. For 'x' or 'y', the arrows get larger as one zooms in; for other units, the arrow size is independent of the zoom state. For 'width or 'height', the arrow size increases with the width and height of the axes, respectively, when the window is resized; for 'dots' or 'inches', resizing does not change the arrows.

angles

[{'uv', 'xy'} or array-like, default: 'uv'] Method for determining the angle of the arrows.

- 'uv': The arrow axis aspect ratio is 1 so that if *U* == *V* the orientation of the arrow on the plot is 45 degrees counter-clockwise from the horizontal axis (positive to the right).

  Use this if the arrows symbolize a quantity that is not based on *X*, *Y* data coordinates.

- 'xy': Arrows point from (x, y) to (x+u, y+v). Use this for plotting a gradient field, for example.
- Alternatively, arbitrary angles may be specified explicitly as an array of values in degrees, counter-clockwise from the horizontal axis.

  In this case *U*, *V* is only used to determine the length of the arrows.

Note: inverting a data axis will correspondingly invert the arrows only with `angles='xy'`.

scale

[float, optional] Number of data units per arrow length unit, e.g., m/s per plot width; a smaller scale parameter makes the arrow longer. Default is *None*.

If *None*, a simple autoscaling algorithm is used, based on the average vector length and the number of vectors. The arrow length unit is given by the *scale_units* parameter.

scale_units

[{'width', 'height', 'dots', 'inches', 'x', 'y', 'xy'}, optional] If the *scale* kwarg is *None*, the arrow length unit. Default is *None*.

e.g. *scale_units* is 'inches', *scale* is 2.0, and `(u, v) = (1, 0)`, then the vector will be 0.5 inches long.

If *scale_units* is 'width' or 'height', then the vector will be half the width/height of the axes.

If *scale_units* is 'x' then the vector will be 0.5 x-axis units. To plot vectors in the x-y plane, with u and v having the same units as x and y, use `angles='xy', scale_units='xy', scale=1`.

width

[float, optional] Shaft width in arrow units; default depends on choice of units, above, and number of vectors; a typical starting value is about 0.005 times the width of the plot.

headwidth

[float, default: 3] Head width as multiple of shaft width.

headlength

[float, default: 5] Head length as multiple of shaft width.

headaxislength

[float, default: 4.5] Head length at shaft intersection.

minshaft

[float, default: 1] Length below which arrow scales, in units of head length. Do not set this to less than 1, or small arrows will look terrible!

minlength

[float, default: 1] Minimum length as a multiple of shaft width; if an arrow length is less than this, plot a dot (hexagon) of this diameter instead.

pivot

[{'tail', 'mid', 'middle', 'tip'}, default: 'tail'] The part of the arrow that is anchored to the *X*, *Y* grid. The arrow rotates about this point.

'mid' is a synonym for 'middle'.

color

[color or color sequence, optional] Explicit color(s) for the arrows. If *C* has been set, *color* has no effect.

This is a synonym for the *PolyCollection* *facecolor* parameter.

Other Parameters

****kwargs**

[*PolyCollection* properties, optional] All other keyword arguments are passed on to *PolyCollection*:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a
alpha	float or None
animated	bool
antialiased or aa or antialiaseds	bool or list of bools
array	ndarray
capstyle	{'butt', 'round', 'projecting'}
clim	(vmin: float, vmax: float)
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
cmap	*Colormap* or str or None
color	color or list of rgba tuples
contains	unknown
edgecolor or ec or edgecolors	color or list of colors or 'face'
facecolor or facecolors or fc	color or list of colors
figure	*Figure*
gid	str

Table 93 – continued from previous page

Property	Description
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or dashes or linestyles or ls	str or tuple or list thereof
linewidth or linewidths or lw	float or list of floats
norm	*Normalize* or None
offset_position	unknown
offsets	array-like (N, 2) or (2,)
path_effects	*AbstractPathEffect*
picker	None or bool or callable
pickradius	unknown
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
urls	list of str or None
visible	bool
zorder	float

See also:

Axes.quiverkey

Add a key to a quiver plot.

Examples using `matplotlib.axes.Axes.quiver`

- sphx_glr_gallery_images_contours_and_fields_quiver_demo.py
- sphx_glr_gallery_images_contours_and_fields_quiver_simple_demo.py
- sphx_glr_gallery_images_contours_and_fields_trigradient_demo.py
- sphx_glr_gallery_mplot3d_quiver3d.py

matplotlib.axes.Axes.quiverkey

Axes.quiverkey(*self*, *Q*, *X*, *Y*, *U*, *label*, ***kw*)

Add a key to a quiver plot.

The positioning of the key depends on *X*, *Y*, *coordinates*, and *labelpos*. If *labelpos* is 'N' or 'S', *X*, *Y* give the position of the middle of the key arrow. If *labelpos* is 'E', *X*, *Y* positions the head, and if *labelpos* is 'W', *X*, *Y* positions the tail; in either of these two cases, *X*, *Y* is somewhere in the middle of the arrow+label key object.

Parameters

Q

[*matplotlib.quiver.Quiver*] A *Quiver* object as returned by a call to *quiver()*.

X, Y

[float] The location of the key.

U

[float] The length of the key.

label

[str] The key label (e.g., length and units of the key).

angle

[float, default: 0] The angle of the key arrow, in degrees anti-clockwise from the x-axis.

coordinates

[{'axes', 'figure', 'data', 'inches'}, default: 'axes'] Coordinate system and units for *X*, *Y*: 'axes' and 'figure' are normalized coordinate systems with (0, 0) in the lower left and (1, 1) in the upper right; 'data' are the axes data coordinates (used for the locations of the vectors in the quiver plot itself); 'inches' is position in the figure in inches, with (0, 0) at the lower left corner.

color

[color] Overrides face and edge colors from *Q*.

labelpos

[{'N', 'S', 'E', 'W'}] Position the label above, below, to the right, to the left of the arrow, respectively.

labelsep

[float, default: 0.1] Distance in inches between the arrow and the label.

labelcolor

[color, default: `rcParams["text.color"]` (default: `'black'`)] Label color.

fontproperties

[dict, optional] A dictionary with keyword arguments accepted by the *FontProperties* initializer: *family*, *style*, *variant*, *size*, *weight*.

****kwargs**

Any additional keyword arguments are used to override vector properties taken from *Q*.

Examples using `matplotlib.axes.Axes.quiverkey`

- sphx_glr_gallery_images_contours_and_fields_quiver_demo.py
- sphx_glr_gallery_images_contours_and_fields_quiver_simple_demo.py

matplotlib.axes.Axes.streamplot

`Axes.streamplot`(*axes*, *x*, *y*, *u*, *v*, *density=1*, *linewidth=None*, *color=None*, *cmap=None*, *norm=None*, *arrowsize=1*, *arrowstyle='-|>'*, *minlength=0.1*, *transform=None*, *zorder=None*, *start_points=None*, *maxlength=4.0*, *integration_direction='both'*, ***, *data=None*)

Draw streamlines of a vector flow.

Parameters

x, y

[1D arrays] An evenly spaced grid.

u, v

[2D arrays] *x* and *y*-velocities. The number of rows and columns must match the length of *y* and *x*, respectively.

density

[float or (float, float)] Controls the closeness of streamlines. When `density = 1`, the domain is divided into a 30x30 grid. *density* linearly scales this grid. Each cell in the grid can have, at most, one traversing streamline. For different densities in each direction, use a tuple (density_x, density_y).

linewidth

[float or 2D array] The width of the stream lines. With a 2D array the line width can be varied across the grid. The array must have the same shape as *u* and *v*.

color

[color or 2D array] The streamline color. If given an array, its values are converted to colors using *cmap* and *norm*. The array must have the same shape as *u* and *v*.

cmap

[*Colormap*] Colormap used to plot streamlines and arrows. This is only used if *color* is an array.

norm

[*Normalize*] Normalize object used to scale luminance data to 0, 1. If `None`, stretch (min, max) to (0, 1). This is only used if *color* is an array.

arrowsize

[float] Scaling factor for the arrow size.

arrowstyle

[str] Arrow style specification. See *FancyArrowPatch*.

minlength

[float] Minimum length of streamline in axes coordinates.

start_points

[Nx2 array] Coordinates of starting points for the streamlines in data coordinates (the same coordinates as the *x* and *y* arrays).

zorder

[int] The zorder of the stream lines and arrows. Artists with lower zorder values are drawn first.

maxlength

[float] Maximum length of streamline in axes coordinates.

integration_direction

[{'forward', 'backward', 'both'}, default: 'both'] Integrate the streamline in forward, backward or both directions.

Returns

StreamplotSet

Container object with attributes

- `lines`: *LineCollection* of streamlines
- `arrows`: *PatchCollection* containing *FancyArrowPatch* objects representing the arrows half-way along stream lines.

This container will probably change in the future to allow changes to the colormap, alpha, etc. for both lines and arrows, but these changes should be backward compatible.

Notes

Note: In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, the following arguments can also be string `s`, which is interpreted as `data[s]` (unless this raises an exception): *x*, *y*, *u*, *v*, *start_points*.

Objects passed as **data** must support item access (`data[s]`) and membership test (`s in data`).

Examples using `matplotlib.axes.Axes.streamplot`

- sphx_glr_gallery_images_contours_and_fields_plot_streamplot.py

18.5.5 Clearing

Axes.cla	Clear the current axes.
Axes.clear	Clear the axes.

matplotlib.axes.Axes.cla

`Axes.cla(`*self*`)`
Clear the current axes.

Examples using `matplotlib.axes.Axes.cla`

- sphx_glr_gallery_animation_animation_demo.py
- sphx_glr_gallery_event_handling_data_browser.py
- sphx_glr_gallery_misc_custom_projection.py

matplotlib.axes.Axes.clear

Axes.clear(*self*)
Clear the axes.

Examples using `matplotlib.axes.Axes.clear`

18.5.6 Appearance

Axes.axis	Convenience method to get or set some axis properties.
Axes.set_axis_off	Turn the x- and y-axis off.
Axes.set_axis_on	Turn the x- and y-axis on.
Axes.set_frame_on	Set whether the axes rectangle patch is drawn.
Axes.get_frame_on	Get whether the axes rectangle patch is drawn.
Axes.set_axisbelow	Set whether axis ticks and gridlines are above or below most artists.
Axes.get_axisbelow	Get whether axis ticks and gridlines are above or below most artists.
Axes.grid	Configure the grid lines.
Axes.get_facecolor	Get the facecolor of the Axes.
Axes.set_facecolor	Set the facecolor of the Axes.

matplotlib.axes.Axes.axis

Axes.axis(*self*, **args*, *emit=True*, ***kwargs*)
Convenience method to get or set some axis properties.

Call signatures:

```
xmin, xmax, ymin, ymax = axis()
xmin, xmax, ymin, ymax = axis([xmin, xmax, ymin, ymax])
xmin, xmax, ymin, ymax = axis(option)
xmin, xmax, ymin, ymax = axis(**kwargs)
```

Parameters

xmin, xmax, ymin, ymax

[float, optional] The axis limits to be set. This can also be achieved using

```
ax.set(xlim=(xmin, xmax), ylim=(ymin, ymax))
```

option

[bool or str] If a bool, turns axis lines and labels on or off. If a string, possible values are:

Value	Description
'on'	Turn on axis lines and labels. Same as `True`.
'off'	Turn off axis lines and labels. Same as `False`.
'equal'	Set equal scaling (i.e., make circles circular) by changing axis limits. This is the same as `ax.set_aspect('equal', adjustable='datalim')`. Explicit data limits may not be respected in this case.
'scaled'	Set equal scaling (i.e., make circles circular) by changing dimensions of the plot box. This is the same as `ax.set_aspect('equal', adjustable='box', anchor='C')`. Additionally, further autoscaling will be disabled.
'tight'	Set limits just large enough to show all data, then disable further autoscaling.
'auto'	Automatic scaling (fill plot box with data).
'image'	'scaled' with axis limits equal to data limits.
'square'	Square plot; similar to 'scaled', but initially forcing `xmax-xmin == ymax-ymin`.

emit

[bool, default: True] Whether observers are notified of the axis limit change. This option is passed on to *set_xlim* and *set_ylim*.

Returns

xmin, xmax, ymin, ymax

[float] The axis limits.

See also:

matplotlib.axes.Axes.set_xlim

matplotlib.axes.Axes.set_ylim

Examples using `matplotlib.axes.Axes.axis`

- sphx_glr_gallery_lines_bars_and_markers_fill.py
- sphx_glr_gallery_images_contours_and_fields_image_clip_path.py
- sphx_glr_gallery_images_contours_and_fields_image_demo.py
- sphx_glr_gallery_pie_and_polar_charts_pie_features.py
- sphx_glr_gallery_pie_and_polar_charts_bar_of_pie.py
- sphx_glr_gallery_shapes_and_collections_path_patch.py
- sphx_glr_gallery_style_sheets_ggplot.py
- sphx_glr_gallery_axes_grid1_parasite_simple2.py
- sphx_glr_gallery_axes_grid1_simple_axisline4.py
- sphx_glr_gallery_axisartist_axis_direction_demo_step01.py
- sphx_glr_gallery_axisartist_axis_direction_demo_step02.py
- sphx_glr_gallery_axisartist_axis_direction_demo_step03.py
- sphx_glr_gallery_axisartist_axis_direction_demo_step04.py
- sphx_glr_gallery_axisartist_demo_axis_direction.py
- sphx_glr_gallery_axisartist_demo_axisline_style.py
- sphx_glr_gallery_axisartist_demo_floating_axes.py
- sphx_glr_gallery_axisartist_demo_parasite_axes.py
- sphx_glr_gallery_axisartist_demo_parasite_axes2.py
- sphx_glr_gallery_axisartist_demo_ticklabel_alignment.py
- sphx_glr_gallery_axisartist_demo_ticklabel_direction.py
- sphx_glr_gallery_axisartist_simple_axis_direction01.py
- sphx_glr_gallery_axisartist_simple_axis_direction03.py
- sphx_glr_gallery_axisartist_simple_axis_pad.py
- sphx_glr_gallery_axisartist_simple_axisartist1.py
- sphx_glr_gallery_axisartist_simple_axisline.py
- sphx_glr_gallery_axisartist_simple_axisline2.py
- sphx_glr_gallery_axisartist_simple_axisline3.py
- sphx_glr_gallery_specialty_plots_mri_demo.py
- sphx_glr_gallery_specialty_plots_mri_with_eeg.py
- *Specifying Colors*

- *Text in Matplotlib Plots*

matplotlib.axes.Axes.set_axis_off

Axes.**set_axis_off**(*self*)

Turn the x- and y-axis off.

This affects the axis lines, ticks, ticklabels, grid and axis labels.

Examples using `matplotlib.axes.Axes.set_axis_off`

- sphx_glr_gallery_lines_bars_and_markers_joinstyle.py
- sphx_glr_gallery_lines_bars_and_markers_marker_reference.py
- sphx_glr_gallery_images_contours_and_fields_barcode_demo.py
- sphx_glr_gallery_images_contours_and_fields_image_transparency_blend.py
- sphx_glr_gallery_pie_and_polar_charts_nested_pie.py
- sphx_glr_gallery_color_colormap_reference.py
- *Choosing Colormaps in Matplotlib*
- *Text properties and layout*

matplotlib.axes.Axes.set_axis_on

Axes.**set_axis_on**(*self*)

Turn the x- and y-axis on.

This affects the axis lines, ticks, ticklabels, grid and axis labels.

Examples using `matplotlib.axes.Axes.set_axis_on`

matplotlib.axes.Axes.set_frame_on

Axes.**set_frame_on**(*self*, *b*)

Set whether the axes rectangle patch is drawn.

Parameters

b

[bool]

Examples using `matplotlib.axes.Axes.set_frame_on`

matplotlib.axes.Axes.get_frame_on

`Axes.get_frame_on`(*self*)
Get whether the axes rectangle patch is drawn.

Examples using `matplotlib.axes.Axes.get_frame_on`

matplotlib.axes.Axes.set_axisbelow

`Axes.set_axisbelow`(*self*, *b*)
Set whether axis ticks and gridlines are above or below most artists.

This controls the zorder of the ticks and gridlines. For more information on the zorder see /gallery/misc/zorder_demo.

Parameters

b

[bool or 'line'] Possible values:

- *True* (zorder = 0.5): Ticks and gridlines are below all Artists.
- 'line' (zorder = 1.5): Ticks and gridlines are above patches (e.g. rectangles, with default zorder = 1) but still below lines and markers (with their default zorder = 2).
- *False* (zorder = 2.5): Ticks and gridlines are above patches and lines / markers.

See also:

get_axisbelow

Examples using `matplotlib.axes.Axes.set_axisbelow`

- sphx_glr_gallery_statistics_boxplot_demo.py

matplotlib.axes.Axes.get_axisbelow

Axes.get_axisbelow(*self*)

Get whether axis ticks and gridlines are above or below most artists.

Returns

bool or 'line'

See also:

set_axisbelow

Examples using matplotlib.axes.Axes.get_axisbelow

matplotlib.axes.Axes.grid

Axes.grid(*self*, *b=None*, *which='major'*, *axis='both'*, ***kwargs*)

Configure the grid lines.

Parameters

b

[bool or None, optional] Whether to show the grid lines. If any *kwargs* are supplied, it is assumed you want the grid on and *b* will be set to True.

If *b* is *None* and there are no *kwargs*, this toggles the visibility of the lines.

which

[{'major', 'minor', 'both'}, optional] The grid lines to apply the changes on.

axis

[{'both', 'x', 'y'}, optional] The axis to apply the changes on.

****kwargs**

[*Line2D* properties] Define the line properties of the grid, e.g.:

```
grid(color='r', linestyle='-', linewidth=2)
```

Valid keyword arguments are:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value

Table 96 – continued from previous page

Property	Description
alpha	float or None
animated	bool
antialiased or aa	bool
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color or c	color
contains	unknown
dash_capstyle	{'butt', 'round', 'projecting'}
dash_joinstyle	{'miter', 'round', 'bevel'}
dashes	sequence of floats (on/off ink in points) or (None, None)
data	(2, N) array or two 1D arrays
drawstyle or ds	{'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'd
figure	*Figure*
fillstyle	{'full', 'left', 'right', 'bottom', 'top', 'none'}
gid	str
in_layout	bool
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float
marker	marker style string, *Path* or *MarkerStyle*
markeredgecolor or mec	color
markeredgewidth or mew	float
markerfacecolor or mfc	color
markerfacecoloralt or mfcalt	color
markersize or ms	float
markevery	None or int or (int, int) or slice or List[int] or float or (float, float)
path_effects	*AbstractPathEffect*
picker	unknown
pickradius	float
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
solid_capstyle	{'butt', 'round', 'projecting'}
solid_joinstyle	{'miter', 'round', 'bevel'}
transform	*matplotlib.transforms.Transform*
url	str
visible	bool
xdata	1D array
ydata	1D array
zorder	float

Notes

The axis is drawn as a unit, so the effective zorder for drawing the grid is determined by the zorder of each axis, not by the zorder of the *Line2D* objects comprising the grid. Therefore, to set grid zorder, use *set_axisbelow* or, for more control, call the *set_zorder* method of each axis.

Examples using `matplotlib.axes.Axes.grid`

- sphx_glr_gallery_lines_bars_and_markers_broken_barh.py
- sphx_glr_gallery_lines_bars_and_markers_csd_demo.py
- sphx_glr_gallery_lines_bars_and_markers_psd_demo.py
- sphx_glr_gallery_lines_bars_and_markers_scatter_demo2.py
- sphx_glr_gallery_lines_bars_and_markers_scatter_with_legend.py
- sphx_glr_gallery_lines_bars_and_markers_simple_plot.py
- sphx_glr_gallery_lines_bars_and_markers_xcorr_acorr_demo.py
- sphx_glr_gallery_images_contours_and_fields_contour_corner_mask.py
- sphx_glr_gallery_images_contours_and_fields_image_annotated_heatmap.py
- sphx_glr_gallery_images_contours_and_fields_image_demo.py
- sphx_glr_gallery_images_contours_and_fields_watermark_image.py
- sphx_glr_gallery_subplots_axes_and_figures_axes_props.py
- sphx_glr_gallery_subplots_axes_and_figures_invert_axes.py
- sphx_glr_gallery_statistics_histogram_cumulative.py
- sphx_glr_gallery_pie_and_polar_charts_polar_demo.py
- sphx_glr_gallery_text_labels_and_annotations_date.py
- sphx_glr_gallery_text_labels_and_annotations_watermark_text.py
- sphx_glr_gallery_shapes_and_collections_path_patch.py
- sphx_glr_gallery_showcase_anatomy.py
- sphx_glr_gallery_showcase_bachelors_degrees_by_gender.py
- *Decay*
- *The double pendulum problem*
- sphx_glr_gallery_misc_patheffect_demo.py
- sphx_glr_gallery_misc_pythonic_matplotlib.py
- sphx_glr_gallery_mplot3d_mixed_subplots.py

- sphx_glr_gallery_recipes_fill_between_alpha.py
- sphx_glr_gallery_scales_log_demo.py
- sphx_glr_gallery_scales_log_test.py
- sphx_glr_gallery_scales_scales.py
- sphx_glr_gallery_units_artist_tests.py

matplotlib.axes.Axes.get_facecolor

Axes.**get_facecolor**(*self*)
Get the facecolor of the Axes.

Examples using `matplotlib.axes.Axes.get_facecolor`

matplotlib.axes.Axes.set_facecolor

Axes.**set_facecolor**(*self*, *color*)
Set the facecolor of the Axes.

Parameters

color

[color]

Examples using `matplotlib.axes.Axes.set_facecolor`

- sphx_glr_gallery_color_color_demo.py
- sphx_glr_gallery_color_color_cycle_default.py

18.5.7 Property cycle

Axes.set_prop_cycle	Set the property cycle of the Axes.

matplotlib.axes.Axes.set_prop_cycle

Axes.set_prop_cycle(*self*, **args*, ***kwargs*)

Set the property cycle of the Axes.

The property cycle controls the style properties such as color, marker and linestyle of future plot commands. The style properties of data already added to the Axes are not modified.

Call signatures:

```
set_prop_cycle(cycler)
set_prop_cycle(label=values[, label2=values2[, ...]])
set_prop_cycle(label, values)
```

Form 1 sets given `Cycler` object.

Form 2 creates a `Cycler` which cycles over one or more properties simultaneously and set it as the property cycle of the axes. If multiple properties are given, their value lists must have the same length. This is just a shortcut for explicitly creating a cycler and passing it to the function, i.e. it's short for `set_prop_cycle(cycler(label=values label2=values2, ...))`.

Form 3 creates a `Cycler` for a single property and set it as the property cycle of the axes. This form exists for compatibility with the original `cycler.cycler` interface. Its use is discouraged in favor of the kwarg form, i.e. `set_prop_cycle(label=values)`.

Parameters

cycler

[Cycler] Set the given Cycler. *None* resets to the cycle defined by the current style.

label

[str] The property key. Must be a valid *Artist* property. For example, 'color' or 'linestyle'. Aliases are allowed, such as 'c' for 'color' and 'lw' for 'linewidth'.

values

[iterable] Finite-length iterable of the property values. These values are validated and will raise a ValueError if invalid.

See also:

matplotlib.rcsetup.cycler

Convenience function for creating validated cyclers for properties.

cycler.cycler

The original function for creating unvalidated cyclers.

Examples

Setting the property cycle for a single property:

```
>>> ax.set_prop_cycle(color=['red', 'green', 'blue'])
```

Setting the property cycle for simultaneously cycling over multiple properties (e.g. red circle, green plus, blue cross):

```
>>> ax.set_prop_cycle(color=['red', 'green', 'blue'],
...                   marker=['o', '+', 'x'])
```

Examples using `matplotlib.axes.Axes.set_prop_cycle`

- sphx_glr_gallery_showcase_bachelors_degrees_by_gender.py
- *Styling with cycler*

18.5.8 Axis / limits

Axes.get_xaxis	Return the XAxis instance.
Axes.get_yaxis	Return the YAxis instance.

matplotlib.axes.Axes.get_xaxis

Axes.**get_xaxis**(*self*)
Return the XAxis instance.

Examples using `matplotlib.axes.Axes.get_xaxis`

- sphx_glr_gallery_lines_bars_and_markers_timeline.py
- sphx_glr_gallery_statistics_customized_violin.py
- sphx_glr_gallery_showcase_bachelors_degrees_by_gender.py

matplotlib.axes.Axes.get_yaxis

Axes.get_yaxis(*self*)
Return the YAxis instance.

Examples using matplotlib.axes.Axes.get_yaxis

- sphx_glr_gallery_lines_bars_and_markers_timeline.py
- sphx_glr_gallery_showcase_bachelors_degrees_by_gender.py

Axis Limits and direction

Axes.invert_xaxis	Invert the x-axis.
Axes.xaxis_inverted	Return whether the xaxis is oriented in the "inverse" direction.
Axes.invert_yaxis	Invert the y-axis.
Axes.yaxis_inverted	Return whether the yaxis is oriented in the "inverse" direction.
Axes.set_xlim	Set the x-axis view limits.
Axes.get_xlim	Return the x-axis view limits.
Axes.set_ylim	Set the y-axis view limits.
Axes.get_ylim	Return the y-axis view limits.
Axes.update_datalim	Extend the dataLim Bbox to include the given points.
Axes.update_datalim_bounds	[*Deprecated*] Extend the datalim Bbox to include the given *Bbox*.
Axes.set_xbound	Set the lower and upper numerical bounds of the x-axis.
Axes.get_xbound	Return the lower and upper x-axis bounds, in increasing order.
Axes.set_ybound	Set the lower and upper numerical bounds of the y-axis.
Axes.get_ybound	Return the lower and upper y-axis bounds, in increasing order.

matplotlib.axes.Axes.invert_xaxis

Axes.**invert_xaxis**(*self*)

Invert the x-axis.

See also:

xaxis_inverted

get_xlim, *set_xlim*

get_xbound, *set_xbound*

Examples using matplotlib.axes.Axes.invert_xaxis

matplotlib.axes.Axes.xaxis_inverted

Axes.**xaxis_inverted**(*self*)

Return whether the xaxis is oriented in the "inverse" direction.

The "normal" direction is increasing to the right for the x-axis and to the top for the y-axis; the "inverse" direction is increasing to the left for the x-axis and to the bottom for the y-axis.

Examples using matplotlib.axes.Axes.xaxis_inverted

matplotlib.axes.Axes.invert_yaxis

Axes.**invert_yaxis**(*self*)

Invert the y-axis.

See also:

yaxis_inverted

get_ylim, *set_ylim*

get_ybound, *set_ybound*

Examples using `matplotlib.axes.Axes.invert_yaxis`

- sphx_glr_gallery_lines_bars_and_markers_barh.py
- sphx_glr_gallery_lines_bars_and_markers_marker_reference.py

matplotlib.axes.Axes.yaxis_inverted

Axes.**yaxis_inverted**(*self*)

Return whether the yaxis is oriented in the "inverse" direction.

The "normal" direction is increasing to the right for the x-axis and to the top for the y-axis; the "inverse" direction is increasing to the left for the x-axis and to the bottom for the y-axis.

Examples using `matplotlib.axes.Axes.yaxis_inverted`

matplotlib.axes.Axes.set_xlim

Axes.**set_xlim**(*self*, *left=None*, *right=None*, *emit=True*, *auto=False*, ***, *xmin=None*, *xmax=None*)

Set the x-axis view limits.

Parameters

left

[float, optional] The left xlim in data coordinates. Passing *None* leaves the limit unchanged.

The left and right xlims may also be passed as the tuple (*left*, *right*) as the first positional argument (or as the *left* keyword argument).

right

[float, optional] The right xlim in data coordinates. Passing *None* leaves the limit unchanged.

emit

[bool, default: True] Whether to notify observers of limit change.

auto

[bool or None, default: False] Whether to turn on autoscaling of the x-axis. True turns on, False turns off, None leaves unchanged.

xmin, xmax

[float, optional] They are equivalent to left and right respectively, and it is an error to pass both *xmin* and *left* or *xmax* and *right*.

Returns

left, right

[(float, float)] The new x-axis limits in data coordinates.

See also:

get_xlim

set_xbound, *get_xbound*

invert_xaxis, *xaxis_inverted*

Notes

The *left* value may be greater than the *right* value, in which case the x-axis values will decrease from left to right.

Examples

```
>>> set_xlim(left, right)
>>> set_xlim((left, right))
>>> left, right = set_xlim(left, right)
```

One limit may be left unchanged.

```
>>> set_xlim(right=right_lim)
```

Limits may be passed in reverse order to flip the direction of the x-axis. For example, suppose *x* represents the number of years before present. The x-axis limits might be set like the following so 5000 years ago is on the left of the plot and the present is on the right.

```
>>> set_xlim(5000, 0)
```

Examples using `matplotlib.axes.Axes.set_xlim`

- sphx_glr_gallery_lines_bars_and_markers_broken_barh.py
- sphx_glr_gallery_lines_bars_and_markers_csd_demo.py
- sphx_glr_gallery_lines_bars_and_markers_eventcollection_demo.py
- sphx_glr_gallery_lines_bars_and_markers_joinstyle.py
- sphx_glr_gallery_lines_bars_and_markers_markevery_demo.py
- sphx_glr_gallery_images_contours_and_fields_image_nonuniform.py
- sphx_glr_gallery_images_contours_and_fields_pcolormesh_grids.py
- sphx_glr_gallery_subplots_axes_and_figures_axes_box_aspect.py
- sphx_glr_gallery_subplots_axes_and_figures_axes_demo.py
- sphx_glr_gallery_subplots_axes_and_figures_axes_zoom_effect.py
- sphx_glr_gallery_subplots_axes_and_figures_invert_axes.py
- sphx_glr_gallery_subplots_axes_and_figures_zoom_inset_axes.py
- sphx_glr_gallery_statistics_boxplot_demo.py
- sphx_glr_gallery_statistics_customized_violin.py
- sphx_glr_gallery_statistics_errorbar_limits.py
- sphx_glr_gallery_pie_and_polar_charts_bar_of_pie.py
- sphx_glr_gallery_text_labels_and_annotations_date.py
- sphx_glr_gallery_text_labels_and_annotations_demo_annotation_box.py
- sphx_glr_gallery_text_labels_and_annotations_demo_text_path.py
- sphx_glr_gallery_text_labels_and_annotations_fancyarrow_demo.py
- sphx_glr_gallery_text_labels_and_annotations_text_rotation_relative_to_line.py
- sphx_glr_gallery_text_labels_and_annotations_usetex_baseline_test.py
- sphx_glr_gallery_pyplots_annotate_transform.py
- sphx_glr_gallery_pyplots_whats_new_99_axes_grid.py
- sphx_glr_gallery_shapes_and_collections_donut.py
- sphx_glr_gallery_shapes_and_collections_ellipse_demo.py
- sphx_glr_gallery_shapes_and_collections_hatch_demo.py
- sphx_glr_gallery_shapes_and_collections_line_collection.py
- sphx_glr_gallery_axes_grid1_inset_locator_demo2.py
- sphx_glr_gallery_axes_grid1_parasite_simple2.py

- sphx_glr_gallery_axisartist_demo_axis_direction.py
- sphx_glr_gallery_axisartist_demo_parasite_axes.py
- sphx_glr_gallery_axisartist_demo_parasite_axes2.py
- sphx_glr_gallery_axisartist_simple_axis_pad.py
- sphx_glr_gallery_showcase_anatomy.py
- sphx_glr_gallery_showcase_bachelors_degrees_by_gender.py
- sphx_glr_gallery_showcase_firefox.py
- sphx_glr_gallery_showcase_xkcd.py
- *Decay*
- *Animated histogram*
- *Rain simulation*
- sphx_glr_gallery_event_handling_path_editor.py
- sphx_glr_gallery_event_handling_poly_editor.py
- sphx_glr_gallery_event_handling_resample.py
- sphx_glr_gallery_event_handling_zoom_window.py
- sphx_glr_gallery_frontpage_contour.py
- sphx_glr_gallery_frontpage_membrane.py
- sphx_glr_gallery_misc_custom_projection.py
- sphx_glr_gallery_misc_histogram_path.py
- sphx_glr_gallery_misc_svg_filter_line.py
- sphx_glr_gallery_mplot3d_2dcollections3d.py
- sphx_glr_gallery_mplot3d_contour3d_3.py
- sphx_glr_gallery_mplot3d_contourf3d_2.py
- sphx_glr_gallery_mplot3d_pathpatch3d.py
- sphx_glr_gallery_mplot3d_polys3d.py
- sphx_glr_gallery_mplot3d_text3d.py
- sphx_glr_gallery_scales_aspect_loglog.py
- sphx_glr_gallery_scales_scales.py
- sphx_glr_gallery_specialty_plots_mri_with_eeg.py
- sphx_glr_gallery_specialty_plots_skewt.py
- sphx_glr_gallery_ticks_and_spines_date_concise_formatter.py
- sphx_glr_gallery_ticks_and_spines_date_demo_convert.py

- sphx_glr_gallery_ticks_and_spines_multiple_yaxis_with_spines.py
- sphx_glr_gallery_ticks_and_spines_spines_bounds.py
- sphx_glr_gallery_units_annotate_with_units.py
- sphx_glr_gallery_units_artist_tests.py
- sphx_glr_gallery_userdemo_annotate_text_arrow.py
- sphx_glr_gallery_userdemo_connect_simple01.py
- sphx_glr_gallery_widgets_cursor.py
- sphx_glr_gallery_widgets_span_selector.py
- *Path Tutorial*
- *Transformations Tutorial*
- *Specifying Colors*
- *Choosing Colormaps in Matplotlib*

matplotlib.axes.Axes.get_xlim

Axes.get_xlim(*self*)

Return the x-axis view limits.

Returns

left, right

[(float, float)] The current x-axis limits in data coordinates.

See also:

set_xlim

set_xbound, *get_xbound*

invert_xaxis, *xaxis_inverted*

Notes

The x-axis may be inverted, in which case the *left* value will be greater than the *right* value.

Examples using `matplotlib.axes.Axes.get_xlim`

- *Decay*

matplotlib.axes.Axes.set_ylim

`Axes.set_ylim`(*self*, *bottom=None*, *top=None*, *emit=True*, *auto=False*, *, *ymin=None*, *ymax=None*)

Set the y-axis view limits.

Parameters

bottom

[float, optional] The bottom ylim in data coordinates. Passing *None* leaves the limit unchanged.

The bottom and top ylims may also be passed as the tuple (*bottom*, *top*) as the first positional argument (or as the *bottom* keyword argument).

top

[float, optional] The top ylim in data coordinates. Passing *None* leaves the limit unchanged.

emit

[bool, default: True] Whether to notify observers of limit change.

auto

[bool or None, default: False] Whether to turn on autoscaling of the y-axis. *True* turns on, *False* turns off, *None* leaves unchanged.

ymin, ymax

[float, optional] They are equivalent to bottom and top respectively, and it is an error to pass both *ymin* and *bottom* or *ymax* and *top*.

Returns

bottom, top

[(float, float)] The new y-axis limits in data coordinates.

See also:

get_ylim

set_ybound, *get_ybound*

invert_yaxis, *yaxis_inverted*

Notes

The *bottom* value may be greater than the *top* value, in which case the y-axis values will decrease from *bottom* to *top*.

Examples

```
>>> set_ylim(bottom, top)
>>> set_ylim((bottom, top))
>>> bottom, top = set_ylim(bottom, top)
```

One limit may be left unchanged.

```
>>> set_ylim(top=top_lim)
```

Limits may be passed in reverse order to flip the direction of the y-axis. For example, suppose `y` represents depth of the ocean in m. The y-axis limits might be set like the following so 5000 m depth is at the bottom of the plot and the surface, 0 m, is at the top.

```
>>> set_ylim(5000, 0)
```

Examples using `matplotlib.axes.Axes.set_ylim`

- sphx_glr_gallery_lines_bars_and_markers_broken_barh.py
- sphx_glr_gallery_lines_bars_and_markers_eventcollection_demo.py
- sphx_glr_gallery_lines_bars_and_markers_joinstyle.py
- sphx_glr_gallery_lines_bars_and_markers_markevery_demo.py
- sphx_glr_gallery_lines_bars_and_markers_psd_demo.py
- sphx_glr_gallery_images_contours_and_fields_image_nonuniform.py
- sphx_glr_gallery_images_contours_and_fields_pcolormesh_grids.py
- sphx_glr_gallery_subplots_axes_and_figures_axes_demo.py
- sphx_glr_gallery_subplots_axes_and_figures_broken_axis.py
- sphx_glr_gallery_subplots_axes_and_figures_zoom_inset_axes.py
- sphx_glr_gallery_statistics_boxplot_demo.py
- sphx_glr_gallery_text_labels_and_annotations_demo_annotation_box.py
- sphx_glr_gallery_text_labels_and_annotations_demo_text_path.py

- sphx_glr_gallery_text_labels_and_annotations_fancyarrow_demo.py
- sphx_glr_gallery_text_labels_and_annotations_usetex_baseline_test.py
- sphx_glr_gallery_pyplots_align_ylabels.py
- sphx_glr_gallery_pyplots_annotate_transform.py
- sphx_glr_gallery_pyplots_annotation_basic.py
- sphx_glr_gallery_pyplots_whats_new_99_axes_grid.py
- sphx_glr_gallery_shapes_and_collections_collections.py
- sphx_glr_gallery_shapes_and_collections_donut.py
- sphx_glr_gallery_shapes_and_collections_ellipse_demo.py
- sphx_glr_gallery_shapes_and_collections_hatch_demo.py
- sphx_glr_gallery_shapes_and_collections_line_collection.py
- sphx_glr_gallery_axes_grid1_inset_locator_demo2.py
- sphx_glr_gallery_axes_grid1_parasite_simple2.py
- sphx_glr_gallery_axisartist_demo_axis_direction.py
- sphx_glr_gallery_axisartist_demo_parasite_axes.py
- sphx_glr_gallery_axisartist_demo_parasite_axes2.py
- sphx_glr_gallery_axisartist_simple_axis_pad.py
- sphx_glr_gallery_axisartist_simple_axisartist1.py
- sphx_glr_gallery_axisartist_simple_axisline.py
- sphx_glr_gallery_showcase_anatomy.py
- sphx_glr_gallery_showcase_bachelors_degrees_by_gender.py
- sphx_glr_gallery_showcase_firefox.py
- sphx_glr_gallery_showcase_integral.py
- sphx_glr_gallery_showcase_xkcd.py
- *Decay*
- *Animated histogram*
- *Rain simulation*
- *MATPLOTLIB UNCHAINED*
- sphx_glr_gallery_event_handling_data_browser.py
- sphx_glr_gallery_event_handling_path_editor.py
- sphx_glr_gallery_event_handling_pick_event_demo2.py
- sphx_glr_gallery_event_handling_poly_editor.py

- sphx_glr_gallery_event_handling_zoom_window.py
- sphx_glr_gallery_frontpage_contour.py
- sphx_glr_gallery_frontpage_membrane.py
- sphx_glr_gallery_misc_custom_projection.py
- sphx_glr_gallery_misc_histogram_path.py
- sphx_glr_gallery_misc_pythonic_matplotlib.py
- sphx_glr_gallery_misc_svg_filter_line.py
- sphx_glr_gallery_mplot3d_2dcollections3d.py
- sphx_glr_gallery_mplot3d_contour3d_3.py
- sphx_glr_gallery_mplot3d_contourf3d_2.py
- sphx_glr_gallery_mplot3d_pathpatch3d.py
- sphx_glr_gallery_mplot3d_polys3d.py
- sphx_glr_gallery_mplot3d_text3d.py
- sphx_glr_gallery_scales_aspect_loglog.py
- sphx_glr_gallery_scales_log_demo.py
- sphx_glr_gallery_specialty_plots_mri_with_eeg.py
- sphx_glr_gallery_specialty_plots_skewt.py
- sphx_glr_gallery_ticks_and_spines_multiple_yaxis_with_spines.py
- sphx_glr_gallery_ticks_and_spines_spines_bounds.py
- sphx_glr_gallery_units_annotate_with_units.py
- sphx_glr_gallery_units_artist_tests.py
- sphx_glr_gallery_userdemo_annotate_text_arrow.py
- sphx_glr_gallery_userdemo_connect_simple01.py
- sphx_glr_gallery_widgets_cursor.py
- sphx_glr_gallery_widgets_span_selector.py
- *Path Tutorial*
- *Transformations Tutorial*
- *Specifying Colors*
- *Choosing Colormaps in Matplotlib*

matplotlib.axes.Axes.get_ylim

Axes.get_ylim(*self*)

Return the y-axis view limits.

Returns

bottom, top

[(float, float)] The current y-axis limits in data coordinates.

See also:

set_ylim

set_ybound, *get_ybound*

invert_yaxis, *yaxis_inverted*

Notes

The y-axis may be inverted, in which case the *bottom* value will be greater than the *top* value.

Examples using matplotlib.axes.Axes.get_ylim

- sphx_glr_gallery_shapes_and_collections_collections.py

matplotlib.axes.Axes.update_datalim

Axes.update_datalim(*self*, *xys*, *updatex=True*, *updatey=True*)

Extend the dataLim Bbox to include the given points.

If no data is set currently, the Bbox will ignore its limits and set the bound to be the bounds of the xydata (*xys*). Otherwise, it will compute the bounds of the union of its current data and the data in *xys*.

Parameters

xys

[2D array-like] The points to include in the data limits Bbox. This can be either a list of (x, y) tuples or a Nx2 array.

updatex, updatey

[bool, default: True] Whether to update the x/y limits.

Examples using `matplotlib.axes.Axes.update_datalim`

matplotlib.axes.Axes.update_datalim_bounds

`Axes.update_datalim_bounds`(*self*, *bounds*)

[*Deprecated*] Extend the `datalim` Bbox to include the given *Bbox*.

Parameters

bounds

[*Bbox*]

Notes

Deprecated since version 3.3.

Examples using `matplotlib.axes.Axes.update_datalim_bounds`

matplotlib.axes.Axes.set_xbound

`Axes.set_xbound`(*self*, *lower=None*, *upper=None*)

Set the lower and upper numerical bounds of the x-axis.

This method will honor axes inversion regardless of parameter order. It will not change the autoscaling setting (*get_autoscalex_on()*).

Parameters

lower, upper

[float or None] The lower and upper bounds. If *None*, the respective axis bound is not modified.

See also:

get_xbound

get_xlim, *set_xlim*

invert_xaxis, *xaxis_inverted*

Examples using `matplotlib.axes.Axes.set_xbound`

matplotlib.axes.Axes.get_xbound

`Axes.get_xbound`(*self*)

Return the lower and upper x-axis bounds, in increasing order.

See also:

set_xbound

get_xlim, *set_xlim*

invert_xaxis, *xaxis_inverted*

Examples using `matplotlib.axes.Axes.get_xbound`

matplotlib.axes.Axes.set_ybound

`Axes.set_ybound`(*self*, *lower=None*, *upper=None*)

Set the lower and upper numerical bounds of the y-axis.

This method will honor axes inversion regardless of parameter order. It will not change the autoscaling setting (*get_autoscaley_on()*).

Parameters

lower, upper

[float or None] The lower and upper bounds. If *None*, the respective axis bound is not modified.

See also:

get_ybound

get_ylim, *set_ylim*

invert_yaxis, *yaxis_inverted*

Examples using `matplotlib.axes.Axes.set_ybound`

matplotlib.axes.Axes.get_ybound

`Axes.get_ybound`(*self*)

Return the lower and upper y-axis bounds, in increasing order.

See also:

set_ybound

get_ylim, *set_ylim*

invert_yaxis, *yaxis_inverted*

Examples using `matplotlib.axes.Axes.get_ybound`

Axis Labels, title, and legend

Axes.set_xlabel	Set the label for the x-axis.
Axes.get_xlabel	Get the xlabel text string.
Axes.set_ylabel	Set the label for the y-axis.
Axes.get_ylabel	Get the ylabel text string.
Axes.set_title	Set a title for the axes.
Axes.get_title	Get an axes title.
Axes.legend	Place a legend on the axes.
Axes.get_legend	Return the *Legend* instance, or None if no legend is defined.
Axes.get_legend_handles_labels	Return handles and labels for legend

matplotlib.axes.Axes.set_xlabel

Axes.**set_xlabel**(*self*, *xlabel*, *fontdict=None*, *labelpad=None*, ***, *loc=None*, ***kwargs*)

Set the label for the x-axis.

Parameters

xlabel

[str] The label text.

labelpad

[float, default: None] Spacing in points from the axes bounding box including ticks and tick labels.

loc

[{'left', 'center', 'right'}, default: `rcParams["xaxis.labellocation"]` (default: `'center'`)] The label position. This is a high-level alternative for passing parameters *x* and *horizontalalignment*.

Other Parameters

****kwargs**

[*Text* properties] *Text* properties control the appearance of the label.

See also:

text

Documents the properties supported by *Text*.

Examples using `matplotlib.axes.Axes.set_xlabel`

- sphx_glr_gallery_lines_bars_and_markers_barh.py
- sphx_glr_gallery_lines_bars_and_markers_broken_barh.py
- sphx_glr_gallery_lines_bars_and_markers_csd_demo.py
- sphx_glr_gallery_lines_bars_and_markers_fill_between_demo.py
- sphx_glr_gallery_lines_bars_and_markers_fill_betweenx_demo.py
- sphx_glr_gallery_lines_bars_and_markers_filled_step.py
- sphx_glr_gallery_lines_bars_and_markers_psd_demo.py
- sphx_glr_gallery_lines_bars_and_markers_scatter_demo2.py
- sphx_glr_gallery_lines_bars_and_markers_stackplot_demo.py
- sphx_glr_gallery_lines_bars_and_markers_vline_hline_demo.py
- sphx_glr_gallery_images_contours_and_fields_contourf_demo.py
- sphx_glr_gallery_images_contours_and_fields_tricontour_demo.py
- sphx_glr_gallery_images_contours_and_fields_tripcolor_demo.py
- sphx_glr_gallery_images_contours_and_fields_triplot_demo.py
- sphx_glr_gallery_subplots_axes_and_figures_align_labels_demo.py
- sphx_glr_gallery_subplots_axes_and_figures_axes_demo.py
- sphx_glr_gallery_subplots_axes_and_figures_axis_labels_demo.py
- sphx_glr_gallery_subplots_axes_and_figures_demo_constrained_layout.py
- sphx_glr_gallery_subplots_axes_and_figures_demo_tight_layout.py
- sphx_glr_gallery_subplots_axes_and_figures_invert_axes.py
- sphx_glr_gallery_subplots_axes_and_figures_secondary_axis.py
- sphx_glr_gallery_subplots_axes_and_figures_subplot.py
- sphx_glr_gallery_subplots_axes_and_figures_two_scales.py
- sphx_glr_gallery_statistics_boxplot_color.py

- sphx_glr_gallery_statistics_boxplot_demo.py
- sphx_glr_gallery_statistics_boxplot_vs_violin.py
- sphx_glr_gallery_statistics_customized_violin.py
- sphx_glr_gallery_statistics_histogram_cumulative.py
- sphx_glr_gallery_statistics_histogram_features.py
- sphx_glr_gallery_statistics_multiple_histograms_side_by_side.py
- sphx_glr_gallery_text_labels_and_annotations_accented_text.py
- sphx_glr_gallery_text_labels_and_annotations_engineering_formatter.py
- sphx_glr_gallery_text_labels_and_annotations_font_file.py
- sphx_glr_gallery_text_labels_and_annotations_legend_demo.py
- sphx_glr_gallery_text_labels_and_annotations_mathtext_demo.py
- sphx_glr_gallery_text_labels_and_annotations_tex_demo.py
- sphx_glr_gallery_text_labels_and_annotations_titles_demo.py
- sphx_glr_gallery_pyplots_fig_axes_labels_simple.py
- sphx_glr_gallery_pyplots_text_commands.py
- sphx_glr_gallery_color_color_demo.py
- sphx_glr_gallery_shapes_and_collections_collections.py
- sphx_glr_gallery_shapes_and_collections_ellipse_collection.py
- sphx_glr_gallery_style_sheets_dark_background.py
- sphx_glr_gallery_axes_grid1_make_room_for_ylabel_using_axesgrid.py
- sphx_glr_gallery_axes_grid1_parasite_simple.py
- sphx_glr_gallery_axisartist_demo_parasite_axes.py
- sphx_glr_gallery_axisartist_demo_parasite_axes2.py
- sphx_glr_gallery_axisartist_demo_ticklabel_alignment.py
- sphx_glr_gallery_axisartist_simple_axis_direction03.py
- sphx_glr_gallery_axisartist_simple_axisline.py
- sphx_glr_gallery_showcase_anatomy.py
- sphx_glr_gallery_showcase_xkcd.py
- *Animated 3D random walk*
- sphx_glr_gallery_event_handling_keypress_demo.py
- sphx_glr_gallery_misc_pythonic_matplotlib.py
- sphx_glr_gallery_mplot3d_2dcollections3d.py

- sphx_glr_gallery_mplot3d_bars3d.py
- sphx_glr_gallery_mplot3d_contour3d_3.py
- sphx_glr_gallery_mplot3d_contourf3d_2.py
- sphx_glr_gallery_mplot3d_lorenz_attractor.py
- sphx_glr_gallery_mplot3d_offset.py
- sphx_glr_gallery_mplot3d_polys3d.py
- sphx_glr_gallery_mplot3d_scatter3d.py
- sphx_glr_gallery_mplot3d_surface3d_radial.py
- sphx_glr_gallery_mplot3d_text3d.py
- sphx_glr_gallery_recipes_fill_between_alpha.py
- sphx_glr_gallery_scales_log_bar.py
- sphx_glr_gallery_specialty_plots_mri_with_eeg.py
- sphx_glr_gallery_ticks_and_spines_centered_ticklabels.py
- sphx_glr_gallery_ticks_and_spines_multiple_yaxis_with_spines.py
- *Usage Guide*
- *Artist tutorial*
- *Constrained Layout Guide*
- *Tight Layout guide*
- *Choosing Colormaps in Matplotlib*
- *Text in Matplotlib Plots*

matplotlib.axes.Axes.get_xlabel

Axes.**get_xlabel**(*self*)
Get the xlabel text string.

Examples using `matplotlib.axes.Axes.get_xlabel`

matplotlib.axes.Axes.set_ylabel

Axes.**set_ylabel**(*self*, *ylabel*, *fontdict=None*, *labelpad=None*, *, *loc=None*, ***kwargs*)
Set the label for the y-axis.

Parameters

ylabel

[str] The label text.

labelpad

[float, default: None] Spacing in points from the axes bounding box including ticks and tick labels.

loc

[{'bottom', 'center', 'top'}, default: `rcParams["yaxis.labellocation"]` (default: `'center'`)] The label position. This is a high-level alternative for passing parameters *y* and *horizontalalignment*.

Other Parameters

****kwargs**

[*Text* properties] *Text* properties control the appearance of the label.

See also:

text

Documents the properties supported by *Text*.

Examples using `matplotlib.axes.Axes.set_ylabel`

- sphx_glr_gallery_lines_bars_and_markers_bar_stacked.py
- sphx_glr_gallery_lines_bars_and_markers_barchart.py
- sphx_glr_gallery_lines_bars_and_markers_csd_demo.py
- sphx_glr_gallery_lines_bars_and_markers_filled_step.py
- sphx_glr_gallery_lines_bars_and_markers_psd_demo.py
- sphx_glr_gallery_lines_bars_and_markers_scatter_demo2.py
- sphx_glr_gallery_lines_bars_and_markers_stackplot_demo.py
- sphx_glr_gallery_images_contours_and_fields_contourf_demo.py
- sphx_glr_gallery_images_contours_and_fields_image_annotated_heatmap.py
- sphx_glr_gallery_images_contours_and_fields_tricontour_demo.py
- sphx_glr_gallery_images_contours_and_fields_tripcolor_demo.py
- sphx_glr_gallery_images_contours_and_fields_triplot_demo.py
- sphx_glr_gallery_subplots_axes_and_figures_align_labels_demo.py

- sphx_glr_gallery_subplots_axes_and_figures_axes_demo.py
- sphx_glr_gallery_subplots_axes_and_figures_axis_labels_demo.py
- sphx_glr_gallery_subplots_axes_and_figures_demo_constrained_layout.py
- sphx_glr_gallery_subplots_axes_and_figures_demo_tight_layout.py
- sphx_glr_gallery_subplots_axes_and_figures_invert_axes.py
- sphx_glr_gallery_subplots_axes_and_figures_secondary_axis.py
- sphx_glr_gallery_subplots_axes_and_figures_subplot.py
- sphx_glr_gallery_subplots_axes_and_figures_two_scales.py
- sphx_glr_gallery_statistics_boxplot_color.py
- sphx_glr_gallery_statistics_boxplot_demo.py
- sphx_glr_gallery_statistics_boxplot_vs_violin.py
- sphx_glr_gallery_statistics_customized_violin.py
- sphx_glr_gallery_statistics_histogram_cumulative.py
- sphx_glr_gallery_statistics_histogram_features.py
- sphx_glr_gallery_statistics_multiple_histograms_side_by_side.py
- sphx_glr_gallery_text_labels_and_annotations_accented_text.py
- sphx_glr_gallery_text_labels_and_annotations_legend_demo.py
- sphx_glr_gallery_text_labels_and_annotations_mathtext_demo.py
- sphx_glr_gallery_text_labels_and_annotations_tex_demo.py
- sphx_glr_gallery_pyplots_align_ylabels.py
- sphx_glr_gallery_pyplots_fig_axes_labels_simple.py
- sphx_glr_gallery_pyplots_text_commands.py
- sphx_glr_gallery_color_color_demo.py
- sphx_glr_gallery_shapes_and_collections_collections.py
- sphx_glr_gallery_shapes_and_collections_ellipse_collection.py
- sphx_glr_gallery_style_sheets_dark_background.py
- sphx_glr_gallery_axes_grid1_make_room_for_ylabel_using_axesgrid.py
- sphx_glr_gallery_axes_grid1_parasite_simple.py
- sphx_glr_gallery_axisartist_demo_parasite_axes.py
- sphx_glr_gallery_axisartist_demo_parasite_axes2.py
- sphx_glr_gallery_axisartist_demo_ticklabel_alignment.py
- sphx_glr_gallery_axisartist_simple_axis_direction03.py

- sphx_glr_gallery_axisartist_simple_axisline.py
- sphx_glr_gallery_showcase_anatomy.py
- sphx_glr_gallery_showcase_xkcd.py
- *Animated 3D random walk*
- sphx_glr_gallery_misc_pythonic_matplotlib.py
- sphx_glr_gallery_mplot3d_2dcollections3d.py
- sphx_glr_gallery_mplot3d_bars3d.py
- sphx_glr_gallery_mplot3d_contour3d_3.py
- sphx_glr_gallery_mplot3d_contourf3d_2.py
- sphx_glr_gallery_mplot3d_lorenz_attractor.py
- sphx_glr_gallery_mplot3d_mixed_subplots.py
- sphx_glr_gallery_mplot3d_offset.py
- sphx_glr_gallery_mplot3d_polys3d.py
- sphx_glr_gallery_mplot3d_scatter3d.py
- sphx_glr_gallery_mplot3d_surface3d_radial.py
- sphx_glr_gallery_mplot3d_text3d.py
- sphx_glr_gallery_recipes_fill_between_alpha.py
- sphx_glr_gallery_scales_log_bar.py
- sphx_glr_gallery_specialty_plots_mri_with_eeg.py
- sphx_glr_gallery_specialty_plots_topographic_hillshading.py
- sphx_glr_gallery_ticks_and_spines_multiple_yaxis_with_spines.py
- *Usage Guide*
- *Artist tutorial*
- *Constrained Layout Guide*
- *Tight Layout guide*
- *Choosing Colormaps in Matplotlib*
- *Text in Matplotlib Plots*

matplotlib.axes.Axes.get_ylabel

Axes.get_ylabel(*self*)
Get the ylabel text string.

Examples using matplotlib.axes.Axes.get_ylabel

matplotlib.axes.Axes.set_title

Axes.set_title(*self*, *label*, *fontdict=None*, *loc=None*, *pad=None*, *, *y=None*, ***kwargs*)
Set a title for the axes.

Set one of the three available axes titles. The available titles are positioned above the axes in the center, flush with the left edge, and flush with the right edge.

Parameters

label

[str] Text to use for the title

fontdict

[dict] A dictionary controlling the appearance of the title text, the default *fontdict* is:

```
{'fontsize': rcParams['axes.titlesize'],
 'fontweight': rcParams['axes.titleweight'],
 'color': rcParams['axes.titlecolor'],
 'verticalalignment': 'baseline',
 'horizontalalignment': loc}
```

loc

[{'center', 'left', 'right'}, default: rcParams["axes.titlelocation"] (default: 'center')] Which title to set.

y

[float, default: rcParams["axes.titley"] (default: None)] Vertical axes loation for the title (1.0 is the top). If None (the default), y is determined automatically to avoid decorators on the axes.

pad

[float, default: rcParams["axes.titlepad"] (default: 6.0)] The offset of the title from the top of the axes, in points.

Returns

Text

The matplotlib text instance representing the title

Other Parameters

****kwargs**

[*Text* properties] Other keyword arguments are text properties, see *Text* for a list of valid text properties.

Examples using `matplotlib.axes.Axes.set_title`

- sphx_glr_gallery_lines_bars_and_markers_bar_stacked.py
- sphx_glr_gallery_lines_bars_and_markers_barchart.py
- sphx_glr_gallery_lines_bars_and_markers_barh.py
- sphx_glr_gallery_lines_bars_and_markers_errorbar_subsample.py
- sphx_glr_gallery_lines_bars_and_markers_eventcollection_demo.py
- sphx_glr_gallery_lines_bars_and_markers_fill_between_demo.py
- sphx_glr_gallery_lines_bars_and_markers_fill_betweenx_demo.py
- sphx_glr_gallery_lines_bars_and_markers_joinstyle.py
- sphx_glr_gallery_lines_bars_and_markers_markevery_demo.py
- sphx_glr_gallery_lines_bars_and_markers_psd_demo.py
- sphx_glr_gallery_lines_bars_and_markers_scatter_demo2.py
- sphx_glr_gallery_lines_bars_and_markers_span_regions.py
- sphx_glr_gallery_lines_bars_and_markers_stackplot_demo.py
- sphx_glr_gallery_lines_bars_and_markers_vline_hline_demo.py
- sphx_glr_gallery_images_contours_and_fields_contour_corner_mask.py
- sphx_glr_gallery_images_contours_and_fields_contour_demo.py
- sphx_glr_gallery_images_contours_and_fields_contour_label_demo.py
- sphx_glr_gallery_images_contours_and_fields_contourf_demo.py
- sphx_glr_gallery_images_contours_and_fields_image_annotated_heatmap.py
- sphx_glr_gallery_images_contours_and_fields_image_antialiasing.py
- sphx_glr_gallery_images_contours_and_fields_image_demo.py
- sphx_glr_gallery_images_contours_and_fields_image_masked.py
- sphx_glr_gallery_images_contours_and_fields_image_nonuniform.py

- sphx_glr_gallery_images_contours_and_fields_interpolation_methods.py
- sphx_glr_gallery_images_contours_and_fields_irregulardatagrid.py
- sphx_glr_gallery_images_contours_and_fields_pcolor_demo.py
- sphx_glr_gallery_images_contours_and_fields_pcolormesh_grids.py
- sphx_glr_gallery_images_contours_and_fields_pcolormesh_levels.py
- sphx_glr_gallery_images_contours_and_fields_plot_streamplot.py
- sphx_glr_gallery_images_contours_and_fields_quiver_demo.py
- sphx_glr_gallery_images_contours_and_fields_tricontour_demo.py
- sphx_glr_gallery_images_contours_and_fields_tricontour_smooth_delaunay.py
- sphx_glr_gallery_images_contours_and_fields_tricontour_smooth_user.py
- sphx_glr_gallery_images_contours_and_fields_trigradient_demo.py
- sphx_glr_gallery_images_contours_and_fields_tripcolor_demo.py
- sphx_glr_gallery_images_contours_and_fields_triplot_demo.py
- sphx_glr_gallery_subplots_axes_and_figures_axes_demo.py
- sphx_glr_gallery_subplots_axes_and_figures_axes_margins.py
- sphx_glr_gallery_subplots_axes_and_figures_demo_constrained_layout.py
- sphx_glr_gallery_subplots_axes_and_figures_demo_tight_layout.py
- sphx_glr_gallery_subplots_axes_and_figures_invert_axes.py
- sphx_glr_gallery_subplots_axes_and_figures_secondary_axis.py
- sphx_glr_gallery_subplots_axes_and_figures_subplots_demo.py
- sphx_glr_gallery_statistics_boxplot_color.py
- sphx_glr_gallery_statistics_confidence_ellipse.py
- sphx_glr_gallery_statistics_customized_violin.py
- sphx_glr_gallery_statistics_errorbar_features.py
- sphx_glr_gallery_statistics_errorbar_limits.py
- sphx_glr_gallery_statistics_hexbin_demo.py
- sphx_glr_gallery_statistics_histogram_cumulative.py
- sphx_glr_gallery_statistics_histogram_features.py
- sphx_glr_gallery_statistics_histogram_multihist.py
- sphx_glr_gallery_pie_and_polar_charts_bar_of_pie.py
- sphx_glr_gallery_pie_and_polar_charts_pie_and_donut_labels.py
- sphx_glr_gallery_pie_and_polar_charts_polar_demo.py

- sphx_glr_gallery_text_labels_and_annotations_accented_text.py
- sphx_glr_gallery_text_labels_and_annotations_date_index_formatter.py
- sphx_glr_gallery_text_labels_and_annotations_engineering_formatter.py
- sphx_glr_gallery_text_labels_and_annotations_font_file.py
- sphx_glr_gallery_text_labels_and_annotations_legend_demo.py
- sphx_glr_gallery_text_labels_and_annotations_mathtext_demo.py
- sphx_glr_gallery_text_labels_and_annotations_tex_demo.py
- sphx_glr_gallery_text_labels_and_annotations_titles_demo.py
- sphx_glr_gallery_text_labels_and_annotations_usetex_baseline_test.py
- sphx_glr_gallery_pyplots_align_ylabels.py
- sphx_glr_gallery_pyplots_boxplot_demo_pyplot.py
- sphx_glr_gallery_pyplots_fig_axes_labels_simple.py
- sphx_glr_gallery_pyplots_text_commands.py
- sphx_glr_gallery_pyplots_whats_new_98_4_fill_between.py
- sphx_glr_gallery_color_color_demo.py
- sphx_glr_gallery_color_custom_cmap.py
- sphx_glr_gallery_shapes_and_collections_collections.py
- sphx_glr_gallery_shapes_and_collections_compound_path.py
- sphx_glr_gallery_shapes_and_collections_donut.py
- sphx_glr_gallery_shapes_and_collections_line_collection.py
- sphx_glr_gallery_shapes_and_collections_quad_bezier.py
- sphx_glr_gallery_style_sheets_bmh.py
- sphx_glr_gallery_style_sheets_dark_background.py
- sphx_glr_gallery_style_sheets_fivethirtyeight.py
- sphx_glr_gallery_axes_grid1_make_room_for_ylabel_using_axesgrid.py
- sphx_glr_gallery_showcase_anatomy.py
- sphx_glr_gallery_showcase_xkcd.py
- sphx_glr_gallery_animation_animation_demo.py
- *Animated 3D random walk*
- sphx_glr_gallery_event_handling_data_browser.py
- sphx_glr_gallery_event_handling_image_slices_viewer.py
- sphx_glr_gallery_event_handling_keypress_demo.py

- sphx_glr_gallery_event_handling_lasso_demo.py
- sphx_glr_gallery_event_handling_legend_picking.py
- sphx_glr_gallery_event_handling_looking_glass.py
- sphx_glr_gallery_event_handling_path_editor.py
- sphx_glr_gallery_event_handling_pick_event_demo2.py
- sphx_glr_gallery_event_handling_poly_editor.py
- sphx_glr_gallery_event_handling_viewlims.py
- sphx_glr_gallery_misc_agg_buffer_to_array.py
- sphx_glr_gallery_misc_cursor_demo.py
- sphx_glr_gallery_misc_pythonic_matplotlib.py
- sphx_glr_gallery_misc_rasterization_demo.py
- sphx_glr_gallery_misc_zorder_demo.py
- sphx_glr_gallery_mplot3d_3d_bars.py
- sphx_glr_gallery_mplot3d_lorenz_attractor.py
- sphx_glr_gallery_mplot3d_wire3d_zero_stride.py
- sphx_glr_gallery_recipes_common_date_problems.py
- sphx_glr_gallery_recipes_fill_between_alpha.py
- sphx_glr_gallery_scales_aspect_loglog.py
- sphx_glr_gallery_scales_power_norm.py
- sphx_glr_gallery_scales_scales.py
- sphx_glr_gallery_specialty_plots_radar_chart.py
- sphx_glr_gallery_specialty_plots_topographic_hillshading.py
- sphx_glr_gallery_ticks_and_spines_colorbar_tick_labelling_demo.py
- sphx_glr_gallery_ticks_and_spines_date_precision_and_epochs.py
- sphx_glr_gallery_ticks_and_spines_spine_placement_demo.py
- sphx_glr_gallery_ticks_and_spines_spines.py
- sphx_glr_gallery_ticks_and_spines_spines_dropped.py
- sphx_glr_gallery_ticks_and_spines_tick_xlabel_top.py
- sphx_glr_gallery_units_artist_tests.py
- sphx_glr_gallery_units_bar_unit_demo.py
- sphx_glr_gallery_units_evans_test.py
- sphx_glr_gallery_widgets_rectangle_selector.py

- sphx_glr_gallery_widgets_span_selector.py
- *Usage Guide*
- *Image tutorial*
- *Artist tutorial*
- *Styling with cycler*
- *Customizing Figure Layouts Using GridSpec and Other Functions*
- *Constrained Layout Guide*
- *Tight Layout guide*
- *Transformations Tutorial*
- *Specifying Colors*
- *Text in Matplotlib Plots*

matplotlib.axes.Axes.get_title

Axes.get_title(*self*, *loc='center'*)

Get an axes title.

Get one of the three available axes titles. The available titles are positioned above the axes in the center, flush with the left edge, and flush with the right edge.

Parameters

loc

[{'center', 'left', 'right'}, str, default: 'center'] Which title to return.

Returns

str

The title text string.

Examples using `matplotlib.axes.Axes.get_title`

matplotlib.axes.Axes.legend

Axes.**legend**(*self*, **args*, ***kwargs*)

Place a legend on the axes.

Call signatures:

```
legend()
legend(labels)
legend(handles, labels)
```

The call signatures correspond to three different ways how to use this method.

1. Automatic detection of elements to be shown in the legend

The elements to be added to the legend are automatically determined, when you do not pass in any extra arguments.

In this case, the labels are taken from the artist. You can specify them either at artist creation or by calling the *set_label()* method on the artist:

```
line, = ax.plot([1, 2, 3], label='Inline label')
ax.legend()
```

or:

```
line, = ax.plot([1, 2, 3])
line.set_label('Label via method')
ax.legend()
```

Specific lines can be excluded from the automatic legend element selection by defining a label starting with an underscore. This is default for all artists, so calling *Axes.legend* without any arguments and without setting the labels manually will result in no legend being drawn.

2. Labeling existing plot elements

To make a legend for lines which already exist on the axes (via plot for instance), simply call this function with an iterable of strings, one for each legend item. For example:

```
ax.plot([1, 2, 3])
ax.legend(['A simple line'])
```

Note: This way of using is discouraged, because the relation between plot elements and labels is only implicit by their order and can easily be mixed up.

3. Explicitly defining the elements in the legend

For full control of which artists have a legend entry, it is possible to pass an iterable of legend artists followed by an iterable of legend labels respectively:

```
legend((line1, line2, line3), ('label1', 'label2', 'label3'))
```

Parameters

handles

[sequence of *Artist*, optional] A list of Artists (lines, patches) to be added to the legend. Use this together with *labels*, if you need full control on what is shown in the legend and the automatic mechanism described above is not sufficient.

The length of handles and labels should be the same in this case. If they are not, they are truncated to the smaller length.

labels

[list of str, optional] A list of labels to show next to the artists. Use this together with *handles*, if you need full control on what is shown in the legend and the automatic mechanism described above is not sufficient.

Returns

Legend

Other Parameters

loc

[str or pair of floats, default: `rcParams["legend.loc"]` (default: `'best'`) ('best' for axes, 'upper right' for figures)] The location of the legend.

The strings `'upper left'`, `'upper right'`, `'lower left'`, `'lower right'` place the legend at the corresponding corner of the axes/figure.

The strings `'upper center'`, `'lower center'`, `'center left'`, `'center right'` place the legend at the center of the corresponding edge of the axes/figure.

The string `'center'` places the legend at the center of the axes/figure.

The string `'best'` places the legend at the location, among the nine locations defined so far, with the minimum overlap with other drawn artists. This option can be quite slow for plots with large amounts of data; your plotting speed may benefit from providing a specific location.

The location can also be a 2-tuple giving the coordinates of the lower-left corner of the legend in axes coordinates (in which case *bbox_to_anchor* will be ignored).

For back-compatibility, `'center right'` (but no other location) can also be spelled `'right'`, and each "string" locations can also be given as a numeric value:

Location String	Location Code
'best'	0
'upper right'	1
'upper left'	2
'lower left'	3
'lower right'	4
'right'	5
'center left'	6
'center right'	7
'lower center'	8
'upper center'	9
'center'	10

bbox_to_anchor

[*BboxBase*, 2-tuple, or 4-tuple of floats] Box that is used to position the legend in conjunction with *loc*. Defaults to `axes.bbox` (if called as a method to *Axes.legend*) or `figure.bbox` (if *Figure.legend*). This argument allows arbitrary placement of the legend.

Bbox coordinates are interpreted in the coordinate system given by *bbox_transform*, with the default transform Axes or Figure coordinates, depending on which `legend` is called.

If a 4-tuple or *BboxBase* is given, then it specifies the bbox `(x, y, width, height)` that the legend is placed in. To put the legend in the best location in the bottom right quadrant of the axes (or figure):

```
loc='best', bbox_to_anchor=(0.5, 0., 0.5, 0.5)
```

A 2-tuple `(x, y)` places the corner of the legend specified by *loc* at x, y. For example, to put the legend's upper right-hand corner in the center of the axes (or figure) the following keywords can be used:

```
loc='upper right', bbox_to_anchor=(0.5, 0.5)
```

ncol

[int, default: 1] The number of columns that the legend has.

prop

[None or *matplotlib.font_manager.FontProperties* or dict] The font properties of the legend. If None (default), the current *matplotlib.rcParams* will be used.

fontsize

[int or {'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large'}] The font size of the legend. If the value is numeric the size will be the absolute font size in points. String values are relative to the current default font size. This argument is only used if *prop* is not specified.

labelcolor

[str or list] Sets the color of the text in the legend. Can be a valid color string (for example, 'red'), or a list of color strings. The labelcolor can also be made to match the color of the line or marker using 'linecolor', 'markerfacecolor' (or 'mfc'), or 'markeredgecolor' (or 'mec').

numpoints

[int, default: `rcParams["legend.numpoints"]` (default: `1`)] The number of marker points in the legend when creating a legend entry for a *Line2D* (line).

scatterpoints

[int, default: `rcParams["legend.scatterpoints"]` (default: `1`)] The number of marker points in the legend when creating a legend entry for a *PathCollection* (scatter plot).

scatteryoffsets

[iterable of floats, default: `[0.375, 0.5, 0.3125]`] The vertical offset (relative to the font size) for the markers created for a scatter plot legend entry. 0.0 is at the base the legend text, and 1.0 is at the top. To draw all markers at the same height, set to `[0.5]`.

markerscale

[float, default: `rcParams["legend.markerscale"]` (default: `1.0`)] The relative size of legend markers compared with the originally drawn ones.

markerfirst

[bool, default: True] If *True*, legend marker is placed to the left of the legend label. If *False*, legend marker is placed to the right of the legend label.

frameon

[bool, default: `rcParams["legend.frameon"]` (default: `True`)] Whether the legend should be drawn on a patch (frame).

fancybox

[bool, default: rcParams["legend.fancybox"] (default: True)] Whether round edges should be enabled around the *FancyBboxPatch* which makes up the legend's background.

shadow

[bool, default: rcParams["legend.shadow"] (default: False)] Whether to draw a shadow behind the legend.

framealpha

[float, default: rcParams["legend.framealpha"] (default: 0.8)] The alpha transparency of the legend's background. If *shadow* is activated and *framealpha* is None, the default value is ignored.

facecolor

["inherit" or color, default: rcParams["legend.facecolor"] (default: 'inherit')] The legend's background color. If "inherit", use rcParams["axes.facecolor"] (default: 'white').

edgecolor

["inherit" or color, default: rcParams["legend.edgecolor"] (default: '0.8')] The legend's background patch edge color. If "inherit", use take rcParams["axes.edgecolor"] (default: 'black').

mode

[{"expand", None}] If *mode* is set to "expand" the legend will be horizontally expanded to fill the axes area (or *bbox_to_anchor* if defines the legend's size).

bbox_transform

[None or *matplotlib.transforms.Transform*] The transform for the bounding box (*bbox_to_anchor*). For a value of None (default) the Axes' transAxes transform will be used.

title

[str or None] The legend's title. Default is no title (None).

title_fontsize

[int or {'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large'}, default: rcParams["legend.title_fontsize"] (default: None)] The font size of the legend's title.

borderpad

[float, default: rcParams["legend.borderpad"] (default: 0.4)] The fractional whitespace inside the legend border, in font-size units.

labelspacing

[float, default: `rcParams["legend.labelspacing"]` (default: `0.5`)]
The vertical space between the legend entries, in font-size units.

handlelength

[float, default: `rcParams["legend.handlelength"]` (default: `2.0`)]
The length of the legend handles, in font-size units.

handletextpad

[float, default: `rcParams["legend.handletextpad"]` (default: `0.8`)]
The pad between the legend handle and text, in font-size units.

borderaxespad

[float, default: `rcParams["legend.borderaxespad"]` (default: `0.5`)]
The pad between the axes and legend border, in font-size units.

columnspacing

[float, default: `rcParams["legend.columnspacing"]` (default: `2.0`)]
The spacing between columns, in font-size units.

handler_map

[dict or None] The custom dictionary mapping instances or types to a legend handler. This *handler_map* updates the default handler map found at *matplotlib.legend.Legend.get_legend_handler_map*.

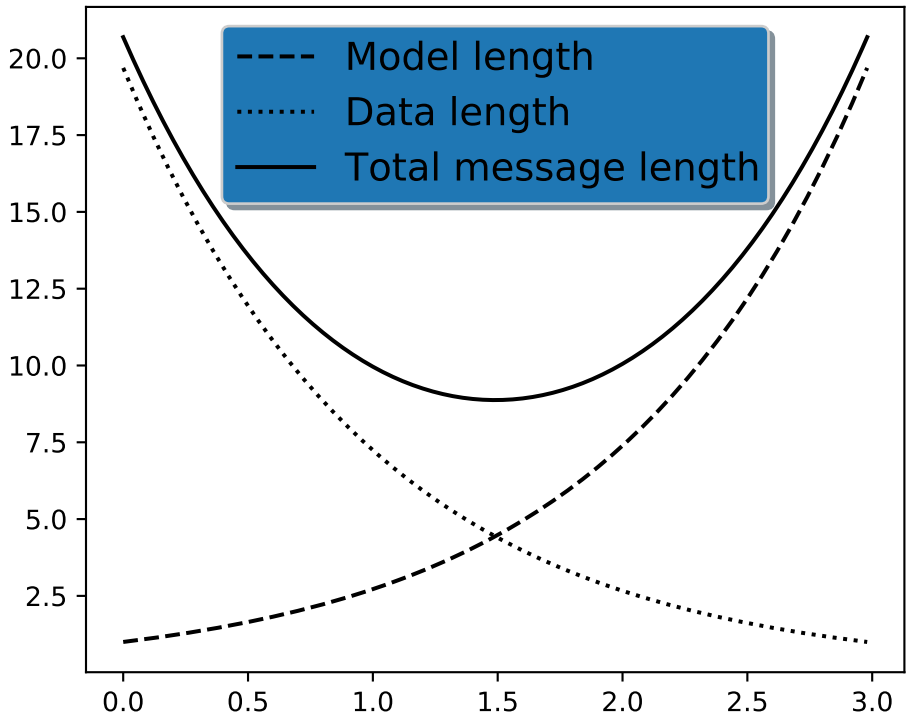
Notes

Some artists are not supported by this function. See *Legend guide* for details.

Examples

Examples using `matplotlib.axes.Axes.legend`

- sphx_glr_gallery_lines_bars_and_markers_bar_stacked.py
- sphx_glr_gallery_lines_bars_and_markers_barchart.py
- sphx_glr_gallery_lines_bars_and_markers_categorical_variables.py
- sphx_glr_gallery_lines_bars_and_markers_horizontal_barchart_distribution.py
- sphx_glr_gallery_lines_bars_and_markers_line_demo_dash_control.py
- sphx_glr_gallery_lines_bars_and_markers_markevery_prop_cycle.py
- sphx_glr_gallery_lines_bars_and_markers_scatter_with_legend.py



- sphx_glr_gallery_lines_bars_and_markers_stackplot_demo.py
- sphx_glr_gallery_images_contours_and_fields_contourf_hatching.py
- sphx_glr_gallery_images_contours_and_fields_contourf_log.py
- sphx_glr_gallery_subplots_axes_and_figures_secondary_axis.py
- sphx_glr_gallery_statistics_confidence_ellipse.py
- sphx_glr_gallery_statistics_histogram_cumulative.py
- sphx_glr_gallery_statistics_histogram_multihist.py
- sphx_glr_gallery_pie_and_polar_charts_bar_of_pie.py
- sphx_glr_gallery_pie_and_polar_charts_pie_and_donut_labels.py
- sphx_glr_gallery_pie_and_polar_charts_polar_legend.py
- sphx_glr_gallery_text_labels_and_annotations_custom_legends.py
- sphx_glr_gallery_text_labels_and_annotations_legend.py
- sphx_glr_gallery_text_labels_and_annotations_legend_demo.py
- sphx_glr_gallery_text_labels_and_annotations_mathtext_demo.py
- sphx_glr_gallery_text_labels_and_annotations_tex_demo.py

- sphx_glr_gallery_pyplots_whats_new_98_4_legend.py
- sphx_glr_gallery_axisartist_demo_parasite_axes.py
- sphx_glr_gallery_axisartist_demo_parasite_axes2.py
- sphx_glr_gallery_showcase_anatomy.py
- sphx_glr_gallery_event_handling_legend_picking.py
- sphx_glr_gallery_misc_patheffect_demo.py
- sphx_glr_gallery_mplot3d_2dcollections3d.py
- sphx_glr_gallery_mplot3d_lines3d.py
- sphx_glr_gallery_recipes_fill_between_alpha.py
- sphx_glr_gallery_ticks_and_spines_multiple_yaxis_with_spines.py
- sphx_glr_gallery_units_bar_unit_demo.py
- sphx_glr_gallery_userdemo_simple_legend02.py
- *Usage Guide*
- *Constrained Layout Guide*
- *Tight Layout guide*
- *Specifying Colors*

matplotlib.axes.Axes.get_legend

Axes.**get_legend**(*self*)
Return the *Legend* instance, or None if no legend is defined.

Examples using `matplotlib.axes.Axes.get_legend`

matplotlib.axes.Axes.get_legend_handles_labels

Axes.**get_legend_handles_labels**(*self*, *legend_handler_map=None*)
Return handles and labels for legend

`ax.legend()` is equivalent to

```
h, l = ax.get_legend_handles_labels()
ax.legend(h, l)
```

Examples using `matplotlib.axes.Axes.get_legend_handles_labels`

- *Legend guide*

Axis scales

Axes.set_xscale	Set the x-axis scale.
Axes.get_xscale	Return the xaxis' scale (as a str).
Axes.set_yscale	Set the y-axis scale.
Axes.get_yscale	Return the yaxis' scale (as a str).

matplotlib.axes.Axes.set_xscale

`Axes.set_xscale`(*self*, *value*, ***kwargs*)

Set the x-axis scale.

Parameters

value

[{"linear", "log", "symlog", "logit", ...}] The axis scale type to apply.

****kwargs**

Different keyword arguments are accepted, depending on the scale. See the respective class keyword arguments:

- *matplotlib.scale.LinearScale*
- *matplotlib.scale.LogScale*
- *matplotlib.scale.SymmetricalLogScale*
- *matplotlib.scale.LogitScale*

Notes

By default, Matplotlib supports the above mentioned scales. Additionally, custom scales may be registered using *matplotlib.scale.register_scale*. These scales can then also be used here.

Examples using `matplotlib.axes.Axes.set_xscale`

- sphx_glr_gallery_lines_bars_and_markers_markevery_demo.py
- sphx_glr_gallery_text_labels_and_annotations_engineering_formatter.py
- sphx_glr_gallery_axes_grid1_inset_locator_demo.py
- sphx_glr_gallery_scales_aspect_loglog.py
- sphx_glr_gallery_scales_log_demo.py
- sphx_glr_gallery_scales_scales.py
- *Transformations Tutorial*

matplotlib.axes.Axes.get_xscale

Axes.**get_xscale**(*self*)
Return the xaxis' scale (as a str).

Examples using `matplotlib.axes.Axes.get_xscale`

matplotlib.axes.Axes.set_yscale

Axes.**set_yscale**(*self*, *value*, ***kwargs*)
Set the y-axis scale.

Parameters

value

[{"linear", "log", "symlog", "logit", ...}] The axis scale type to apply.

****kwargs**

Different keyword arguments are accepted, depending on the scale. See the respective class keyword arguments:

- *matplotlib.scale.LinearScale*
- *matplotlib.scale.LogScale*
- *matplotlib.scale.SymmetricalLogScale*
- *matplotlib.scale.LogitScale*

Notes

By default, Matplotlib supports the above mentioned scales. Additionally, custom scales may be registered using *matplotlib.scale.register_scale*. These scales can then also be used here.

Examples using `matplotlib.axes.Axes.set_yscale`

- sphx_glr_gallery_lines_bars_and_markers_markevery_demo.py
- sphx_glr_gallery_statistics_boxplot.py
- sphx_glr_gallery_statistics_bxp.py
- sphx_glr_gallery_statistics_errorbar_features.py
- sphx_glr_gallery_scales_aspect_loglog.py
- sphx_glr_gallery_scales_log_bar.py
- sphx_glr_gallery_scales_log_demo.py
- sphx_glr_gallery_scales_scales.py

matplotlib.axes.Axes.get_yscale

`Axes.get_yscale`(*self*)
Return the yaxis' scale (as a str).

Examples using `matplotlib.axes.Axes.get_yscale`

Autoscaling and margins

Axes.use_sticky_edges	When autoscaling, whether to obey all `Artist.sticky_edges`.
Axes.margins	Set or retrieve autoscaling margins.
Axes.set_xmargin	Set padding of X data limits prior to autoscaling.
Axes.set_ymargin	Set padding of Y data limits prior to autoscaling.
Axes.relim	Recompute the data limits based on current artists.
Axes.autoscale	Autoscale the axis view to the data (toggle).

continues on next page

Table 102 – continued from previous page

Axes.autoscale_view	Autoscale the view limits using the data limits.
Axes.set_autoscale_on	Set whether autoscaling is applied on plot commands
Axes.get_autoscale_on	Get whether autoscaling is applied for both axes on plot commands
Axes.set_autoscalex_on	Set whether autoscaling for the x-axis is applied on plot commands
Axes.get_autoscalex_on	Get whether autoscaling for the x-axis is applied on plot commands
Axes.set_autoscaley_on	Set whether autoscaling for the y-axis is applied on plot commands
Axes.get_autoscaley_on	Get whether autoscaling for the y-axis is applied on plot commands

matplotlib.axes.Axes.use_sticky_edges

property Axes.use_sticky_edges

When autoscaling, whether to obey all `Artist.sticky_edges`.

Default is `True`.

Setting this to `False` ensures that the specified margins will be applied, even if the plot includes an image, for example, which would otherwise force a view limit to coincide with its data limit.

The changing this property does not change the plot until *autoscale* or *autoscale_view* is called.

matplotlib.axes.Axes.margins

Axes.margins(*self*, **margins*, *x=None*, *y=None*, *tight=True*)

Set or retrieve autoscaling margins.

The padding added to each limit of the axes is the *margin* times the data interval. All input parameters must be floats within the range [0, 1]. Passing both positional and keyword arguments is invalid and will raise a TypeError. If no arguments (positional or otherwise) are provided, the current margins will remain in place and simply be returned.

Specifying any margin changes only the autoscaling; for example, if *xmargin* is not None, then *xmargin* times the X data interval will be added to each end of that interval before it is used in autoscaling.

Parameters

***margins**

[float, optional] If a single positional argument is provided, it specifies both margins of the x-axis and y-axis limits. If two positional arguments are provided, they will be interpreted as *xmargin*, *ymargin*. If setting the margin on a single axis is desired, use the keyword arguments described below.

x, y

[float, optional] Specific margin values for the x-axis and y-axis, respectively. These cannot be used with positional arguments, but can be used individually to alter on e.g., only the y-axis.

tight

[bool or None, default: True] The *tight* parameter is passed to *autoscale_view()*, which is executed after a margin is changed; the default here is *True*, on the assumption that when margins are specified, no additional padding to match tick marks is usually desired. Set *tight* to *None* will preserve the previous setting.

Returns

xmargin, ymargin

[float]

Notes

If a previously used Axes method such as *pcolor()* has set *use_sticky_edges* to `True`, only the limits not set by the "sticky artists" will be modified. To force all of the margins to be set, set *use_sticky_edges* to `False` before calling *margins()*.

Examples using `matplotlib.axes.Axes.margins`

- sphx_glr_gallery_lines_bars_and_markers_marker_reference.py
- sphx_glr_gallery_lines_bars_and_markers_timeline.py
- sphx_glr_gallery_images_contours_and_fields_trigradient_demo.py
- sphx_glr_gallery_subplots_axes_and_figures_axes_margins.py
- sphx_glr_gallery_style_sheets_ggplot.py
- sphx_glr_gallery_widgets_slider_demo.py

matplotlib.axes.Axes.set_xmargin

Axes.**set_xmargin**(*self*, *m*)

Set padding of X data limits prior to autoscaling.

m times the data interval will be added to each end of that interval before it is used in autoscaling. For example, if your data is in the range [0, 2], a factor of `m = 0.1` will result in a range [-0.2, 2.2].

Negative values -0.5 < m < 0 will result in clipping of the data range. I.e. for a data range [0, 2], a factor of `m = -0.1` will result in a range [0.2, 1.8].

Parameters

m

[float greater than -0.5]

Examples using `matplotlib.axes.Axes.set_xmargin`

matplotlib.axes.Axes.set_ymargin

Axes.**set_ymargin**(*self*, *m*)

Set padding of Y data limits prior to autoscaling.

m times the data interval will be added to each end of that interval before it is used in autoscaling. For example, if your data is in the range [0, 2], a factor of `m = 0.1` will result in a range [-0.2, 2.2].

Negative values -0.5 < m < 0 will result in clipping of the data range. I.e. for a data range [0, 2], a factor of `m = -0.1` will result in a range [0.2, 1.8].

Parameters

m

[float greater than -0.5]

Examples using `matplotlib.axes.Axes.set_ymargin`

matplotlib.axes.Axes.relim

Axes.**relim**(*self*, *visible_only=False*)

Recompute the data limits based on current artists.

At present, *Collection* instances are not supported.

Parameters

visible_only

[bool, default: False] Whether to exclude invisible artists.

Examples using `matplotlib.axes.Axes.relim`

- sphx_glr_gallery_widgets_textbox.py

matplotlib.axes.Axes.autoscale

`Axes.autoscale`(*self*, *enable=True*, *axis='both'*, *tight=None*)

Autoscale the axis view to the data (toggle).

Convenience method for simple axis view autoscaling. It turns autoscaling on or off, and then, if autoscaling for either axis is on, it performs the autoscaling on the specified axis or axes.

Parameters

enable

[bool or None, default: True] True turns autoscaling on, False turns it off. None leaves the autoscaling state unchanged.

axis

[{'both', 'x', 'y'}, default: 'both'] Which axis to operate on.

tight

[bool or None, default: None] If True, first set the margins to zero. Then, this argument is forwarded to *autoscale_view* (regardless of its value); see the description of its behavior there.

Examples using `matplotlib.axes.Axes.autoscale`

- sphx_glr_gallery_subplots_axes_and_figures_axes_box_aspect.py

matplotlib.axes.Axes.autoscale_view

`Axes.autoscale_view`(*self*, *tight=None*, *scalex=True*, *scaley=True*)

Autoscale the view limits using the data limits.

Parameters

tight

[bool or None] If *True*, only expand the axis limits using the margins. Note that unlike for *autoscale*, `tight=True` does *not* set the margins to zero.

If *False* and `rcParams["axes.autolimit_mode"]` (default: `'data'`) is 'round_numbers', then after expansion by the margins, further expand the axis limits using the axis major locator.

If None (the default), reuse the value set in the previous call to *autoscale_view* (the initial value is False, but the default style sets `rcParams["axes.autolimit_mode"]` (default: `'data'`) to 'data', in which case this behaves like True).

scalex

[bool, default: True] Whether to autoscale the x axis.

scaley

[bool, default: True] Whether to autoscale the y axis.

Notes

The autoscaling preserves any preexisting axis direction reversal.

The data limits are not updated automatically when artist data are changed after the artist has been added to an Axes instance. In that case, use *matplotlib.axes.Axes.relim()* prior to calling autoscale_view.

If the views of the axes are fixed, e.g. via *set_xlim*, they will not be changed by autoscale_view(). See *matplotlib.axes.Axes.autoscale()* for an alternative.

Examples using `matplotlib.axes.Axes.autoscale_view`

- sphx_glr_gallery_shapes_and_collections_collections.py
- sphx_glr_gallery_shapes_and_collections_compound_path.py
- sphx_glr_gallery_shapes_and_collections_ellipse_collection.py
- sphx_glr_gallery_units_bar_unit_demo.py
- sphx_glr_gallery_widgets_textbox.py

matplotlib.axes.Axes.set_autoscale_on

Axes.set_autoscale_on(*self*, *b*)
Set whether autoscaling is applied on plot commands

Parameters

b

[bool]

Examples using matplotlib.axes.Axes.set_autoscale_on

- sphx_glr_gallery_event_handling_resample.py

matplotlib.axes.Axes.get_autoscale_on

Axes.get_autoscale_on(*self*)
Get whether autoscaling is applied for both axes on plot commands

Examples using matplotlib.axes.Axes.get_autoscale_on

matplotlib.axes.Axes.set_autoscalex_on

Axes.set_autoscalex_on(*self*, *b*)
Set whether autoscaling for the x-axis is applied on plot commands

Parameters

b

[bool]

Examples using matplotlib.axes.Axes.set_autoscalex_on

matplotlib.axes.Axes.get_autoscalex_on

Axes.get_autoscalex_on(*self*)
Get whether autoscaling for the x-axis is applied on plot commands

Examples using `matplotlib.axes.Axes.get_autoscalex_on`

matplotlib.axes.Axes.set_autoscaley_on

`Axes.set_autoscaley_on(`*self*, *b*`)`

Set whether autoscaling for the y-axis is applied on plot commands

Parameters

b

[bool]

Examples using `matplotlib.axes.Axes.set_autoscaley_on`

matplotlib.axes.Axes.get_autoscaley_on

`Axes.get_autoscaley_on(`*self*`)`

Get whether autoscaling for the y-axis is applied on plot commands

Examples using `matplotlib.axes.Axes.get_autoscaley_on`

Aspect ratio

Axes.apply_aspect	Adjust the Axes for a specified data aspect ratio.
Axes.set_aspect	Set the aspect of the axis scaling, i.e.
Axes.get_aspect	
Axes.set_box_aspect	Set the axes box aspect.
Axes.get_box_aspect	Get the axes box aspect.
Axes.set_adjustable	Set how the Axes adjusts to achieve the required aspect ratio.
Axes.get_adjustable	Return whether the Axes will adjust its physical dimension ('box') or its data limits ('datalim') to achieve the desired aspect ratio.

matplotlib.axes.Axes.apply_aspect

Axes.apply_aspect(*self*, *position=None*)

Adjust the Axes for a specified data aspect ratio.

Depending on *get_adjustable* this will modify either the Axes box (position) or the view limits. In the former case, *get_anchor* will affect the position.

See also:

matplotlib.axes.Axes.set_aspect

For a description of aspect ratio handling.

matplotlib.axes.Axes.set_adjustable

Set how the Axes adjusts to achieve the required aspect ratio.

matplotlib.axes.Axes.set_anchor

Set the position in case of extra space.

Notes

This is called automatically when each Axes is drawn. You may need to call it yourself if you need to update the Axes position and/or view limits before the Figure is drawn.

Examples using `matplotlib.axes.Axes.apply_aspect`

matplotlib.axes.Axes.set_aspect

Axes.set_aspect(*self*, *aspect*, *adjustable=None*, *anchor=None*, *share=False*)

Set the aspect of the axis scaling, i.e. the ratio of y-unit to x-unit.

Parameters

aspect

[{'auto'} or num] Possible values:

value	description
'auto'	automatic; fill the position rectangle with data.
num	a circle will be stretched such that the height is *num* times the width. 'equal' is a synonym for `aspect=1`, i.e. same scaling for x and y.

adjustable

[None or {'box', 'datalim'}, optional] If not `None`, this defines which parameter will be adjusted to meet the required aspect. See *set_adjustable* for further details.

anchor

[None or str or 2-tuple of float, optional] If not `None`, this defines where the Axes will be drawn if there is extra space due to aspect constraints. The most common way to to specify the anchor are abbreviations of cardinal directions:

value	description
'C'	centered
'SW'	lower left corner
'S'	middle of bottom edge
'SE'	lower right corner
etc.	

See *set_anchor* for further details.

share

[bool, default: False] If `True`, apply the settings to all shared Axes.

See also:

matplotlib.axes.Axes.set_adjustable

Set how the Axes adjusts to achieve the required aspect ratio.

matplotlib.axes.Axes.set_anchor

Set the position in case of extra space.

Examples using `matplotlib.axes.Axes.set_aspect`

- sphx_glr_gallery_lines_bars_and_markers_gradient_bar.py
- sphx_glr_gallery_images_contours_and_fields_plot_streamplot.py
- sphx_glr_gallery_images_contours_and_fields_tricontour_demo.py
- sphx_glr_gallery_images_contours_and_fields_tricontour_smooth_delaunay.py
- sphx_glr_gallery_images_contours_and_fields_tricontour_smooth_user.py
- sphx_glr_gallery_images_contours_and_fields_trigradient_demo.py
- sphx_glr_gallery_images_contours_and_fields_tripcolor_demo.py
- sphx_glr_gallery_images_contours_and_fields_triplot_demo.py
- sphx_glr_gallery_subplots_axes_and_figures_axes_box_aspect.py

- sphx_glr_gallery_subplots_axes_and_figures_axes_margins.py
- sphx_glr_gallery_text_labels_and_annotations_multiline.py
- sphx_glr_gallery_shapes_and_collections_donut.py
- sphx_glr_gallery_axes_grid1_inset_locator_demo2.py
- sphx_glr_gallery_axes_grid1_scatter_hist_locatable_axes.py
- sphx_glr_gallery_axes_grid1_simple_anchored_artists.py
- sphx_glr_gallery_axisartist_demo_axis_direction.py
- sphx_glr_gallery_axisartist_simple_axis_pad.py
- *The double pendulum problem*
- sphx_glr_gallery_misc_anchored_artists.py
- sphx_glr_gallery_misc_rasterization_demo.py
- sphx_glr_gallery_scales_aspect_loglog.py
- sphx_glr_gallery_userdemo_annotate_text_arrow.py
- *Transformations Tutorial*
- *Colormap Normalization*

matplotlib.axes.Axes.get_aspect

`Axes.get_aspect(`*self*`)`

Examples using `matplotlib.axes.Axes.get_aspect`

matplotlib.axes.Axes.set_box_aspect

`Axes.set_box_aspect(`*self*, *aspect=None*`)`

Set the axes box aspect. The box aspect is the ratio of the axes height to the axes width in physical units. This is not to be confused with the data aspect, set via *set_aspect*.

Parameters

aspect

[None, or a number] Changes the physical dimensions of the Axes, such that the ratio of the axes height to the axes width in physical units is equal to *aspect*. If *None*, the axes geometry will not be adjusted.

Note that calling this function with a number changes the *adjustable*

to *datalim*.

See also:

matplotlib.axes.Axes.set_aspect

for a description of aspect handling.

Examples using `matplotlib.axes.Axes.set_box_aspect`

- sphx_glr_gallery_subplots_axes_and_figures_axes_box_aspect.py

matplotlib.axes.Axes.get_box_aspect

`Axes.get_box_aspect(`*self*`)`

Get the axes box aspect. Will be `None` if not explicitly specified.

See also:

matplotlib.axes.Axes.set_box_aspect

for a description of box aspect.

matplotlib.axes.Axes.set_aspect

for a description of aspect handling.

Examples using `matplotlib.axes.Axes.get_box_aspect`

matplotlib.axes.Axes.set_adjustable

`Axes.set_adjustable(`*self*, *adjustable*, *share=False*`)`

Set how the Axes adjusts to achieve the required aspect ratio.

Parameters

adjustable

[{'box', 'datalim'}] If 'box', change the physical dimensions of the Axes. If 'datalim', change the `x` or `y` data limits.

share

[bool, default: False] If `True`, apply the settings to all shared Axes.

See also:

matplotlib.axes.Axes.set_aspect

For a description of aspect handling.

Notes

Shared Axes (of which twinned Axes are a special case) impose restrictions on how aspect ratios can be imposed. For twinned Axes, use 'datalim'. For Axes that share both x and y, use 'box'. Otherwise, either 'datalim' or 'box' may be used. These limitations are partly a requirement to avoid over-specification, and partly a result of the particular implementation we are currently using, in which the adjustments for aspect ratios are done sequentially and independently on each Axes as it is drawn.

Examples using `matplotlib.axes.Axes.set_adjustable`

- sphx_glr_gallery_scales_aspect_loglog.py

matplotlib.axes.Axes.get_adjustable

`Axes.get_adjustable`(*self*)

Return whether the Axes will adjust its physical dimension ('box') or its data limits ('datalim') to achieve the desired aspect ratio.

See also:

matplotlib.axes.Axes.set_adjustable

Set how the Axes adjusts to achieve the required aspect ratio.

matplotlib.axes.Axes.set_aspect

For a description of aspect handling.

Examples using `matplotlib.axes.Axes.get_adjustable`

Ticks and tick labels

Axes.set_xticks	Set the xaxis' tick locations.
Axes.get_xticks	Return the xaxis' tick locations in data coordinates.
Axes.set_xticklabels	Set the xaxis' labels with list of string labels.
Axes.get_xticklabels	Get the xaxis' tick labels.

continues on next page

Table 104 – continued from previous page

Axes.get_xmajorticklabels	Return the xaxis' major tick labels, as a list of *Text*.
Axes.get_xminorticklabels	Return the xaxis' minor tick labels, as a list of *Text*.
Axes.get_xgridlines	Return the xaxis' grid lines as a list of *Line2D*s.
Axes.get_xticklines	Return the xaxis' tick lines as a list of *Line2D*s.
Axes.xaxis_date	Sets up axis ticks and labels to treat data along the xaxis as dates.
Axes.set_yticks	Set the yaxis' tick locations.
Axes.get_yticks	Return the yaxis' tick locations in data coordinates.
Axes.set_yticklabels	Set the yaxis' labels with list of string labels.
Axes.get_yticklabels	Get the yaxis' tick labels.
Axes.get_ymajorticklabels	Return the yaxis' major tick labels, as a list of *Text*.
Axes.get_yminorticklabels	Return the yaxis' minor tick labels, as a list of *Text*.
Axes.get_ygridlines	Return the yaxis' grid lines as a list of *Line2D*s.
Axes.get_yticklines	Return the yaxis' tick lines as a list of *Line2D*s.
Axes.yaxis_date	Sets up axis ticks and labels to treat data along the yaxis as dates.
Axes.minorticks_off	Remove minor ticks from the axes.
Axes.minorticks_on	Display minor ticks on the axes.
Axes.ticklabel_format	Configure the *ScalarFormatter* used by default for linear axes.
Axes.tick_params	Change the appearance of ticks, tick labels, and gridlines.
Axes.locator_params	Control behavior of major tick locators.

matplotlib.axes.Axes.set_xticks

`Axes.set_xticks`(*self*, *ticks*, *, *minor=False*)

Set the xaxis' tick locations.

Parameters

ticks

[list of floats] List of tick locations.

minor

[bool, default: False] If `False`, set the major ticks; if `True`, the minor ticks.

Examples using `matplotlib.axes.Axes.set_xticks`

- sphx_glr_gallery_lines_bars_and_markers_barchart.py
- sphx_glr_gallery_lines_bars_and_markers_psd_demo.py
- sphx_glr_gallery_images_contours_and_fields_image_annotated_heatmap.py
- sphx_glr_gallery_statistics_boxplot_vs_violin.py
- sphx_glr_gallery_statistics_customized_violin.py
- sphx_glr_gallery_statistics_multiple_histograms_side_by_side.py
- sphx_glr_gallery_text_labels_and_annotations_tex_demo.py
- sphx_glr_gallery_shapes_and_collections_hatch_demo.py
- sphx_glr_gallery_style_sheets_ggplot.py
- sphx_glr_gallery_axes_grid1_scatter_hist_locatable_axes.py
- sphx_glr_gallery_axes_grid1_simple_axisline4.py
- sphx_glr_gallery_axisartist_demo_ticklabel_alignment.py
- sphx_glr_gallery_axisartist_demo_ticklabel_direction.py
- sphx_glr_gallery_showcase_bachelors_degrees_by_gender.py
- sphx_glr_gallery_showcase_firefox.py
- sphx_glr_gallery_showcase_integral.py
- sphx_glr_gallery_showcase_mandelbrot.py
- sphx_glr_gallery_showcase_xkcd.py
- *Rain simulation*
- *MATPLOTLIB UNCHAINED*
- sphx_glr_gallery_frontpage_3D.py
- sphx_glr_gallery_frontpage_contour.py
- sphx_glr_gallery_frontpage_histogram.py
- sphx_glr_gallery_frontpage_membrane.py
- sphx_glr_gallery_scales_log_bar.py
- sphx_glr_gallery_specialty_plots_mri_with_eeg.py
- sphx_glr_gallery_ticks_and_spines_spines_bounds.py
- sphx_glr_gallery_units_bar_unit_demo.py

- *The Lifecycle of a Plot*

matplotlib.axes.Axes.get_xticks

Axes.get_xticks(*self*, *, *minor=False*)
Return the xaxis' tick locations in data coordinates.

Examples using matplotlib.axes.Axes.get_xticks

matplotlib.axes.Axes.set_xticklabels

Axes.set_xticklabels(*self*, *labels*, *, *fontdict=None*, *minor=False*, ***kwargs*)
Set the xaxis' labels with list of string labels.

Warning: This method should only be used after fixing the tick positions using *Axes.set_xticks*. Otherwise, the labels may end up in unexpected positions.

Parameters

labels

[list of str] The label texts.

fontdict

[dict, optional] A dictionary controlling the appearance of the ticklabels. The default *fontdict* is:

```
{'fontsize': rcParams['axes.titlesize'],
 'fontweight': rcParams['axes.titleweight'],
 'verticalalignment': 'baseline',
 'horizontalalignment': loc}
```

minor

[bool, default: False] Whether to set the minor ticklabels rather than the major ones.

Returns

list of *Text*

The labels.

Other Parameters

****kwargs**

[*Text* properties.]

Examples using `matplotlib.axes.Axes.set_xticklabels`

- sphx_glr_gallery_lines_bars_and_markers_barchart.py
- sphx_glr_gallery_images_contours_and_fields_image_annotated_heatmap.py
- sphx_glr_gallery_subplots_axes_and_figures_multiple_figs_demo.py
- sphx_glr_gallery_subplots_axes_and_figures_zoom_inset_axes.py
- sphx_glr_gallery_statistics_boxplot_demo.py
- sphx_glr_gallery_statistics_customized_violin.py
- sphx_glr_gallery_statistics_multiple_histograms_side_by_side.py
- sphx_glr_gallery_text_labels_and_annotations_tex_demo.py
- sphx_glr_gallery_style_sheets_ggplot.py
- sphx_glr_gallery_axes_grid1_simple_axisline4.py
- sphx_glr_gallery_axisartist_demo_ticklabel_alignment.py
- sphx_glr_gallery_showcase_integral.py
- sphx_glr_gallery_showcase_xkcd.py
- sphx_glr_gallery_scales_log_bar.py
- sphx_glr_gallery_ticks_and_spines_colorbar_tick_labelling_demo.py
- sphx_glr_gallery_ticks_and_spines_spines_bounds.py
- sphx_glr_gallery_units_bar_unit_demo.py
- *Constrained Layout Guide*

matplotlib.axes.Axes.get_xticklabels

`Axes.get_xticklabels`(*self*, *minor=False*, *which=None*)

Get the xaxis' tick labels.

Parameters

minor

[bool] Whether to return the minor or the major ticklabels.

which

[None, ('minor', 'major', 'both')] Overrides *minor*.

Selects which ticklabels to return

Returns

list of *Text*

Notes

The tick label strings are not populated until a `draw` method has been called.

See also: *draw* and *draw*.

Examples using `matplotlib.axes.Axes.get_xticklabels`

- sphx_glr_gallery_lines_bars_and_markers_timeline.py
- sphx_glr_gallery_images_contours_and_fields_image_annotated_heatmap.py
- sphx_glr_gallery_subplots_axes_and_figures_align_labels_demo.py
- sphx_glr_gallery_subplots_axes_and_figures_shared_axis_demo.py
- sphx_glr_gallery_statistics_boxplot_demo.py
- sphx_glr_gallery_axes_grid1_inset_locator_demo2.py
- sphx_glr_gallery_ticks_and_spines_date_concise_formatter.py
- sphx_glr_gallery_units_evans_test.py
- *The Lifecycle of a Plot*

matplotlib.axes.Axes.get_xmajorticklabels

`Axes.get_xmajorticklabels(`*self*`)`

Return the xaxis' major tick labels, as a list of *Text*.

Examples using `matplotlib.axes.Axes.get_xmajorticklabels`

matplotlib.axes.Axes.get_xminorticklabels

`Axes.get_xminorticklabels`(*self*)
Return the xaxis' minor tick labels, as a list of *Text*.

Examples using `matplotlib.axes.Axes.get_xminorticklabels`

matplotlib.axes.Axes.get_xgridlines

`Axes.get_xgridlines`(*self*)
Return the xaxis' grid lines as a list of *Line2D*s.

Examples using `matplotlib.axes.Axes.get_xgridlines`

matplotlib.axes.Axes.get_xticklines

`Axes.get_xticklines`(*self*, *minor=False*)
Return the xaxis' tick lines as a list of *Line2D*s.

Examples using `matplotlib.axes.Axes.get_xticklines`

matplotlib.axes.Axes.xaxis_date

`Axes.xaxis_date`(*self*, *tz=None*)
Sets up axis ticks and labels to treat data along the xaxis as dates.

Parameters

tz

[str or `datetime.tzinfo`, default: `rcParams["timezone"]` (default: `'UTC'`)] The timezone used to create date labels.

Examples using `matplotlib.axes.Axes.xaxis_date`

matplotlib.axes.Axes.set_yticks

`Axes.set_yticks`(*self*, *ticks*, *, *minor=False*)

Set the yaxis' tick locations.

Parameters

ticks

[list of floats] List of tick locations.

minor

[bool, default: False] If `False`, set the major ticks; if `True`, the minor ticks.

Examples using `matplotlib.axes.Axes.set_yticks`

- sphx_glr_gallery_lines_bars_and_markers_barh.py
- sphx_glr_gallery_lines_bars_and_markers_broken_barh.py
- sphx_glr_gallery_lines_bars_and_markers_psd_demo.py
- sphx_glr_gallery_images_contours_and_fields_image_annotated_heatmap.py
- sphx_glr_gallery_text_labels_and_annotations_tex_demo.py
- sphx_glr_gallery_pyplots_auto_subplots_adjust.py
- sphx_glr_gallery_axes_grid1_make_room_for_ylabel_using_axesgrid.py
- sphx_glr_gallery_axes_grid1_scatter_hist_locatable_axes.py
- sphx_glr_gallery_axisartist_demo_ticklabel_alignment.py
- sphx_glr_gallery_axisartist_demo_ticklabel_direction.py
- sphx_glr_gallery_showcase_bachelors_degrees_by_gender.py
- sphx_glr_gallery_showcase_firefox.py
- sphx_glr_gallery_showcase_integral.py
- sphx_glr_gallery_showcase_mandelbrot.py
- sphx_glr_gallery_showcase_xkcd.py
- *Rain simulation*
- *MATPLOTLIB UNCHAINED*
- sphx_glr_gallery_frontpage_3D.py
- sphx_glr_gallery_frontpage_contour.py

- sphx_glr_gallery_frontpage_histogram.py
- sphx_glr_gallery_frontpage_membrane.py
- sphx_glr_gallery_mplot3d_bars3d.py
- sphx_glr_gallery_specialty_plots_mri_with_eeg.py
- sphx_glr_gallery_specialty_plots_skewt.py
- sphx_glr_gallery_ticks_and_spines_spines_bounds.py

matplotlib.axes.Axes.get_yticks

Axes.get_yticks(*self*, *, *minor=False*)
Return the yaxis' tick locations in data coordinates.

Examples using matplotlib.axes.Axes.get_yticks

matplotlib.axes.Axes.set_yticklabels

Axes.set_yticklabels(*self*, *labels*, *, *fontdict=None*, *minor=False*, ***kwargs*)
Set the yaxis' labels with list of string labels.

> **Warning:** This method should only be used after fixing the tick positions using *Axes.set_yticks*. Otherwise, the labels may end up in unexpected positions.

Parameters

labels

[list of str] The label texts.

fontdict

[dict, optional] A dictionary controlling the appearance of the ticklabels. The default *fontdict* is:

```
{'fontsize': rcParams['axes.titlesize'],
 'fontweight': rcParams['axes.titleweight'],
 'verticalalignment': 'baseline',
 'horizontalalignment': loc}
```

minor

[bool, default: False] Whether to set the minor ticklabels rather than the major ones.

Returns

list of *Text*

The labels.

Other Parameters

****kwargs**

[*Text* properties.]

Examples using `matplotlib.axes.Axes.set_yticklabels`

- sphx_glr_gallery_lines_bars_and_markers_barh.py
- sphx_glr_gallery_lines_bars_and_markers_broken_barh.py
- sphx_glr_gallery_images_contours_and_fields_image_annotated_heatmap.py
- sphx_glr_gallery_subplots_axes_and_figures_zoom_inset_axes.py
- sphx_glr_gallery_statistics_boxplot.py
- sphx_glr_gallery_statistics_bxp.py
- sphx_glr_gallery_statistics_violinplot.py
- sphx_glr_gallery_text_labels_and_annotations_tex_demo.py
- sphx_glr_gallery_pyplots_auto_subplots_adjust.py
- sphx_glr_gallery_axes_grid1_make_room_for_ylabel_using_axesgrid.py
- sphx_glr_gallery_axisartist_demo_ticklabel_alignment.py
- sphx_glr_gallery_specialty_plots_mri_with_eeg.py
- sphx_glr_gallery_ticks_and_spines_colorbar_tick_labelling_demo.py
- *Constrained Layout Guide*

matplotlib.axes.Axes.get_yticklabels

`Axes.get_yticklabels(`*self*, *minor=False*, *which=None*`)`

Get the yaxis' tick labels.

Parameters

minor

[bool] Whether to return the minor or the major ticklabels.

which

[None, ('minor', 'major', 'both')] Overrides *minor*.

Selects which ticklabels to return

Returns

list of *Text*

Notes

The tick label strings are not populated until a `draw` method has been called.

See also: *draw* and *draw*.

Examples using `matplotlib.axes.Axes.get_yticklabels`

- sphx_glr_gallery_axes_grid1_inset_locator_demo2.py
- sphx_glr_gallery_recipes_fill_between_alpha.py

matplotlib.axes.Axes.get_ymajorticklabels

Axes.**get_ymajorticklabels**(*self*)
Return the yaxis' major tick labels, as a list of *Text*.

Examples using `matplotlib.axes.Axes.get_ymajorticklabels`

matplotlib.axes.Axes.get_yminorticklabels

Axes.**get_yminorticklabels**(*self*)
Return the yaxis' minor tick labels, as a list of *Text*.

Examples using `matplotlib.axes.Axes.get_yminorticklabels`

matplotlib.axes.Axes.get_ygridlines

Axes.**get_ygridlines**(*self*)
Return the yaxis' grid lines as a list of *Line2D*s.

Examples using `matplotlib.axes.Axes.get_ygridlines`

matplotlib.axes.Axes.get_yticklines

Axes.**get_yticklines**(*self*, *minor=False*)
Return the yaxis' tick lines as a list of *Line2D*s.

Examples using `matplotlib.axes.Axes.get_yticklines`

matplotlib.axes.Axes.yaxis_date

Axes.**yaxis_date**(*self*, *tz=None*)
Sets up axis ticks and labels to treat data along the yaxis as dates.

Parameters

tz

[str or `datetime.tzinfo`, default: `rcParams["timezone"]` (default: `'UTC'`)] The timezone used to create date labels.

Examples using `matplotlib.axes.Axes.yaxis_date`

matplotlib.axes.Axes.minorticks_off

Axes.**minorticks_off**(*self*)
Remove minor ticks from the axes.

Examples using `matplotlib.axes.Axes.minorticks_off`

matplotlib.axes.Axes.minorticks_on

Axes.**minorticks_on**(*self*)
Display minor ticks on the axes.

Displaying minor ticks may reduce performance; you may turn them off using *minorticks_off()* if drawing speed is a problem.

Examples using `matplotlib.axes.Axes.minorticks_on`

- sphx_glr_gallery_specialty_plots_mri_with_eeg.py

matplotlib.axes.Axes.ticklabel_format

Axes.ticklabel_format(*self*, ***, *axis='both'*, *style=''*, *scilimits=None*, *useOffset=None*, *useLocale=None*, *useMathText=None*)

Configure the *ScalarFormatter* used by default for linear axes.

If a parameter is not set, the corresponding property of the formatter is left unchanged.

Parameters

axis

[{'x', 'y', 'both'}, default: 'both'] The axes to configure. Only major ticks are affected.

style

[{'sci', 'scientific', 'plain'}] Whether to use scientific notation. The formatter default is to use scientific notation.

scilimits

[pair of ints (m, n)] Scientific notation is used only for numbers outside the range 10^m to 10^n (and only if the formatter is configured to use scientific notation at all). Use (0, 0) to include all numbers. Use (m, m) where m != 0 to fix the order of magnitude to 10^m. The formatter default is `rcParams["axes.formatter.limits"]` (default: `[-5, 6]`).

useOffset

[bool or float] If True, the offset is calculated as needed. If False, no offset is used. If a numeric value, it sets the offset. The formatter default is `rcParams["axes.formatter.useoffset"]` (default: `True`).

useLocale

[bool] Whether to format the number using the current locale or using the C (English) locale. This affects e.g. the decimal separator. The formatter default is `rcParams["axes.formatter.use_locale"]` (default: `False`).

useMathText

[bool] Render the offset and scientific notation in mathtext. The formatter default is `rcParams["axes.formatter.use_mathtext"]` (default: `False`).

Raises

AttributeError

If the current formatter is not a *ScalarFormatter*.

Examples using `matplotlib.axes.Axes.ticklabel_format`

- sphx_glr_gallery_ticks_and_spines_scalarformatter.py

matplotlib.axes.Axes.tick_params

`Axes.tick_params`(*self*, *axis='both'*, ***kwargs*)

Change the appearance of ticks, tick labels, and gridlines.

Tick properties that are not explicitly set using the keyword arguments remain unchanged unless *reset* is True.

Parameters

axis

[{'x', 'y', 'both'}, default: 'both'] The axis to which the parameters are applied.

which

[{'major', 'minor', 'both'}, default: 'major'] The group of ticks to which the parameters are applied.

reset

[bool, default: False] Whether to reset the ticks to defaults before updating them.

Other Parameters

direction

[{'in', 'out', 'inout'}] Puts ticks inside the axes, outside the axes, or both.

length

[float] Tick length in points.

width

[float] Tick width in points.

color

[color] Tick color.

pad

[float] Distance in points between tick and label.

labelsize

[float or str] Tick label font size in points or as a string (e.g., 'large').

labelcolor

[color] Tick label color.

colors

[color] Tick color and label color.

zorder

[float] Tick and label zorder.

bottom, top, left, right

[bool] Whether to draw the respective ticks.

labelbottom, labeltop, labelleft, labelright

[bool] Whether to draw the respective tick labels.

labelrotation

[float] Tick label rotation

grid_color

[color] Gridline color.

grid_alpha

[float] Transparency of gridlines: 0 (transparent) to 1 (opaque).

grid_linewidth

[float] Width of gridlines in points.

grid_linestyle

[str] Any valid *Line2D* line style spec.

Examples

```
ax.tick_params(direction='out', length=6, width=2, colors='r',
               grid_color='r', grid_alpha=0.5)
```

This will make all major ticks be red, pointing out of the box, and with dimensions 6 points by 2 points. Tick labels will also be red. Gridlines will be red and translucent.

Examples using `matplotlib.axes.Axes.tick_params`

- sphx_glr_gallery_lines_bars_and_markers_scatter_hist.py
- sphx_glr_gallery_images_contours_and_fields_image_annotated_heatmap.py
- sphx_glr_gallery_subplots_axes_and_figures_axes_props.py
- sphx_glr_gallery_subplots_axes_and_figures_broken_axis.py
- sphx_glr_gallery_subplots_axes_and_figures_two_scales.py
- sphx_glr_gallery_pie_and_polar_charts_polar_legend.py
- sphx_glr_gallery_color_color_demo.py
- sphx_glr_gallery_axes_grid1_inset_locator_demo.py
- sphx_glr_gallery_axes_grid1_make_room_for_ylabel_using_axesgrid.py
- sphx_glr_gallery_axes_grid1_simple_axes_divider1.py
- sphx_glr_gallery_axes_grid1_simple_axes_divider3.py
- sphx_glr_gallery_showcase_anatomy.py
- sphx_glr_gallery_showcase_bachelors_degrees_by_gender.py
- sphx_glr_gallery_specialty_plots_anscombe.py
- sphx_glr_gallery_ticks_and_spines_major_minor_demo.py
- sphx_glr_gallery_ticks_and_spines_multiple_yaxis_with_spines.py
- *Text in Matplotlib Plots*

matplotlib.axes.Axes.locator_params

Axes.locator_params(*self*, *axis='both'*, *tight=None*, ***kwargs*)

Control behavior of major tick locators.

Because the locator is involved in autoscaling, *autoscale_view* is called automatically after the parameters are changed.

Parameters

axis

[{'both', 'x', 'y'}, default: 'both'] The axis on which to operate.

tight

[bool or None, optional] Parameter passed to *autoscale_view*. Default is None, for no change.

Other Parameters

****kwargs**

Remaining keyword arguments are passed to directly to the `set_params()` method of the locator. Supported keywords depend on the type of the locator. See for example *set_params* for the *ticker.MaxNLocator* used by default for linear axes.

Examples

When plotting small subplots, one might want to reduce the maximum number of ticks and use tight bounds, for example:

```
ax.locator_params(tight=True, nbins=4)
```

Examples using `matplotlib.axes.Axes.locator_params`

- sphx_glr_gallery_images_contours_and_fields_contourf_demo.py
- *Constrained Layout Guide*
- *Tight Layout guide*

18.5.9 Units

Axes.convert_xunits	Convert *x* using the unit type of the xaxis.
Axes.convert_yunits	Convert *y* using the unit type of the yaxis.
Axes.have_units	Return *True* if units are set on any axis.

matplotlib.axes.Axes.convert_xunits

`Axes.convert_xunits`(*self*, *x*)

Convert *x* using the unit type of the xaxis.

If the artist is not in contained in an Axes or if the xaxis does not have units, *x* itself is returned.

Examples using `matplotlib.axes.Axes.convert_xunits`

matplotlib.axes.Axes.convert_yunits

`Axes.convert_yunits`(*self*, *y*)
Convert *y* using the unit type of the yaxis.

If the artist is not in contained in an Axes or if the yaxis does not have units, *y* itself is returned.

Examples using `matplotlib.axes.Axes.convert_yunits`

matplotlib.axes.Axes.have_units

`Axes.have_units`(*self*)
Return *True* if units are set on any axis.

Examples using `matplotlib.axes.Axes.have_units`

18.5.10 Adding Artists

Axes.add_artist	Add an *Artist* to the axes, and return the artist.
Axes.add_child_axes	Add an `AxesBase` to the axes' children; return the child axes.
Axes.add_collection	Add a *Collection* to the axes' collections; return the collection.
Axes.add_container	Add a *Container* to the axes' containers; return the container.
Axes.add_image	Add an *AxesImage* to the axes' images; return the image.
Axes.add_line	Add a *Line2D* to the axes' lines; return the line.
Axes.add_patch	Add a *Patch* to the axes' patches; return the patch.
Axes.add_table	Add a *Table* to the axes' tables; return the table.

matplotlib.axes.Axes.add_artist

Axes.add_artist(*self*, *a*)

Add an *Artist* to the axes, and return the artist.

Use *add_artist* only for artists for which there is no dedicated "add" method; and if necessary, use a method such as *update_datalim* to manually update the dataLim if the artist is to be included in autoscaling.

If no `transform` has been specified when creating the artist (e.g. `artist.get_transform() == None`) then the transform is set to `ax.transData`.

Examples using `matplotlib.axes.Axes.add_artist`

- sphx_glr_gallery_lines_bars_and_markers_scatter_with_legend.py
- sphx_glr_gallery_images_contours_and_fields_demo_bboximage.py
- sphx_glr_gallery_pie_and_polar_charts_bar_of_pie.py
- sphx_glr_gallery_text_labels_and_annotations_annotation_demo.py
- sphx_glr_gallery_text_labels_and_annotations_demo_annotation_box.py
- sphx_glr_gallery_text_labels_and_annotations_demo_text_path.py
- sphx_glr_gallery_shapes_and_collections_ellipse_demo.py
- sphx_glr_gallery_axes_grid1_demo_anchored_direction_arrows.py
- sphx_glr_gallery_axes_grid1_demo_axes_grid2.py
- sphx_glr_gallery_axes_grid1_inset_locator_demo2.py
- sphx_glr_gallery_axes_grid1_simple_anchored_artists.py
- sphx_glr_gallery_showcase_anatomy.py
- sphx_glr_gallery_misc_anchored_artists.py
- sphx_glr_gallery_units_artist_tests.py
- sphx_glr_gallery_userdemo_anchored_box01.py
- sphx_glr_gallery_userdemo_anchored_box02.py
- sphx_glr_gallery_userdemo_anchored_box03.py
- sphx_glr_gallery_userdemo_anchored_box04.py
- sphx_glr_gallery_userdemo_annotate_explain.py
- sphx_glr_gallery_userdemo_connect_simple01.py
- sphx_glr_gallery_userdemo_simple_annotate01.py
- sphx_glr_gallery_userdemo_simple_legend02.py

matplotlib.axes.Axes.add_child_axes

Axes.add_child_axes(*self*, *ax*)
Add an AxesBase to the axes' children; return the child axes.

This is the lowlevel version. See *axes.Axes.inset_axes*.

Examples using matplotlib.axes.Axes.add_child_axes

matplotlib.axes.Axes.add_collection

Axes.add_collection(*self*, *collection*, *autolim=True*)
Add a *Collection* to the axes' collections; return the collection.

Examples using matplotlib.axes.Axes.add_collection

- sphx_glr_gallery_lines_bars_and_markers_eventcollection_demo.py
- sphx_glr_gallery_lines_bars_and_markers_span_regions.py
- sphx_glr_gallery_statistics_errorbars_and_boxes.py
- sphx_glr_gallery_shapes_and_collections_artist_reference.py
- sphx_glr_gallery_shapes_and_collections_collections.py
- sphx_glr_gallery_shapes_and_collections_ellipse_collection.py
- sphx_glr_gallery_shapes_and_collections_line_collection.py
- sphx_glr_gallery_shapes_and_collections_patch_collection.py
- sphx_glr_gallery_event_handling_lasso_demo.py
- sphx_glr_gallery_specialty_plots_mri_with_eeg.py
- sphx_glr_gallery_units_artist_tests.py

matplotlib.axes.Axes.add_container

Axes.add_container(*self*, *container*)
Add a *Container* to the axes' containers; return the container.

Examples using `matplotlib.axes.Axes.add_container`

matplotlib.axes.Axes.add_image

Axes.add_image(*self*, *image*)
Add an *AxesImage* to the axes' images; return the image.

Examples using `matplotlib.axes.Axes.add_image`

matplotlib.axes.Axes.add_line

Axes.add_line(*self*, *line*)
Add a *Line2D* to the axes' lines; return the line.

Examples using `matplotlib.axes.Axes.add_line`

- sphx_glr_gallery_text_labels_and_annotations_line_with_text.py
- sphx_glr_gallery_shapes_and_collections_artist_reference.py
- sphx_glr_gallery_units_artist_tests.py
- *Artist tutorial*

matplotlib.axes.Axes.add_patch

Axes.add_patch(*self*, *p*)
Add a *Patch* to the axes' patches; return the patch.

Examples using `matplotlib.axes.Axes.add_patch`

- sphx_glr_gallery_lines_bars_and_markers_curve_error_band.py
- sphx_glr_gallery_images_contours_and_fields_image_demo.py
- sphx_glr_gallery_subplots_axes_and_figures_axes_box_aspect.py
- sphx_glr_gallery_subplots_axes_and_figures_axes_margins.py
- sphx_glr_gallery_subplots_axes_and_figures_axes_zoom_effect.py
- sphx_glr_gallery_statistics_boxplot_demo.py
- sphx_glr_gallery_statistics_confidence_ellipse.py
- sphx_glr_gallery_text_labels_and_annotations_annotation_demo.py
- sphx_glr_gallery_text_labels_and_annotations_fancyarrow_demo.py

- sphx_glr_gallery_text_labels_and_annotations_text_alignment.py
- sphx_glr_gallery_shapes_and_collections_compound_path.py
- sphx_glr_gallery_shapes_and_collections_dolphin.py
- sphx_glr_gallery_shapes_and_collections_donut.py
- sphx_glr_gallery_shapes_and_collections_fancybox_demo.py
- sphx_glr_gallery_shapes_and_collections_hatch_demo.py
- sphx_glr_gallery_shapes_and_collections_path_patch.py
- sphx_glr_gallery_shapes_and_collections_quad_bezier.py
- sphx_glr_gallery_style_sheets_ggplot.py
- sphx_glr_gallery_axes_grid1_inset_locator_demo.py
- sphx_glr_gallery_showcase_firefox.py
- sphx_glr_gallery_showcase_integral.py
- *Animated histogram*
- sphx_glr_gallery_event_handling_looking_glass.py
- sphx_glr_gallery_event_handling_path_editor.py
- sphx_glr_gallery_event_handling_poly_editor.py
- sphx_glr_gallery_event_handling_viewlims.py
- sphx_glr_gallery_misc_bbox_intersect.py
- sphx_glr_gallery_misc_histogram_path.py
- sphx_glr_gallery_misc_svg_filter_pie.py
- sphx_glr_gallery_mplot3d_pathpatch3d.py
- sphx_glr_gallery_units_artist_tests.py
- sphx_glr_gallery_units_ellipse_with_units.py
- *Artist tutorial*
- *Path Tutorial*
- *Transformations Tutorial*
- *Specifying Colors*
- *Text properties and layout*

matplotlib.axes.Axes.add_table

Axes.add_table(*self*, *tab*)
Add a *Table* to the axes' tables; return the table.

Examples using matplotlib.axes.Axes.add_table

18.5.11 Twinning and sharing

Axes.twinx	Create a twin Axes sharing the xaxis.
Axes.twiny	Create a twin Axes sharing the yaxis.
Axes.sharex	Share the x-axis with *other*.
Axes.sharey	Share the y-axis with *other*.
Axes.get_shared_x_axes	Return a reference to the shared axes Grouper object for x axes.
Axes.get_shared_y_axes	Return a reference to the shared axes Grouper object for y axes.

matplotlib.axes.Axes.twinx

Axes.twinx(*self*)
Create a twin Axes sharing the xaxis.

Create a new Axes with an invisible x-axis and an independent y-axis positioned opposite to the original one (i.e. at right). The x-axis autoscale setting will be inherited from the original Axes. To ensure that the tick marks of both y-axes align, see *LinearLocator*.

Returns

Axes

The newly created Axes instance

Notes

For those who are 'picking' artists while using twinx, pick events are only called for the artists in the top-most axes.

Examples using `matplotlib.axes.Axes.twinx`

- sphx_glr_gallery_subplots_axes_and_figures_axes_box_aspect.py
- sphx_glr_gallery_subplots_axes_and_figures_two_scales.py
- sphx_glr_gallery_statistics_barchart_demo.py
- sphx_glr_gallery_axes_grid1_parasite_simple.py
- sphx_glr_gallery_axisartist_demo_parasite_axes2.py
- sphx_glr_gallery_ticks_and_spines_multiple_yaxis_with_spines.py

matplotlib.axes.Axes.twiny

`Axes.twiny`(*self*)

Create a twin Axes sharing the yaxis.

Create a new Axes with an invisible y-axis and an independent x-axis positioned opposite to the original one (i.e. at top). The y-axis autoscale setting will be inherited from the original Axes. To ensure that the tick marks of both x-axes align, see *LinearLocator*.

Returns

Axes

The newly created Axes instance

Notes

For those who are 'picking' artists while using twiny, pick events are only called for the artists in the top-most axes.

Examples using `matplotlib.axes.Axes.twiny`

- sphx_glr_gallery_subplots_axes_and_figures_two_scales.py

matplotlib.axes.Axes.sharex

Axes.**sharex**(*self*, *other*)
Share the x-axis with *other*.

This is equivalent to passing `sharex=other` when constructing the axes, and cannot be used if the x-axis is already being shared with another axes.

Examples using `matplotlib.axes.Axes.sharex`

matplotlib.axes.Axes.sharey

Axes.**sharey**(*self*, *other*)
Share the y-axis with *other*.

This is equivalent to passing `sharey=other` when constructing the axes, and cannot be used if the y-axis is already being shared with another axes.

Examples using `matplotlib.axes.Axes.sharey`

matplotlib.axes.Axes.get_shared_x_axes

Axes.**get_shared_x_axes**(*self*)
Return a reference to the shared axes Grouper object for x axes.

Examples using `matplotlib.axes.Axes.get_shared_x_axes`

matplotlib.axes.Axes.get_shared_y_axes

Axes.**get_shared_y_axes**(*self*)
Return a reference to the shared axes Grouper object for y axes.

Examples using `matplotlib.axes.Axes.get_shared_y_axes`

18.5.12 Axes Position

Axes.get_anchor	Get the anchor location.
Axes.set_anchor	Define the anchor location.
Axes.get_axes_locator	Return the axes_locator.
Axes.set_axes_locator	Set the axes locator.

continues on next page

Table 108 – continued from previous page

Axes.reset_position	Reset the active position to the original position.
Axes.get_position	Get a copy of the axes rectangle as a *Bbox*.
Axes.set_position	Set the axes position.

matplotlib.axes.Axes.get_anchor

Axes.get_anchor(*self*)

Get the anchor location.

See also:

matplotlib.axes.Axes.set_anchor

for a description of the anchor.

matplotlib.axes.Axes.set_aspect

for a description of aspect handling.

Examples using matplotlib.axes.Axes.get_anchor

matplotlib.axes.Axes.set_anchor

Axes.set_anchor(*self*, *anchor*, *share=False*)

Define the anchor location.

The actual drawing area (active position) of the Axes may be smaller than the Bbox (original position) when a fixed aspect is required. The anchor defines where the drawing area will be located within the available space.

Parameters

anchor

[2-tuple of floats or {'C', 'SW', 'S', 'SE', ...}] The anchor position may be either:

- a sequence (*cx*, *cy*). *cx* and *cy* may range from 0 to 1, where 0 is left or bottom and 1 is right or top.
- a string using cardinal directions as abbreviation:
  - 'C' for centered
  - 'S' (south) for bottom-center
  - 'SW' (south west) for bottom-left

- etc.

Here is an overview of the possible positions:

'NW'	'N'	'NE'
'W'	'C'	'E'
'SW'	'S'	'SE'

share

[bool, default: False] If `True`, apply the settings to all shared Axes.

See also:

matplotlib.axes.Axes.set_aspect

for a description of aspect handling.

Examples using `matplotlib.axes.Axes.set_anchor`

matplotlib.axes.Axes.get_axes_locator

`Axes.get_axes_locator`(*self*)

Return the axes_locator.

Examples using `matplotlib.axes.Axes.get_axes_locator`

matplotlib.axes.Axes.set_axes_locator

`Axes.set_axes_locator`(*self*, *locator*)

Set the axes locator.

Parameters

locator

[Callable[[Axes, Renderer], Bbox]]

Examples using `matplotlib.axes.Axes.set_axes_locator`

- sphx_glr_gallery_axes_grid1_demo_axes_hbox_divider.py

matplotlib.axes.Axes.reset_position

Axes.**reset_position**(*self*)
Reset the active position to the original position.

This resets the a possible position change due to aspect constraints. For an explanation of the positions see *set_position*.

Examples using `matplotlib.axes.Axes.reset_position`

matplotlib.axes.Axes.get_position

Axes.**get_position**(*self*, *original=False*)
Get a copy of the axes rectangle as a *Bbox*.

Parameters

original

[bool] If `True`, return the original position. Otherwise return the active position. For an explanation of the positions see *set_position*.

Returns

Bbox

Examples using `matplotlib.axes.Axes.get_position`

- sphx_glr_gallery_images_contours_and_fields_contour_demo.py
- *Choosing Colormaps in Matplotlib*

matplotlib.axes.Axes.set_position

Axes.set_position(*self*, *pos*, *which='both'*)

Set the axes position.

Axes have two position attributes. The 'original' position is the position allocated for the Axes. The 'active' position is the position the Axes is actually drawn at. These positions are usually the same unless a fixed aspect is set to the Axes. See *set_aspect* for details.

Parameters

pos

[[left, bottom, width, height] or *Bbox*] The new position of the in *Figure* coordinates.

which

[{'both', 'active', 'original'}, default: 'both'] Determines which position variables to change.

Examples using matplotlib.axes.Axes.set_position

- sphx_glr_gallery_images_contours_and_fields_contour_demo.py

18.5.13 Async/Event based

Axes.stale	Whether the artist is 'stale' and needs to be re-drawn for the output to match the internal state of the artist.
Axes.pchanged	Call all of the registered callbacks.
Axes.add_callback	Add a callback function that will be called whenever one of the *Artist*'s properties changes.
Axes.remove_callback	Remove a callback based on its observer id.

matplotlib.axes.Axes.stale

property Axes.stale
Whether the artist is 'stale' and needs to be re-drawn for the output to match the internal state of the artist.

matplotlib.axes.Axes.pchanged

Axes.pchanged(*self*)
Call all of the registered callbacks.

This function is triggered internally when a property is changed.

See also:

add_callback

remove_callback

Examples using matplotlib.axes.Axes.pchanged

matplotlib.axes.Axes.add_callback

Axes.add_callback(*self*, *func*)
Add a callback function that will be called whenever one of the *Artist*'s properties changes.

Parameters

func

[callable] The callback function. It must have the signature:

```
def func(artist: Artist) -> Any
```

where *artist* is the calling *Artist*. Return values may exist but are ignored.

Returns

int

The observer id associated with the callback. This id can be used for removing the callback with *remove_callback* later.

See also:

remove_callback

Examples using `matplotlib.axes.Axes.add_callback`

matplotlib.axes.Axes.remove_callback

`Axes.remove_callback`(*self*, *oid*)

Remove a callback based on its observer id.

See also:

add_callback

Examples using `matplotlib.axes.Axes.remove_callback`

18.5.14 Interactive

Axes.can_pan	Return *True* if this axes supports any pan/zoom button functionality.
Axes.can_zoom	Return *True* if this axes supports the zoom box button functionality.
Axes.get_navigate	Get whether the axes responds to navigation commands
Axes.set_navigate	Set whether the axes responds to navigation toolbar commands
Axes.get_navigate_mode	Get the navigation toolbar button status: 'PAN', 'ZOOM', or None
Axes.set_navigate_mode	Set the navigation toolbar button status;
Axes.start_pan	Called when a pan operation has started.
Axes.drag_pan	Called when the mouse moves during a pan operation.
Axes.end_pan	Called when a pan operation completes (when the mouse button is up.)
Axes.format_coord	Return a format string formatting the *x*, *y* coordinates.
Axes.format_cursor_data	Return a string representation of *data*.
Axes.format_xdata	Return *x* formatted as an x-value.
Axes.format_ydata	Return *y* formatted as an y-value.
Axes.mouseover	If this property is set to *True*, the artist will be queried for custom context information when the mouse cursor moves over it.
Axes.in_axes	Return *True* if the given *mouseevent* (in display coords) is in the Axes
Axes.pick	Process a pick event.
Axes.pickable	Return whether the artist is pickable.

continues on next page

Table 110 – continued from previous page

Axes.get_picker	Return the picking behavior of the artist.
Axes.set_picker	Define the picking behavior of the artist.
Axes.set_contains	[*Deprecated*] Define a custom contains test for the artist.
Axes.get_contains	[*Deprecated*] Return the custom contains function of the artist if set, or *None*.
Axes.contains	Test whether the artist contains the mouse event.
Axes.contains_point	Return whether *point* (pair of pixel coordinates) is inside the axes patch.
Axes.get_cursor_data	Return the cursor data for a given event.

matplotlib.axes.Axes.can_pan

Axes.**can_pan**(*self*)
Return *True* if this axes supports any pan/zoom button functionality.

Examples using matplotlib.axes.Axes.can_pan

matplotlib.axes.Axes.can_zoom

Axes.**can_zoom**(*self*)
Return *True* if this axes supports the zoom box button functionality.

Examples using matplotlib.axes.Axes.can_zoom

matplotlib.axes.Axes.get_navigate

Axes.**get_navigate**(*self*)
Get whether the axes responds to navigation commands

Examples using matplotlib.axes.Axes.get_navigate

matplotlib.axes.Axes.set_navigate

Axes.**set_navigate**(*self*, *b*)
Set whether the axes responds to navigation toolbar commands

Parameters

b

[bool]

Examples using `matplotlib.axes.Axes.set_navigate`

matplotlib.axes.Axes.get_navigate_mode

`Axes.get_navigate_mode(`*self*`)`
Get the navigation toolbar button status: 'PAN', 'ZOOM', or None

Examples using `matplotlib.axes.Axes.get_navigate_mode`

matplotlib.axes.Axes.set_navigate_mode

`Axes.set_navigate_mode(`*self*, *b*`)`
Set the navigation toolbar button status;

> **Warning:** this is not a user-API function.

Examples using `matplotlib.axes.Axes.set_navigate_mode`

matplotlib.axes.Axes.start_pan

`Axes.start_pan(`*self*, *x*, *y*, *button*`)`
Called when a pan operation has started.

Parameters

x, y

[float] The mouse coordinates in display coords.

button

[*MouseButton*] The pressed mouse button.

Notes

This is intended to be overridden by new projection types.

Examples using `matplotlib.axes.Axes.start_pan`

matplotlib.axes.Axes.drag_pan

Axes.drag_pan(*self*, *button*, *key*, *x*, *y*)
Called when the mouse moves during a pan operation.

Parameters

button

[*MouseButton*] The pressed mouse button.

key

[str or None] The pressed key, if any.

x, y

[float] The mouse coordinates in display coords.

Notes

This is intended to be overridden by new projection types.

Examples using `matplotlib.axes.Axes.drag_pan`

matplotlib.axes.Axes.end_pan

Axes.end_pan(*self*)
Called when a pan operation completes (when the mouse button is up.)

Notes

This is intended to be overridden by new projection types.

Examples using `matplotlib.axes.Axes.end_pan`

matplotlib.axes.Axes.format_coord

`Axes.format_coord`(*self*, *x*, *y*)
Return a format string formatting the *x*, *y* coordinates.

Examples using `matplotlib.axes.Axes.format_coord`

- sphx_glr_gallery_images_contours_and_fields_image_zcoord.py

matplotlib.axes.Axes.format_cursor_data

`Axes.format_cursor_data`(*self*, *data*)
Return a string representation of *data*.

Note: This method is intended to be overridden by artist subclasses. As an end-user of Matplotlib you will most likely not call this method yourself.

The default implementation converts ints and floats and arrays of ints and floats into a comma-separated string enclosed in square brackets.

See also:

get_cursor_data

Examples using `matplotlib.axes.Axes.format_cursor_data`

matplotlib.axes.Axes.format_xdata

`Axes.format_xdata`(*self*, *x*)
Return *x* formatted as an x-value.

This function will use the `fmt_xdata` attribute if it is not None, else will fall back on the xaxis major formatter.

Examples using `matplotlib.axes.Axes.format_xdata`

matplotlib.axes.Axes.format_ydata

`Axes.format_ydata`(*self*, *y*)

Return *y* formatted as an y-value.

This function will use the `fmt_ydata` attribute if it is not None, else will fall back on the yaxis major formatter.

Examples using `matplotlib.axes.Axes.format_ydata`

matplotlib.axes.Axes.mouseover

`property Axes.mouseover`

If this property is set to *True*, the artist will be queried for custom context information when the mouse cursor moves over it.

See also *get_cursor_data()*, *ToolCursorPosition* and *NavigationToolbar2*.

matplotlib.axes.Axes.in_axes

`Axes.in_axes`(*self*, *mouseevent*)

Return *True* if the given *mouseevent* (in display coords) is in the Axes

Examples using `matplotlib.axes.Axes.in_axes`

matplotlib.axes.Axes.pick

`Axes.pick`(*self*, *mouseevent*)

Process a pick event.

Each child artist will fire a pick event if *mouseevent* is over the artist and the artist has picker set.

See also:

set_picker, *get_picker*, *pickable*

Examples using `matplotlib.axes.Axes.pick`

matplotlib.axes.Axes.pickable

`Axes.pickable`(*self*)
Return whether the artist is pickable.

See also:

set_picker, *get_picker*, *pick*

Examples using `matplotlib.axes.Axes.pickable`

matplotlib.axes.Axes.get_picker

`Axes.get_picker`(*self*)
Return the picking behavior of the artist.

The possible values are described in *set_picker*.

See also:

set_picker, *pickable*, *pick*

Examples using `matplotlib.axes.Axes.get_picker`

matplotlib.axes.Axes.set_picker

`Axes.set_picker`(*self*, *picker*)
Define the picking behavior of the artist.

Parameters

picker

[None or bool or callable] This can be one of the following:

- *None*: Picking is disabled for this artist (default).
- A boolean: If *True* then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist.
- A function: If picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

to determine the hit test. if the mouse event is over the artist, return *hit=True* and props is a dictionary of properties you want added to the PickEvent attributes.

- *deprecated*: For *Line2D* only, *picker* can also be a float that sets the tolerance for checking whether an event occurred "on" the line; this is deprecated. Use *Line2D.set_pickradius* instead.

Examples using `matplotlib.axes.Axes.set_picker`

matplotlib.axes.Axes.set_contains

`Axes.set_contains`(*self*, *picker*)

[*Deprecated*] Define a custom contains test for the artist.

The provided callable replaces the default *contains* method of the artist.

Parameters

picker

[callable] A custom picker function to evaluate if an event is within the artist. The function must have the signature:

```
def contains(artist: Artist, event: MouseEvent) -> bool, dict
```

that returns:

- a bool indicating if the event is within the artist
- a dict of additional information. The dict should at least return the same information as the default `contains()` implementation of the respective artist, but may provide additional information.

Notes

Deprecated since version 3.3.

Examples using `matplotlib.axes.Axes.set_contains`

matplotlib.axes.Axes.get_contains

`Axes.get_contains`(*self*)

[*Deprecated*] Return the custom contains function of the artist if set, or *None*.

See also:

set_contains

Notes

Deprecated since version 3.3.

Examples using `matplotlib.axes.Axes.get_contains`

matplotlib.axes.Axes.contains

`Axes.contains`(*self*, *mouseevent*)
Test whether the artist contains the mouse event.

Parameters

mouseevent

[*matplotlib.backend_bases.MouseEvent*]

Returns

contains

[bool] Whether any values are within the radius.

details

[dict] An artist-specific dictionary of details of the event context, such as which points are contained in the pick radius. See the individual Artist subclasses for details.

Examples using `matplotlib.axes.Axes.contains`

matplotlib.axes.Axes.contains_point

`Axes.contains_point`(*self*, *point*)
Return whether *point* (pair of pixel coordinates) is inside the axes patch.

Examples using `matplotlib.axes.Axes.contains_point`

matplotlib.axes.Axes.get_cursor_data

Axes.get_cursor_data(*self*, *event*)

Return the cursor data for a given event.

Note: This method is intended to be overridden by artist subclasses. As an end-user of Matplotlib you will most likely not call this method yourself.

Cursor data can be used by Artists to provide additional context information for a given event. The default implementation just returns *None*.

Subclasses can override the method and return arbitrary data. However, when doing so, they must ensure that *format_cursor_data* can convert the data to a string representation.

The only current use case is displaying the z-value of an *AxesImage* in the status bar of a plot window, while moving the mouse.

Parameters

event

[*matplotlib.backend_bases.MouseEvent*]

See also:

format_cursor_data

Examples using `matplotlib.axes.Axes.get_cursor_data`

18.5.15 Children

Axes.get_children	Return a list of the child *Artist*s of this *Artist*.
Axes.get_images	Return a list of *AxesImage*s contained by the Axes.
Axes.get_lines	Return a list of lines contained by the Axes.
Axes.findobj	Find artist objects.

matplotlib.axes.Axes.get_children

Axes.get_children(*self*)
Return a list of the child *Artist*s of this *Artist*.

Examples using matplotlib.axes.Axes.get_children

matplotlib.axes.Axes.get_images

Axes.get_images(*self*)
Return a list of *AxesImage*s contained by the Axes.

Examples using matplotlib.axes.Axes.get_images

matplotlib.axes.Axes.get_lines

Axes.get_lines(*self*)
Return a list of lines contained by the Axes.

Examples using matplotlib.axes.Axes.get_lines

matplotlib.axes.Axes.findobj

Axes.findobj(*self, match=None, include_self=True*)
Find artist objects.

Recursively find all *Artist* instances contained in the artist.

Parameters

match

A filter criterion for the matches. This can be

- *None*: Return all objects contained in artist.
- A function with signature `def match(artist: Artist) -> bool`. The result will only contain artists for which the function returns *True*.
- A class instance: e.g., *Line2D*. The result will only contain artists of this class or its subclasses (`isinstance` check).

include_self

[bool] Include *self* in the list to be checked for a match.

Returns

list of *Artist*

Examples using `matplotlib.axes.Axes.findobj`

18.5.16 Drawing

Axes.draw	Draw the Artist (and its children) using the given renderer.
Axes.draw_artist	Efficiently redraw a single artist.
Axes.redraw_in_frame	Efficiently redraw Axes data, but not axis ticks, labels, etc.
Axes.get_renderer_cache	
Axes.get_rasterization_zorder	Return the zorder value below which artists will be rasterized.
Axes.set_rasterization_zorder	**Parameters**
Axes.get_window_extent	Return the axes bounding box in display space; *args* and *kwargs* are empty.
Axes.get_tightbbox	Return the tight bounding box of the axes, including axis and their decorators (xlabel, title, etc).

matplotlib.axes.Axes.draw

`Axes.draw`(*self*, *renderer=None*, *inframe=<deprecated parameter>*)

Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

Parameters

renderer

[*RendererBase* subclass.]

Notes

This method is overridden in the Artist subclasses.

Examples using `matplotlib.axes.Axes.draw`

matplotlib.axes.Axes.draw_artist

`Axes.draw_artist`(*self*, *a*)

Efficiently redraw a single artist.

This method can only be used after an initial draw which caches the renderer.

Examples using `matplotlib.axes.Axes.draw_artist`

- *Blitting tutorial*

matplotlib.axes.Axes.redraw_in_frame

`Axes.redraw_in_frame`(*self*)

Efficiently redraw Axes data, but not axis ticks, labels, etc.

This method can only be used after an initial draw which caches the renderer.

Examples using `matplotlib.axes.Axes.redraw_in_frame`

matplotlib.axes.Axes.get_renderer_cache

`Axes.get_renderer_cache`(*self*)

Examples using `matplotlib.axes.Axes.get_renderer_cache`

matplotlib.axes.Axes.get_rasterization_zorder

`Axes.get_rasterization_zorder`(*self*)

Return the zorder value below which artists will be rasterized.

Examples using `matplotlib.axes.Axes.get_rasterization_zorder`

matplotlib.axes.Axes.set_rasterization_zorder

`Axes.set_rasterization_zorder`(*self*, *z*)

Parameters

z

[float or None] zorder below which artists are rasterized. `None` means that artists do not get rasterized based on zorder.

Examples using `matplotlib.axes.Axes.set_rasterization_zorder`

- sphx_glr_gallery_misc_rasterization_demo.py

matplotlib.axes.Axes.get_window_extent

`Axes.get_window_extent`(*self*, **args*, ***kwargs*)

Return the axes bounding box in display space; *args* and *kwargs* are empty.

This bounding box does not include the spines, ticks, ticklables, or other labels. For a bounding box including these elements use *get_tightbbox*.

See also:

matplotlib.axes.Axes.get_tightbbox

matplotlib.axis.Axis.get_tightbbox

`matplotlib.spines.get_window_extent`

Examples using `matplotlib.axes.Axes.get_window_extent`

matplotlib.axes.Axes.get_tightbbox

`Axes.get_tightbbox`(*self*, *renderer*, *call_axes_locator=True*, *bbox_extra_artists=None*, ***, *for_layout_only=False*)

Return the tight bounding box of the axes, including axis and their decorators (xlabel, title, etc).

Artists that have `artist.set_in_layout(False)` are not included in the bbox.

Parameters

renderer

[*RendererBase* subclass] renderer that will be used to draw the figures (i.e. `fig.canvas.get_renderer()`)

bbox_extra_artists

[list of *Artist* or `None`] List of artists to include in the tight bounding box. If `None` (default), then all artist children of the axes are included in the tight bounding box.

call_axes_locator

[bool, default: True] If *call_axes_locator* is `False`, it does not call the `_axes_locator` attribute, which is necessary to get the correct bounding box. `call_axes_locator=False` can be used if the caller is only interested in the relative size of the tightbbox compared to the axes bbox.

for_layout_only

[default: False] The bounding box will *not* include the x-extent of the title and the xlabel, or the y-extent of the ylabel.

Returns

BboxBase

Bounding box in figure pixel coordinates.

See also:

matplotlib.axes.Axes.get_window_extent

matplotlib.axis.Axis.get_tightbbox

matplotlib.spines.Spine.get_window_extent

Examples using `matplotlib.axes.Axes.get_tightbbox`

18.5.17 Projection

Methods used by *Axis* that must be overridden for non-rectilinear Axes.

Axes.name	
Axes.get_xaxis_transform	Get the transformation used for drawing x-axis labels, ticks and gridlines.
Axes.get_yaxis_transform	Get the transformation used for drawing y-axis labels, ticks and gridlines.
Axes.get_data_ratio	Return the aspect ratio of the scaled data.
Axes.get_data_ratio_log	[*Deprecated*] Return the aspect ratio of the raw data in log scale.
Axes.get_xaxis_text1_transform	**Returns**
Axes.get_xaxis_text2_transform	**Returns**

continues on next page

Table 113 – continued from previous page

Axes.get_yaxis_text1_transform	**Returns**
Axes.get_yaxis_text2_transform	**Returns**

matplotlib.axes.Axes.name

```
Axes.name = 'rectilinear'
```

matplotlib.axes.Axes.get_xaxis_transform

Axes.**get_xaxis_transform**(*self*, *which='grid'*)

Get the transformation used for drawing x-axis labels, ticks and gridlines. The x-direction is in data coordinates and the y-direction is in axis coordinates.

Note: This transformation is primarily used by the *Axis* class, and is meant to be overridden by new kinds of projections that may need to place axis elements in different locations.

Examples using `matplotlib.axes.Axes.get_xaxis_transform`

- sphx_glr_gallery_lines_bars_and_markers_fill_between_demo.py
- sphx_glr_gallery_lines_bars_and_markers_vline_hline_demo.py
- sphx_glr_gallery_statistics_boxplot_demo.py
- sphx_glr_gallery_recipes_centered_spines_with_arrows.py
- *Transformations Tutorial*

matplotlib.axes.Axes.get_yaxis_transform

Axes.**get_yaxis_transform**(*self*, *which='grid'*)

Get the transformation used for drawing y-axis labels, ticks and gridlines. The x-direction is in axis coordinates and the y-direction is in data coordinates.

Note: This transformation is primarily used by the *Axis* class, and is meant to be overridden by new kinds of projections that may need to place axis elements

in different locations.

Examples using `matplotlib.axes.Axes.get_yaxis_transform`

- sphx_glr_gallery_recipes_centered_spines_with_arrows.py
- sphx_glr_gallery_userdemo_connect_simple01.py
- *Transformations Tutorial*

matplotlib.axes.Axes.get_data_ratio

`Axes.get_data_ratio`(*self*)

Return the aspect ratio of the scaled data.

Notes

This method is intended to be overridden by new projection types.

Examples using `matplotlib.axes.Axes.get_data_ratio`

matplotlib.axes.Axes.get_data_ratio_log

`Axes.get_data_ratio_log`(*self*)

[*Deprecated*] Return the aspect ratio of the raw data in log scale.

Notes

Will be used when both axis are in log scale.

Deprecated since version 3.2.

Examples using `matplotlib.axes.Axes.get_data_ratio_log`

matplotlib.axes.Axes.get_xaxis_text1_transform

`Axes.get_xaxis_text1_transform`(*self*, *pad_points*)

Returns

transform

[Transform] The transform used for drawing x-axis labels, which will add *pad_points* of padding (in points) between the axes and the label. The x-direction is in data coordinates and the y-direction is in axis corrdinates

valign

[{'center', 'top', 'bottom', 'baseline', 'center_baseline'}] The text vertical alignment.

halign

[{'center', 'left', 'right'}] The text horizontal alignment.

Notes

This transformation is primarily used by the *Axis* class, and is meant to be overridden by new kinds of projections that may need to place axis elements in different locations.

Examples using `matplotlib.axes.Axes.get_xaxis_text1_transform`

matplotlib.axes.Axes.get_xaxis_text2_transform

`Axes.get_xaxis_text2_transform`(*self*, *pad_points*)

Returns

transform

[Transform] The transform used for drawing secondary x-axis labels, which will add *pad_points* of padding (in points) between the axes and the label. The x-direction is in data coordinates and the y-direction is in axis corrdinates

valign

[{'center', 'top', 'bottom', 'baseline', 'center_baseline'}] The text vertical alignment.

halign

[{'center', 'left', 'right'}] The text horizontal alignment.

Notes

This transformation is primarily used by the *Axis* class, and is meant to be overridden by new kinds of projections that may need to place axis elements in different locations.

Examples using `matplotlib.axes.Axes.get_xaxis_text2_transform`

matplotlib.axes.Axes.get_yaxis_text1_transform

`Axes.get_yaxis_text1_transform`(*self*, *pad_points*)

Returns

transform

[Transform] The transform used for drawing y-axis labels, which will add *pad_points* of padding (in points) between the axes and the label. The x-direction is in axis coordinates and the y-direction is in data corrdinates

valign

[{'center', 'top', 'bottom', 'baseline', 'center_baseline'}] The text vertical alignment.

halign

[{'center', 'left', 'right'}] The text horizontal alignment.

Notes

This transformation is primarily used by the *Axis* class, and is meant to be overridden by new kinds of projections that may need to place axis elements in different locations.

Examples using `matplotlib.axes.Axes.get_yaxis_text1_transform`

matplotlib.axes.Axes.get_yaxis_text2_transform

`Axes.get_yaxis_text2_transform`(*self*, *pad_points*)

Returns

transform

[Transform] The transform used for drawing secondart y-axis labels, which will add *pad_points* of padding (in points) between

the axes and the label. The x-direction is in axis coordinates and the y-direction is in data corrdinates

valign

[{'center', 'top', 'bottom', 'baseline', 'center_baseline'}] The text vertical alignment.

halign

[{'center', 'left', 'right'}] The text horizontal alignment.

Notes

This transformation is primarily used by the *Axis* class, and is meant to be overridden by new kinds of projections that may need to place axis elements in different locations.

Examples using `matplotlib.axes.Axes.get_yaxis_text2_transform`

18.5.18 Other

Axes.zorder	
Axes.get_default_bbox_extra_artists	Return a default list of artists that are used for the bounding box calculation.
Axes.get_transformed_clip_path_and_affine	Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.
Axes.has_data	Return *True* if any artists have been added to axes.

matplotlib.axes.Axes.zorder

```
Axes.zorder = 0
```

matplotlib.axes.Axes.get_default_bbox_extra_artists

Axes.get_default_bbox_extra_artists(*self*)
Return a default list of artists that are used for the bounding box calculation.

Artists are excluded either by not being visible or `artist.set_in_layout(False)`.

Examples using `matplotlib.axes.Axes.get_default_bbox_extra_artists`

matplotlib.axes.Axes.get_transformed_clip_path_and_affine

Axes.get_transformed_clip_path_and_affine(*self*)
Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

Examples using `matplotlib.axes.Axes.get_transformed_clip_path_and_affine`

matplotlib.axes.Axes.has_data

Axes.has_data(*self*)
Return *True* if any artists have been added to axes.

This should not be used to determine whether the *dataLim* need to be updated, and may not actually be useful for anything.

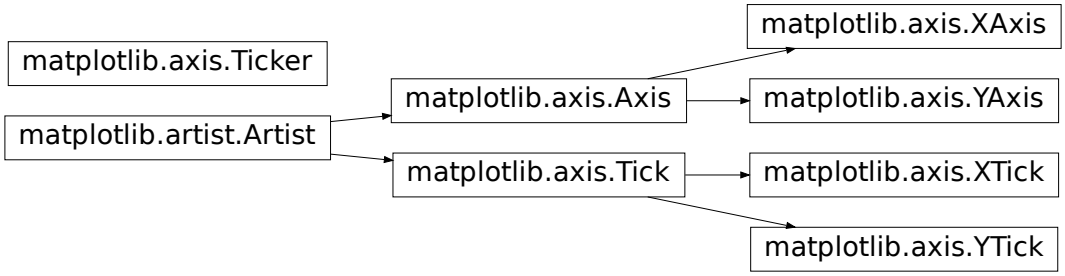
Examples using `matplotlib.axes.Axes.has_data`

18.6 matplotlib.axis

Table of Contents

Classes for the ticks and x and y axis.

18.6.1 Inheritance



18.6.2 `Axis` objects

`class matplotlib.axis.Axis(`*axes, pickradius=15*`)`

Base class for *XAxis* and *YAxis*.

Attributes

isDefault_label

[bool]

axes

[*matplotlib.axes.Axes*] The *Axes* instance the artist resides in, or *None*.

major

[*matplotlib.axis.Ticker*] Determines the major tick positions and their label format.

minor

[*matplotlib.axis.Ticker*] Determines the minor tick positions and their label format.

callbacks

[*matplotlib.cbook.CallbackRegistry*]

label

[*Text*] The axis label.

labelpad

[float] The distance between the axis label and the tick labels. Defaults to `rcParams["axes.labelpad"]` (default: `4.0`) = 4.

offsetText

[*Text*] A *Text* object containing the data offset of the ticks (if any).

pickradius

[float] The acceptance radius for containment tests. See also *Axis.contains*.

majorTicks

[list of *Tick*] The major ticks.

minorTicks

[list of *Tick*] The minor ticks.

Parameters

axes

[*matplotlib.axes.Axes*] The *Axes* to which the created Axis belongs.

pickradius

[float] The acceptance radius for containment tests. See also *Axis.contains*.

`class matplotlib.axis.XAxis(*args, **kwargs)`

Parameters

axes

[*matplotlib.axes.Axes*] The *Axes* to which the created Axis belongs.

pickradius

[float] The acceptance radius for containment tests. See also *Axis.contains*.

class matplotlib.axis.YAxis(*args, **kwargs)

Parameters

axes

[*matplotlib.axes.Axes*] The *Axes* to which the created Axis belongs.

pickradius

[float] The acceptance radius for containment tests. See also *Axis.contains*.

class matplotlib.axis.Ticker

A container for the objects defining tick position and format.

Attributes

locator

[*matplotlib.ticker.Locator* subclass] Determines the positions of the ticks.

formatter

[*matplotlib.ticker.Formatter* subclass] Determines the format of the tick labels.

Axis.cla	Clear this axis.
Axis.get_scale	Return this Axis' scale (as a str).

matplotlib.axis.Axis.cla

Axis.cla(*self*)

Clear this axis.

Examples using `matplotlib.axis.Axis.cla`

matplotlib.axis.Axis.get_scale

Axis.get_scale(*self*)
Return this Axis' scale (as a str).

Examples using `matplotlib.axis.Axis.get_scale`

Formatters and Locators

Axis.get_major_formatter	Get the formatter of the major ticker.
Axis.get_major_locator	Get the locator of the major ticker.
Axis.get_minor_formatter	Get the formatter of the minor ticker.
Axis.get_minor_locator	Get the locator of the minor ticker.
Axis.set_major_formatter	Set the formatter of the major ticker.
Axis.set_major_locator	Set the locator of the major ticker.
Axis.set_minor_formatter	Set the formatter of the minor ticker.
Axis.set_minor_locator	Set the locator of the minor ticker.
Axis.remove_overlapping_locs	If minor ticker locations that overlap with major ticker locations should be trimmed.
Axis.get_remove_overlapping_locs	
Axis.set_remove_overlapping_locs	

matplotlib.axis.Axis.get_major_formatter

Axis.get_major_formatter(*self*)
Get the formatter of the major ticker.

Examples using `matplotlib.axis.Axis.get_major_formatter`

matplotlib.axis.Axis.get_major_locator

Axis.get_major_locator(*self*)
Get the locator of the major ticker.

Examples using `matplotlib.axis.Axis.get_major_locator`

- sphx_glr_gallery_axes_grid1_inset_locator_demo2.py

matplotlib.axis.Axis.get_minor_formatter

`Axis.get_minor_formatter`(*self*)
Get the formatter of the minor ticker.

Examples using `matplotlib.axis.Axis.get_minor_formatter`

matplotlib.axis.Axis.get_minor_locator

`Axis.get_minor_locator`(*self*)
Get the locator of the minor ticker.

Examples using `matplotlib.axis.Axis.get_minor_locator`

matplotlib.axis.Axis.set_major_formatter

`Axis.set_major_formatter`(*self*, *formatter*)
Set the formatter of the major ticker.

In addition to a *Formatter* instance, this also accepts a `str` or function.

For a `str` a *StrMethodFormatter* is used. The field used for the value must be labeled `'x'` and the field used for the position must be labeled `'pos'`. See the *StrMethodFormatter* documentation for more information.

For a function, a *FuncFormatter* is used. The function must take two inputs (a tick value `x` and a position `pos`), and return a string containing the corresponding tick label. See the *FuncFormatter* documentation for more information.

Parameters

formatter

[*Formatter*, `str`, or function]

Examples using `matplotlib.axis.Axis.set_major_formatter`

- sphx_glr_gallery_lines_bars_and_markers_timeline.py
- sphx_glr_gallery_text_labels_and_annotations_date.py
- sphx_glr_gallery_text_labels_and_annotations_date_index_formatter.py
- sphx_glr_gallery_text_labels_and_annotations_engineering_formatter.py
- sphx_glr_gallery_pyplots_dollar_ticks.py
- sphx_glr_gallery_showcase_bachelors_degrees_by_gender.py
- sphx_glr_gallery_mplot3d_surface3d.py
- sphx_glr_gallery_specialty_plots_skewt.py
- sphx_glr_gallery_ticks_and_spines_centered_ticklabels.py
- sphx_glr_gallery_ticks_and_spines_custom_ticker1.py
- sphx_glr_gallery_ticks_and_spines_date_concise_formatter.py
- sphx_glr_gallery_ticks_and_spines_date_demo_convert.py
- sphx_glr_gallery_ticks_and_spines_date_demo_rrule.py
- sphx_glr_gallery_ticks_and_spines_date_index_formatter2.py
- sphx_glr_gallery_ticks_and_spines_major_minor_demo.py
- sphx_glr_gallery_ticks_and_spines_tick-formatters.py
- sphx_glr_gallery_ticks_and_spines_tick_labels_from_values.py
- *The Lifecycle of a Plot*
- *Artist tutorial*
- *Choosing Colormaps in Matplotlib*
- *Text in Matplotlib Plots*

matplotlib.axis.Axis.set_major_locator

`Axis.set_major_locator`(*self*, *locator*)

Set the locator of the major ticker.

Parameters

locator

[*Locator*]

Examples using `matplotlib.axis.Axis.set_major_locator`

- sphx_glr_gallery_lines_bars_and_markers_filled_step.py
- sphx_glr_gallery_lines_bars_and_markers_timeline.py
- sphx_glr_gallery_text_labels_and_annotations_date.py
- sphx_glr_gallery_showcase_anatomy.py
- sphx_glr_gallery_mplot3d_surface3d.py
- sphx_glr_gallery_mplot3d_surface3d_3.py
- sphx_glr_gallery_scales_scales.py
- sphx_glr_gallery_specialty_plots_mri_with_eeg.py
- sphx_glr_gallery_specialty_plots_skewt.py
- sphx_glr_gallery_ticks_and_spines_centered_ticklabels.py
- sphx_glr_gallery_ticks_and_spines_date_concise_formatter.py
- sphx_glr_gallery_ticks_and_spines_date_demo_convert.py
- sphx_glr_gallery_ticks_and_spines_date_demo_rrule.py
- sphx_glr_gallery_ticks_and_spines_major_minor_demo.py
- sphx_glr_gallery_ticks_and_spines_tick-formatters.py
- sphx_glr_gallery_ticks_and_spines_tick_labels_from_values.py
- *Choosing Colormaps in Matplotlib*
- *Text in Matplotlib Plots*

matplotlib.axis.Axis.set_minor_formatter

`Axis.set_minor_formatter`(*self*, *formatter*)

Set the formatter of the minor ticker.

In addition to a *Formatter* instance, this also accepts a `str` or function. See *Axis.set_major_formatter* for more information.

Parameters

formatter

[*Formatter*, `str`, or function]

Examples using `matplotlib.axis.Axis.set_minor_formatter`

- sphx_glr_gallery_showcase_anatomy.py
- sphx_glr_gallery_scales_scales.py
- sphx_glr_gallery_specialty_plots_skewt.py
- sphx_glr_gallery_ticks_and_spines_centered_ticklabels.py

matplotlib.axis.Axis.set_minor_locator

`Axis.set_minor_locator`(*self*, *locator*)

Set the locator of the minor ticker.

Parameters

locator

[*Locator*]

Examples using `matplotlib.axis.Axis.set_minor_locator`

- sphx_glr_gallery_subplots_axes_and_figures_secondary_axis.py
- sphx_glr_gallery_text_labels_and_annotations_date.py
- sphx_glr_gallery_showcase_anatomy.py
- sphx_glr_gallery_ticks_and_spines_centered_ticklabels.py
- sphx_glr_gallery_ticks_and_spines_date_demo_convert.py
- sphx_glr_gallery_ticks_and_spines_major_minor_demo.py
- sphx_glr_gallery_ticks_and_spines_tick-formatters.py

matplotlib.axis.Axis.remove_overlapping_locs

`property Axis.remove_overlapping_locs`

If minor ticker locations that overlap with major ticker locations should be trimmed.

matplotlib.axis.Axis.get_remove_overlapping_locs

Axis.get_remove_overlapping_locs(*self*)

Examples using matplotlib.axis.Axis.get_remove_overlapping_locs

matplotlib.axis.Axis.set_remove_overlapping_locs

Axis.set_remove_overlapping_locs(*self*, *val*)

Examples using matplotlib.axis.Axis.set_remove_overlapping_locs

Axis Label

Axis.set_label_coords	Set the coordinates of the label.
Axis.set_label_position	Set the label position (top or bottom)
Axis.set_label_text	Set the text value of the axis label.
Axis.get_label_position	Return the label position (top or bottom)
Axis.get_label_text	Get the text of the label.

matplotlib.axis.Axis.set_label_coords

Axis.set_label_coords(*self*, *x*, *y*, *transform=None*)

Set the coordinates of the label.

By default, the x coordinate of the y label and the y coordinate of the x label are determined by the tick label bounding boxes, but this can lead to poor alignment of multiple labels if there are multiple axes.

You can also specify the coordinate system of the label with the transform. If None, the default coordinate system will be the axes coordinate system: (0, 0) is bottom left, (0.5, 0.5) is center, etc.

Examples using matplotlib.axis.Axis.set_label_coords

- sphx_glr_gallery_pyplots_align_ylabels.py

matplotlib.axis.Axis.set_label_position

Axis.set_label_position(*self*, *position*)
Set the label position (top or bottom)

Parameters

position

[{'top', 'bottom'}]

Examples using matplotlib.axis.Axis.set_label_position

- sphx_glr_gallery_text_labels_and_annotations_titles_demo.py

matplotlib.axis.Axis.set_label_text

Axis.set_label_text(*self*, *label*, *fontdict=None*, ***kwargs*)
Set the text value of the axis label.

Parameters

label

[str] Text string.

fontdict

[dict] Text properties.

****kwargs**

Merged into fontdict.

Examples using matplotlib.axis.Axis.set_label_text

matplotlib.axis.Axis.get_label_position

Axis.get_label_position(*self*)
Return the label position (top or bottom)

Examples using `matplotlib.axis.Axis.get_label_position`

matplotlib.axis.Axis.get_label_text

Axis.**get_label_text**(*self*)
Get the text of the label.

Examples using `matplotlib.axis.Axis.get_label_text`

Ticks, tick labels and Offset text

Axis.get_major_ticks	Return the list of major *Tick*s.
Axis.get_majorticklabels	Return this Axis' major tick labels, as a list of *Text*.
Axis.get_majorticklines	Return this Axis' major tick lines as a list of *Line2D*s.
Axis.get_majorticklocs	Return this Axis' major tick locations in data coordinates.
Axis.get_minor_ticks	Return the list of minor *Tick*s.
Axis.get_minorticklabels	Return this Axis' minor tick labels, as a list of *Text*.
Axis.get_minorticklines	Return this Axis' minor tick lines as a list of *Line2D*s.
Axis.get_minorticklocs	Return this Axis' minor tick locations in data coordinates.
Axis.get_offset_text	Return the axis offsetText as a Text instance.
Axis.get_tick_padding	
Axis.get_ticklabels	Get this Axis' tick labels.
Axis.get_ticklines	Return this Axis' tick lines as a list of *Line2D*s.
Axis.get_ticklocs	Return this Axis' tick locations in data coordinates.
Axis.get_gridlines	Return this Axis' grid lines as a list of *Line2D*s.
Axis.grid	Configure the grid lines.
Axis.set_tick_params	Set appearance parameters for ticks, ticklabels, and gridlines.
Axis.axis_date	Sets up axis ticks and labels to treat data along this Axis as dates.

matplotlib.axis.Axis.get_major_ticks

Axis.get_major_ticks(*self*, *numticks=None*)
Return the list of major *Tick*s.

Examples using matplotlib.axis.Axis.get_major_ticks

- sphx_glr_gallery_pyplots_dollar_ticks.py
- *Artist tutorial*

matplotlib.axis.Axis.get_majorticklabels

Axis.get_majorticklabels(*self*)
Return this Axis' major tick labels, as a list of *Text*.

Examples using matplotlib.axis.Axis.get_majorticklabels

- sphx_glr_gallery_ticks_and_spines_date_precision_and_epochs.py

matplotlib.axis.Axis.get_majorticklines

Axis.get_majorticklines(*self*)
Return this Axis' major tick lines as a list of *Line2D*s.

Examples using matplotlib.axis.Axis.get_majorticklines

matplotlib.axis.Axis.get_majorticklocs

Axis.get_majorticklocs(*self*)
Return this Axis' major tick locations in data coordinates.

Examples using matplotlib.axis.Axis.get_majorticklocs

matplotlib.axis.Axis.get_minor_ticks

Axis.get_minor_ticks(*self*, *numticks=None*)
Return the list of minor *Tick*s.

Examples using `matplotlib.axis.Axis.get_minor_ticks`

- sphx_glr_gallery_ticks_and_spines_centered_ticklabels.py

matplotlib.axis.Axis.get_minorticklabels

`Axis.get_minorticklabels(`*self*`)`
Return this Axis' minor tick labels, as a list of *Text*.

Examples using `matplotlib.axis.Axis.get_minorticklabels`

matplotlib.axis.Axis.get_minorticklines

`Axis.get_minorticklines(`*self*`)`
Return this Axis' minor tick lines as a list of *Line2D*s.

Examples using `matplotlib.axis.Axis.get_minorticklines`

matplotlib.axis.Axis.get_minorticklocs

`Axis.get_minorticklocs(`*self*`)`
Return this Axis' minor tick locations in data coordinates.

Examples using `matplotlib.axis.Axis.get_minorticklocs`

matplotlib.axis.Axis.get_offset_text

`Axis.get_offset_text(`*self*`)`
Return the axis offsetText as a Text instance.

Examples using `matplotlib.axis.Axis.get_offset_text`

matplotlib.axis.Axis.get_tick_padding

`Axis.get_tick_padding(`*self*`)`

Examples using `matplotlib.axis.Axis.get_tick_padding`

matplotlib.axis.Axis.get_ticklabels

`Axis.get_ticklabels`(*self*, *minor=False*, *which=None*)
Get this Axis' tick labels.

Parameters

minor

[bool] Whether to return the minor or the major ticklabels.

which

[None, ('minor', 'major', 'both')] Overrides *minor*.

Selects which ticklabels to return

Returns

list of *Text*

Notes

The tick label strings are not populated until a `draw` method has been called.

See also: *draw* and *draw*.

Examples using `matplotlib.axis.Axis.get_ticklabels`

- sphx_glr_gallery_images_contours_and_fields_image_masked.py
- sphx_glr_gallery_pyplots_fig_axes_customize_simple.py
- *Artist tutorial*

matplotlib.axis.Axis.get_ticklines

`Axis.get_ticklines`(*self*, *minor=False*)
Return this Axis' tick lines as a list of *Line2D*s.

Examples using `matplotlib.axis.Axis.get_ticklines`

- sphx_glr_gallery_pyplots_fig_axes_customize_simple.py
- *Artist tutorial*

matplotlib.axis.Axis.get_ticklocs

`Axis.get_ticklocs`(*self*, *, *minor=False*)
Return this Axis' tick locations in data coordinates.

Examples using `matplotlib.axis.Axis.get_ticklocs`

- *Artist tutorial*

matplotlib.axis.Axis.get_gridlines

`Axis.get_gridlines`(*self*)
Return this Axis' grid lines as a list of *Line2D*s.

Examples using `matplotlib.axis.Axis.get_gridlines`

matplotlib.axis.Axis.grid

`Axis.grid`(*self*, *b=None*, *which='major'*, ***kwargs*)
Configure the grid lines.

Parameters

b

[bool or None] Whether to show the grid lines. If any *kwargs* are supplied, it is assumed you want the grid on and *b* will be set to True.

If *b* is *None* and there are no *kwargs*, this toggles the visibility of the lines.

which

[{'major', 'minor', 'both'}] The grid lines to apply the changes on.

****kwargs**

[*Line2D* properties] Define the line properties of the grid, e.g.:

```
grid(color='r', linestyle='-', linewidth=2)
```

Examples using `matplotlib.axis.Axis.grid`

- sphx_glr_gallery_statistics_boxplot_color.py
- sphx_glr_gallery_statistics_boxplot_demo.py
- sphx_glr_gallery_statistics_boxplot_vs_violin.py

matplotlib.axis.Axis.set_tick_params

Axis.**set_tick_params**(*self*, *which='major'*, *reset=False*, ***kw*)
Set appearance parameters for ticks, ticklabels, and gridlines.

For documentation of keyword arguments, see *matplotlib.axes.Axes.tick_params()*.

Examples using `matplotlib.axis.Axis.set_tick_params`

- sphx_glr_gallery_pyplots_dollar_ticks.py
- sphx_glr_gallery_axes_grid1_scatter_hist_locatable_axes.py
- sphx_glr_gallery_misc_pythonic_matplotlib.py
- sphx_glr_gallery_ticks_and_spines_date_demo_rrule.py
- *Choosing Colormaps in Matplotlib*

matplotlib.axis.Axis.axis_date

Axis.**axis_date**(*self*, *tz=None*)
Sets up axis ticks and labels to treat data along this Axis as dates.

Parameters

tz

[str or `datetime.tzinfo`, default: `rcParams["timezone"]` (default: `'UTC'`)] The timezone used to create date labels.

Examples using `matplotlib.axis.Axis.axis_date`

Data and view intervals

Axis.get_data_interval	Return the Interval instance for this axis data limits.
Axis.get_view_interval	Return the view limits `(min, max)` of this axis.
Axis.set_data_interval	Set the axis data limits.
Axis.set_view_interval	Set the axis view limits.

matplotlib.axis.Axis.get_data_interval

`Axis.get_data_interval`(*self*)
Return the Interval instance for this axis data limits.

Examples using `matplotlib.axis.Axis.get_data_interval`

matplotlib.axis.Axis.get_view_interval

`Axis.get_view_interval`(*self*)
Return the view limits `(min, max)` of this axis.

Examples using `matplotlib.axis.Axis.get_view_interval`

matplotlib.axis.Axis.set_data_interval

`Axis.set_data_interval`(*self*, *vmin*, *vmax*, *ignore=False*)
Set the axis data limits. This method is for internal use.

If *ignore* is False (the default), this method will never reduce the preexisting data limits, only expand them if *vmin* or *vmax* are not within them. Moreover, the order of *vmin* and *vmax* does not matter; the orientation of the axis will not change.

If *ignore* is True, the data limits will be set exactly to `(vmin, vmax)` in that order.

Examples using `matplotlib.axis.Axis.set_data_interval`

matplotlib.axis.Axis.set_view_interval

Axis.**set_view_interval**(*self*, *vmin*, *vmax*, *ignore=False*)

Set the axis view limits. This method is for internal use; Matplotlib users should typically use e.g. *set_xlim* or *set_ylim*.

If *ignore* is False (the default), this method will never reduce the preexisting view limits, only expand them if *vmin* or *vmax* are not within them. Moreover, the order of *vmin* and *vmax* does not matter; the orientation of the axis will not change.

If *ignore* is True, the view limits will be set exactly to `(vmin, vmax)` in that order.

Examples using `matplotlib.axis.Axis.set_view_interval`

Rendering helpers

Axis.get_minpos	
Axis.get_tick_space	Return the estimated number of ticks that can fit on the axis.
Axis.get_ticklabel_extents	Get the extents of the tick labels on either side of the axes.
Axis.get_tightbbox	Return a bounding box that encloses the axis.

matplotlib.axis.Axis.get_minpos

Axis.**get_minpos**(*self*)

Examples using `matplotlib.axis.Axis.get_minpos`

matplotlib.axis.Axis.get_tick_space

Axis.**get_tick_space**(*self*)

Return the estimated number of ticks that can fit on the axis.

Examples using `matplotlib.axis.Axis.get_tick_space`

matplotlib.axis.Axis.get_ticklabel_extents

`Axis.get_ticklabel_extents`(*self*, *renderer*)
Get the extents of the tick labels on either side of the axes.

Examples using `matplotlib.axis.Axis.get_ticklabel_extents`

matplotlib.axis.Axis.get_tightbbox

`Axis.get_tightbbox`(*self*, *renderer*, *, *for_layout_only=False*)
Return a bounding box that encloses the axis. It only accounts tick labels, axis label, and offsetText.

If *for_layout_only* is True, then the width of the label (if this is an x-axis) or the height of the label (if this is a y-axis) is collapsed to near zero. This allows tight/constrained_layout to ignore too-long labels when doing their layout.

Examples using `matplotlib.axis.Axis.get_tightbbox`

Interactive

Axis.contains	Test whether the artist contains the mouse event.
Axis.get_pickradius	Return the depth of the axis used by the picker.
Axis.set_pickradius	Set the depth of the axis used by the picker.

matplotlib.axis.Axis.contains

`Axis.contains`(*self*, *mouseevent*)
Test whether the artist contains the mouse event.

Parameters

mouseevent

[*matplotlib.backend_bases.MouseEvent*]

Returns

contains

[bool] Whether any values are within the radius.

details

[dict] An artist-specific dictionary of details of the event context, such as which points are contained in the pick radius. See the individual Artist subclasses for details.

Examples using `matplotlib.axis.Axis.contains`

matplotlib.axis.Axis.get_pickradius

`Axis.get_pickradius`(*self*)
Return the depth of the axis used by the picker.

Examples using `matplotlib.axis.Axis.get_pickradius`

matplotlib.axis.Axis.set_pickradius

`Axis.set_pickradius`(*self*, *pickradius*)
Set the depth of the axis used by the picker.

Parameters

pickradius

[float]

Examples using `matplotlib.axis.Axis.set_pickradius`

Units

Axis.convert_units	
Axis.set_units	Set the units for axis.
Axis.get_units	Return the units for axis.
Axis.update_units	Introspect *data* for units converter and update the axis.converter instance if necessary.

matplotlib.axis.Axis.convert_units

Axis.convert_units(*self*, *x*)

Examples using matplotlib.axis.Axis.convert_units

matplotlib.axis.Axis.set_units

Axis.set_units(*self*, *u*)

Set the units for axis.

Parameters

u

[units tag]

Examples using matplotlib.axis.Axis.set_units

- sphx_glr_gallery_units_artist_tests.py
- sphx_glr_gallery_units_bar_unit_demo.py
- sphx_glr_gallery_units_units_scatter.py

matplotlib.axis.Axis.get_units

Axis.get_units(*self*)

Return the units for axis.

Examples using matplotlib.axis.Axis.get_units

matplotlib.axis.Axis.update_units

Axis.update_units(*self*, *data*)

Introspect *data* for units converter and update the axis.converter instance if necessary. Return *True* if *data* is registered for unit conversion.

Examples using `matplotlib.axis.Axis.update_units`

Incremental navigation

Axis.pan	[*Deprecated*] Pan by *numsteps* (can be positive or negative).
Axis.zoom	[*Deprecated*] Zoom in/out on axis; if *direction* is >0 zoom in, else zoom out.

matplotlib.axis.Axis.pan

Axis.**pan**(*self*, *numsteps*)

[*Deprecated*] Pan by *numsteps* (can be positive or negative).

Notes

Deprecated since version 3.3.

Examples using `matplotlib.axis.Axis.pan`

matplotlib.axis.Axis.zoom

Axis.**zoom**(*self*, *direction*)

[*Deprecated*] Zoom in/out on axis; if *direction* is >0 zoom in, else zoom out.

Notes

Deprecated since version 3.3.

Examples using `matplotlib.axis.Axis.zoom`

XAxis Specific

XAxis.axis_name	Read-only name identifying the axis.
XAxis.get_text_heights	Return how much space should be reserved for text above and below the axes, as a pair of floats.

continues on next page

Table 124 – continued from previous page

XAxis.get_ticks_position	Return the ticks position ("top", "bottom", "default", or "unknown").
XAxis.set_ticks_position	Set the ticks position.
XAxis.tick_bottom	Move ticks and ticklabels (if present) to the bottom of the axes.
XAxis.tick_top	Move ticks and ticklabels (if present) to the top of the axes.

matplotlib.axis.XAxis.axis_name

XAxis.**axis_name** = 'x'
Read-only name identifying the axis.

matplotlib.axis.XAxis.get_text_heights

XAxis.**get_text_heights**(*self*, *renderer*)
Return how much space should be reserved for text above and below the axes, as a pair of floats.

Examples using matplotlib.axis.XAxis.get_text_heights

matplotlib.axis.XAxis.get_ticks_position

XAxis.**get_ticks_position**(*self*)
Return the ticks position ("top", "bottom", "default", or "unknown").

Examples using matplotlib.axis.XAxis.get_ticks_position

matplotlib.axis.XAxis.set_ticks_position

XAxis.**set_ticks_position**(*self*, *position*)
Set the ticks position.

Parameters

position

[{'top', 'bottom', 'both', 'default', 'none'}] 'both' sets the ticks to appear on both positions, but does not change the tick labels. 'default' resets the tick positions to the default: ticks on both positions, labels at bottom. 'none' can be used if you don't want any ticks. 'none' and 'both' affect only the ticks, not the labels.

Examples using `matplotlib.axis.XAxis.set_ticks_position`

- sphx_glr_gallery_statistics_customized_violin.py
- sphx_glr_gallery_pyplots_whats_new_99_spines.py
- sphx_glr_gallery_axes_grid1_demo_colorbar_with_axes_divider.py
- sphx_glr_gallery_axes_grid1_demo_colorbar_with_inset_locator.py
- sphx_glr_gallery_showcase_integral.py
- sphx_glr_gallery_showcase_xkcd.py
- sphx_glr_gallery_ticks_and_spines_spine_placement_demo.py
- sphx_glr_gallery_ticks_and_spines_spines.py
- sphx_glr_gallery_ticks_and_spines_spines_bounds.py
- sphx_glr_gallery_ticks_and_spines_spines_dropped.py
- sphx_glr_gallery_ticks_and_spines_tick-formatters.py
- *Choosing Colormaps in Matplotlib*

matplotlib.axis.XAxis.tick_bottom

`XAxis.tick_bottom(`*self*`)`
Move ticks and ticklabels (if present) to the bottom of the axes.

Examples using `matplotlib.axis.XAxis.tick_bottom`

- sphx_glr_gallery_subplots_axes_and_figures_broken_axis.py
- sphx_glr_gallery_showcase_bachelors_degrees_by_gender.py

matplotlib.axis.XAxis.tick_top

`XAxis.tick_top(`*self*`)`
Move ticks and ticklabels (if present) to the top of the axes.

Examples using `matplotlib.axis.XAxis.tick_top`

- sphx_glr_gallery_subplots_axes_and_figures_broken_axis.py
- sphx_glr_gallery_text_labels_and_annotations_titles_demo.py

YAxis Specific

YAxis.axis_name	Read-only name identifying the axis.
YAxis.get_text_widths	
YAxis.get_ticks_position	Return the ticks position ("left", "right", "default", or "unknown").
YAxis.set_offset_position	**Parameters**
YAxis.set_ticks_position	Set the ticks position.
YAxis.tick_left	Move ticks and ticklabels (if present) to the left of the axes.
YAxis.tick_right	Move ticks and ticklabels (if present) to the right of the axes.

matplotlib.axis.YAxis.axis_name

YAxis.**axis_name** = 'y'
Read-only name identifying the axis.

matplotlib.axis.YAxis.get_text_widths

YAxis.**get_text_widths**(*self*, *renderer*)

Examples using `matplotlib.axis.YAxis.get_text_widths`

matplotlib.axis.YAxis.get_ticks_position

YAxis.**get_ticks_position**(*self*)
Return the ticks position ("left", "right", "default", or "unknown").

Examples using `matplotlib.axis.YAxis.get_ticks_position`

matplotlib.axis.YAxis.set_offset_position

`YAxis.set_offset_position(`*self*, *position*`)`

Parameters

position

[{'left', 'right'}]

Examples using `matplotlib.axis.YAxis.set_offset_position`

matplotlib.axis.YAxis.set_ticks_position

`YAxis.set_ticks_position(`*self*, *position*`)`

Set the ticks position.

Parameters

position

[{'left', 'right', 'both', 'default', 'none'}] 'both' sets the ticks to appear on both positions, but does not change the tick labels. 'default' resets the tick positions to the default: ticks on both positions, labels at left. 'none' can be used if you don't want any ticks. 'none' and 'both' affect only the ticks, not the labels.

Examples using `matplotlib.axis.YAxis.set_ticks_position`

- sphx_glr_gallery_pyplots_whats_new_99_spines.py
- sphx_glr_gallery_ticks_and_spines_spine_placement_demo.py
- sphx_glr_gallery_ticks_and_spines_spines.py
- sphx_glr_gallery_ticks_and_spines_spines_bounds.py
- sphx_glr_gallery_ticks_and_spines_spines_dropped.py
- sphx_glr_gallery_ticks_and_spines_tick-formatters.py

matplotlib.axis.YAxis.tick_left

YAxis.**tick_left**(*self*)
Move ticks and ticklabels (if present) to the left of the axes.

Examples using matplotlib.axis.YAxis.tick_left

- sphx_glr_gallery_showcase_bachelors_degrees_by_gender.py
- sphx_glr_gallery_ticks_and_spines_tick_label_right.py

matplotlib.axis.YAxis.tick_right

YAxis.**tick_right**(*self*)
Move ticks and ticklabels (if present) to the right of the axes.

Examples using matplotlib.axis.YAxis.tick_right

Other

Axis.OFFSETTEXTPAD	
Axis.axes	The *Axes* instance the artist resides in, or *None*.
Axis.limit_range_for_scale	
Axis.reset_ticks	Re-initialize the major and minor Tick lists.
Axis.set_default_intervals	Set the default limits for the axis data and view interval if they have not been not mutated yet.
Axis.get_smart_bounds	[*Deprecated*] Return whether the axis has smart bounds.
Axis.set_smart_bounds	[*Deprecated*] Set the axis to have smart bounds.

matplotlib.axis.Axis.OFFSETTEXTPAD

```
Axis.OFFSETTEXTPAD = 3
```

matplotlib.axis.Axis.axes

```
property Axis.axes
```
The *Axes* instance the artist resides in, or *None*.

matplotlib.axis.Axis.limit_range_for_scale

```
Axis.limit_range_for_scale(self, vmin, vmax)
```

Examples using matplotlib.axis.Axis.limit_range_for_scale

matplotlib.axis.Axis.reset_ticks

```
Axis.reset_ticks(self)
```
Re-initialize the major and minor Tick lists.

Each list starts with a single fresh Tick.

Examples using matplotlib.axis.Axis.reset_ticks

matplotlib.axis.Axis.set_default_intervals

```
Axis.set_default_intervals(self)
```
Set the default limits for the axis data and view interval if they have not been not mutated yet.

Examples using matplotlib.axis.Axis.set_default_intervals

matplotlib.axis.Axis.get_smart_bounds

```
Axis.get_smart_bounds(self)
```
[*Deprecated*] Return whether the axis has smart bounds.

Notes

Deprecated since version 3.2.

Examples using `matplotlib.axis.Axis.get_smart_bounds`

matplotlib.axis.Axis.set_smart_bounds

`Axis.set_smart_bounds`(*self*, *value*)
[*Deprecated*] Set the axis to have smart bounds.

Notes

Deprecated since version 3.2.

Examples using `matplotlib.axis.Axis.set_smart_bounds`

Discouraged

These methods should be used together with care, calling `set_ticks` to specify the desired tick locations **before** calling `set_ticklabels` to specify a matching series of labels. Calling `set_ticks` makes a *FixedLocator*; it's list of locations is then used by `set_ticklabels` to make an appropriate *FuncFormatter*.

Axis.set_ticks	Set this Axis' tick locations.
Axis.set_ticklabels	Set the text values of the tick labels.

matplotlib.axis.Axis.set_ticks

`Axis.set_ticks`(*self*, *ticks*, *, *minor=False*)
Set this Axis' tick locations.

Parameters

ticks

[list of floats] List of tick locations.

minor

[bool, default: False] If `False`, set the major ticks; if `True`, the minor ticks.

Examples using `matplotlib.axis.Axis.set_ticks`

- sphx_glr_gallery_pyplots_whats_new_99_spines.py
- sphx_glr_gallery_ticks_and_spines_spine_placement_demo.py

matplotlib.axis.Axis.set_ticklabels

`Axis.set_ticklabels`(*self*, *ticklabels*, *, *minor=False*, ***kwargs*)

Set the text values of the tick labels.

> **Warning:** This method should only be used after fixing the tick positions using *Axis.set_ticks*. Otherwise, the labels may end up in unexpected positions.

Parameters

ticklabels

[sequence of str or of *Text*s] Texts for labeling each tick location in the sequence set by *Axis.set_ticks*; the number of labels must match the number of locations.

minor

[bool] If True, set minor ticks instead of major ticks.

****kwargs**

Text properties.

Returns

list of *Text*s

For each tick, includes `tick.label1` if it is visible, then `tick.label2` if it is visible, in that order.

Examples using `matplotlib.axis.Axis.set_ticklabels`

18.6.3 `Tick` objects

`class matplotlib.axis.Tick(`*axes*, *loc*, *label=<deprecated parameter>*, *size=None*, *width=None*, *color=None*, *tickdir=None*, *pad=None*, *labelsize=None*, *labelcolor=None*, *zorder=None*, *gridOn=None*, *tick1On=True*, *tick2On=True*, *label1On=True*, *label2On=False*, *major=True*, *labelrotation=0*, *grid_color=None*, *grid_linestyle=None*, *grid_linewidth=None*, *grid_alpha=None*, ***kw*`)`

Abstract base class for the axis ticks, grid lines and labels.

Ticks mark a position on an Axis. They contain two lines as markers and two labels; one each for the bottom and top positions (in case of an *XAxis*) or for the left and right positions (in case of a *YAxis*).

Attributes

tick1line

[*Line2D*] The left/bottom tick marker.

tick2line

[*Line2D*] The right/top tick marker.

gridline

[*Line2D*] The grid line associated with the label position.

label1

[*Text*] The left/bottom tick label.

label2

[*Text*] The right/top tick label.

bbox is the Bound2D bounding box in display coords of the Axes loc is the tick location in data coords size is the tick size in points

`class matplotlib.axis.XTick(`**args*, ***kwargs*`)`

Contains all the Artists needed to make an x tick - the tick line, the label text and the grid line

bbox is the Bound2D bounding box in display coords of the Axes loc is the tick location in data coords size is the tick size in points

`class matplotlib.axis.YTick(`**args*, ***kwargs*`)`

Contains all the Artists needed to make a Y tick - the tick line, the label text and the grid line

bbox is the Bound2D bounding box in display coords of the Axes loc is the tick location in data coords size is the tick size in points

Tick.apply_tickdir	Calculate `self._pad` and `self._tickmarkers`.
Tick.get_loc	Return the tick location (data coords) as a scalar.
Tick.get_pad	Get the value of the tick label pad in points.
Tick.get_pad_pixels	
Tick.get_tick_padding	Get the length of the tick outside of the axes.
Tick.get_tickdir	
Tick.get_view_interval	Return the view limits `(min, max)` of the axis the tick belongs to.
Tick.set_label1	Set the label1 text.
Tick.set_label2	Set the label2 text.
Tick.set_pad	Set the tick label pad in points
Tick.update_position	Set the location of tick in data coords with scalar *loc*.

matplotlib.axis.Tick.apply_tickdir

Tick.**apply_tickdir**(*self*, *tickdir*)

Calculate `self._pad` and `self._tickmarkers`.

Examples using `matplotlib.axis.Tick.apply_tickdir`

matplotlib.axis.Tick.get_loc

Tick.**get_loc**(*self*)

Return the tick location (data coords) as a scalar.

Examples using `matplotlib.axis.Tick.get_loc`

matplotlib.axis.Tick.get_pad

Tick.**get_pad**(*self*)

Get the value of the tick label pad in points.

Examples using `matplotlib.axis.Tick.get_pad`

matplotlib.axis.Tick.get_pad_pixels

`Tick.get_pad_pixels(self)`

Examples using `matplotlib.axis.Tick.get_pad_pixels`

matplotlib.axis.Tick.get_tick_padding

`Tick.get_tick_padding(self)`
Get the length of the tick outside of the axes.

Examples using `matplotlib.axis.Tick.get_tick_padding`

matplotlib.axis.Tick.get_tickdir

`Tick.get_tickdir(self)`

Examples using `matplotlib.axis.Tick.get_tickdir`

matplotlib.axis.Tick.get_view_interval

`Tick.get_view_interval(self)`
Return the view limits `(min, max)` of the axis the tick belongs to.

Examples using `matplotlib.axis.Tick.get_view_interval`

matplotlib.axis.Tick.set_label1

`Tick.set_label1(self, s)`
Set the label1 text.

Parameters

s

[str]

Examples using `matplotlib.axis.Tick.set_label1`

matplotlib.axis.Tick.set_label2

`Tick.set_label2`(*self*, *s*)
Set the label2 text.

Parameters

s

[str]

Examples using `matplotlib.axis.Tick.set_label2`

matplotlib.axis.Tick.set_pad

`Tick.set_pad`(*self*, *val*)
Set the tick label pad in points

Parameters

val

[float]

Examples using `matplotlib.axis.Tick.set_pad`

matplotlib.axis.Tick.update_position

`Tick.update_position`(*self*, *loc*)
Set the location of tick in data coords with scalar *loc*.

Examples using `matplotlib.axis.Tick.update_position`

18.7 matplotlib.backend_bases

Abstract base classes define the primitives that renderers and graphics contexts must implement to serve as a Matplotlib backend.

RendererBase

An abstract base class to handle drawing/rendering operations.

FigureCanvasBase

The abstraction layer that separates the *Figure* from the backend specific details like a user interface drawing area.

GraphicsContextBase

An abstract base class that provides color, line styles, etc.

Event

The base class for all of the Matplotlib event handling. Derived classes such as *KeyEvent* and *MouseEvent* store the meta data like keys and buttons pressed, x and y locations in pixel and *Axes* coordinates.

ShowBase

The base class for the `Show` class of each interactive backend; the 'show' callable is then set to `Show.__call__`.

ToolContainerBase

The base class for the Toolbar class of each interactive backend.

class matplotlib.backend_bases.**CloseEvent**(*name*, *canvas*, *guiEvent=None*)

Bases: *matplotlib.backend_bases.Event*

An event triggered by a figure being closed.

class matplotlib.backend_bases.**DrawEvent**(*name*, *canvas*, *renderer*)

Bases: *matplotlib.backend_bases.Event*

An event triggered by a draw operation on the canvas

In most backends callbacks subscribed to this callback will be fired after the rendering is complete but before the screen is updated. Any extra artists drawn to the canvas's renderer will be reflected without an explicit call to `blit`.

> **Warning:** Calling `canvas.draw` and `canvas.blit` in these callbacks may not be safe with all backends and may cause infinite recursion.

In addition to the *Event* attributes, the following event attributes are defined:

Attributes

renderer

[*RendererBase*] The renderer for the draw event.

class matplotlib.backend_bases.**Event**(*name*, *canvas*, *guiEvent=None*)

Bases: `object`

A Matplotlib event. Attach additional attributes as defined in *FigureCanvasBase.mpl_connect()*. The following attributes are defined and shown with their default values

Attributes

name

[str] The event name.

canvas

[*FigureCanvasBase*] The backend-specific canvas instance generating the event.

guiEvent

The GUI event that triggered the Matplotlib event.

class matplotlib.backend_bases.**FigureCanvasBase**(*figure*)

Bases: object

The canvas the figure renders into.

Attributes

figure

[*matplotlib.figure.Figure*] A high-level figure instance.

blit(*self*, *bbox=None*)

Blit the canvas in bbox (default entire canvas).

button_press_event(*self*, *x*, *y*, *button*, *dblclick=False*, *guiEvent=None*)

Callback processing for mouse button press events.

Backend derived classes should call this function on any mouse button press. (*x*, *y*) are the canvas coords ((0, 0) is lower left). button and key are as defined in *MouseEvent*.

This method will call all functions connected to the 'button_press_event' with a *MouseEvent* instance.

button_release_event(*self*, *x*, *y*, *button*, *guiEvent=None*)

Callback processing for mouse button release events.

Backend derived classes should call this function on any mouse button release.

This method will call all functions connected to the 'button_release_event' with a *MouseEvent* instance.

Parameters

x

[float] The canvas coordinates where 0=left.

y

[float] The canvas coordinates where 0=bottom.

guiEvent

The native UI event that generated the Matplotlib event.

`close_event`(*self*, *guiEvent=None*)
Pass a *CloseEvent* to all functions connected to `close_event`.

`draw`(*self*, **args*, ***kwargs*)
Render the *Figure*.

`draw_cursor`(*self*, *event*)
[*Deprecated*] Draw a cursor in the event.axes if inaxes is not None. Use native GUI drawing for efficiency if possible

Notes

Deprecated since version 3.2.

`draw_event`(*self*, *renderer*)
Pass a *DrawEvent* to all functions connected to `draw_event`.

`draw_idle`(*self*, **args*, ***kwargs*)
Request a widget redraw once control returns to the GUI event loop.

Even if multiple calls to *draw_idle* occur before control returns to the GUI event loop, the figure will only be rendered once.

Notes

Backends may choose to override the method and implement their own strategy to prevent multiple renderings.

`enter_notify_event`(*self*, *guiEvent=None*, *xy=None*)
Callback processing for the mouse cursor entering the canvas.

Backend derived classes should call this function when entering canvas.

Parameters

guiEvent

The native UI event that generated the Matplotlib event.

xy

[(float, float)] The coordinate location of the pointer when the canvas is entered.

```
events = ['resize_event', 'draw_event', 'key_press_event', 'key_release_event', 'button_pr
```

```
filetypes = {'eps': 'Encapsulated Postscript', 'jpeg': 'Joint Photographic Experts Group'
```

```
fixed_dpi = None
```

`flush_events`(*self*)
Flush the GUI events for the figure.

Interactive backends need to reimplement this method.

`get_default_filename`(*self*)
Return a string, which includes extension, suitable for use as a default filename.

`classmethod get_default_filetype`()
Return the default savefig file format as specified in `rcParams["savefig.format"]` (default: `'png'`).

The returned string does not include a period. This method is overridden in backends that only support a single file type.

`classmethod get_supported_filetypes`()
Return dict of savefig file formats supported by this backend.

`classmethod get_supported_filetypes_grouped`()
Return a dict of savefig file formats supported by this backend, where the keys are a file type name, such as 'Joint Photographic Experts Group', and the values are a list of filename extensions used for that filetype, such as ['jpg', 'jpeg'].

`get_width_height`(*self*)
Return the figure width and height in points or pixels (depending on the backend), truncated to integers.

`get_window_title`(*self*)
Return the title text of the window containing the figure, or None if there is no window (e.g., a PS backend).

`grab_mouse`(*self*, *ax*)
Set the child *Axes* which is grabbing the mouse events.

Usually called by the widgets themselves. It is an error to call this if the mouse is already grabbed by another axes.

`inaxes`(*self*, *xy*)
Return the topmost visible *Axes* containing the point *xy*.

Parameters

xy

[(float, float)] (x, y) pixel positions from left/bottom of the canvas.

Returns

Axes **or None**

The topmost visible axes containing the point, or None if no axes.

`is_saving`(*self*)
Return whether the renderer is in the process of saving to a file, rather than rendering for an on-screen buffer.

`key_press_event`(*self*, *key*, *guiEvent=None*)
Pass a *KeyEvent* to all functions connected to `key_press_event`.

`key_release_event`(*self*, *key*, *guiEvent=None*)
Pass a *KeyEvent* to all functions connected to `key_release_event`.

`leave_notify_event`(*self*, *guiEvent=None*)
Callback processing for the mouse cursor leaving the canvas.

Backend derived classes should call this function when leaving canvas.

Parameters

guiEvent

The native UI event that generated the Matplotlib event.

`motion_notify_event`(*self*, *x*, *y*, *guiEvent=None*)
Callback processing for mouse movement events.

Backend derived classes should call this function on any motion-notify-event.

This method will call all functions connected to the 'motion_notify_event' with a *MouseEvent* instance.

Parameters

x

[float] The canvas coordinates where 0=left.

y

[float] The canvas coordinates where 0=bottom.

guiEvent

The native UI event that generated the Matplotlib event.

`mpl_connect`(*self*, *s*, *func*)
Bind function *func* to event *s*.

Parameters

s

[str] One of the following events ids:

- 'button_press_event'
- 'button_release_event'
- 'draw_event'

- 'key_press_event'
- 'key_release_event'
- 'motion_notify_event'
- 'pick_event'
- 'resize_event'
- 'scroll_event'
- 'figure_enter_event',
- 'figure_leave_event',
- 'axes_enter_event',
- 'axes_leave_event'
- 'close_event'.

func

[callable] The callback function to be executed, which must have the signature:

```
def func(event: Event) -> Any
```

For the location events (button and key press/release), if the mouse is over the axes, the `inaxes` attribute of the event will be set to the *Axes* the event occurs is over, and additionally, the variables `xdata` and `ydata` attributes will be set to the mouse location in data coordinates. See *KeyEvent* and *MouseEvent* for more info.

Returns

cid

A connection id that can be used with *FigureCanvasBase.mpl_disconnect*.

Examples

```
def on_press(event):
    print('you pressed', event.button, event.xdata, event.ydata)

cid = canvas.mpl_connect('button_press_event', on_press)
```

`mpl_disconnect`(*self*, *cid*)

Disconnect the callback with id *cid*.

Examples

```
cid = canvas.mpl_connect('button_press_event', on_press)
# ... later
canvas.mpl_disconnect(cid)
```

`new_timer`(*self*, *interval=None*, *callbacks=None*)
Create a new backend-specific subclass of `Timer`.

This is useful for getting periodic events through the backend's native event loop. Implemented only for backends with GUIs.

Parameters

interval

[int] Timer interval in milliseconds.

callbacks

[List[Tuple[callable, Tuple, Dict]]] Sequence of (func, args, kwargs) where `func(*args, **kwargs)` will be executed by the timer every *interval*.

Callbacks which return `False` or `0` will be removed from the timer.

Examples

```
>>> timer = fig.canvas.new_timer(callbacks=[(f1, (1,), {'a': 3})])
```

`pick`(*self*, *mouseevent*)

`pick_event`(*self*, *mouseevent*, *artist*, ***kwargs*)
Callback processing for pick events.

This method will be called by artists who are picked and will fire off *PickEvent* callbacks registered listeners.

`print_figure`(*self*, *filename*, *dpi=None*, *facecolor=None*, *edgecolor=None*, *orientation='portrait'*, *format=None*, ***, *bbox_inches=None*, *pad_inches=None*, *bbox_extra_artists=None*, *backend=None*, ***kwargs*)
Render the figure to hardcopy. Set the figure patch face and edge colors. This is useful because some of the GUIs have a gray figure face color background and you'll probably want to override this on hardcopy.

Parameters

filename

[str or path-like or file-like] The file where the figure is saved.

dpi

[float, default: `rcParams["savefig.dpi"]` (default: `'figure'`)] The dots per inch to save the figure in.

facecolor

[color or 'auto', default: `rcParams["savefig.facecolor"]` (default: `'auto'`)] The facecolor of the figure. If 'auto', use the current figure facecolor.

edgecolor

[color or 'auto', default: `rcParams["savefig.edgecolor"]` (default: `'auto'`)] The edgecolor of the figure. If 'auto', use the current figure edgecolor.

orientation

[{'landscape', 'portrait'}, default: 'portrait'] Only currently applies to PostScript printing.

format

[str, optional] Force a specific file format. If not given, the format is inferred from the *filename* extension, and if that fails from `rcParams["savefig.format"]` (default: `'png'`).

bbox_inches

['tight' or *Bbox*, default: `rcParams["savefig.bbox"]` (default: `None`)] Bounding box in inches: only the given portion of the figure is saved. If 'tight', try to figure out the tight bbox of the figure.

pad_inches

[float, default: `rcParams["savefig.pad_inches"]` (default: `0.1`)] Amount of padding around the figure when *bbox_inches* is 'tight'.

bbox_extra_artists

[list of *Artist*, optional] A list of extra artists that will be considered when the tight bbox is calculated.

backend

[str, optional] Use a non-default backend to render the file, e.g. to render a png file with the "cairo" backend rather than the default "agg", or a pdf file with the "pgf" backend rather than the default "pdf". Note that the default backend is normally sufficient. See *The builtin backends* for a list of valid backends for each file format. Custom backends can be referenced as "module://...".

`release_mouse`(*self*, *ax*)
Release the mouse grab held by the *Axes* *ax*.

Usually called by the widgets. It is ok to call this even if *ax* doesn't have the mouse grab currently.

`required_interactive_framework = None`

`resize`(*self*, *w*, *h*)
Set the canvas size in pixels.

`resize_event`(*self*)
Pass a *ResizeEvent* to all functions connected to `resize_event`.

`scroll_event`(*self*, *x*, *y*, *step*, *guiEvent=None*)
Callback processing for scroll events.

Backend derived classes should call this function on any scroll wheel event. (*x*, *y*) are the canvas coords ((0, 0) is lower left). button and key are as defined in *MouseEvent*.

This method will call all functions connected to the 'scroll_event' with a *MouseEvent* instance.

`set_window_title`(*self*, *title*)
Set the title text of the window containing the figure. Note that this has no effect if there is no window (e.g., a PS backend).

`start_event_loop`(*self*, *timeout=0*)
Start a blocking event loop.

Such an event loop is used by interactive functions, such as *ginput* and *waitforbuttonpress*, to wait for events.

The event loop blocks until a callback function triggers *stop_event_loop*, or *timeout* is reached.

If *timeout* is 0 or negative, never timeout.

Only interactive backends need to reimplement this method and it relies on *flush_events* being properly implemented.

Interactive backends should implement this in a more native way.

`stop_event_loop`(*self*)
Stop the current blocking event loop.

Interactive backends need to reimplement this to match *start_event_loop*

`supports_blit = False`

`switch_backends`(*self*, *FigureCanvasClass*)
Instantiate an instance of FigureCanvasClass

This is used for backend switching, e.g., to instantiate a FigureCanvasPS from a FigureCanvasGTK. Note, deep copying is not done, so any changes to

one of the instances (e.g., setting figure size or line props), will be reflected in the other

class matplotlib.backend_bases.**FigureManagerBase**(*canvas*, *num*)

Bases: object

A backend-independent abstraction of a figure container and controller.

The figure manager is used by pyplot to interact with the window in a backend-independent way. It's an adapter for the real (GUI) framework that represents the visual figure on screen.

GUI backends define from this class to translate common operations such as *show* or *resize* to the GUI-specific code. Non-GUI backends do not support these operations an can just use the base class.

This following basic operations are accessible:

Window operations

- *show*
- *destroy*
- *full_screen_toggle*
- *resize*
- *get_window_title*
- *set_window_title*

Key and mouse button press handling

The figure manager sets up default key and mouse button press handling by hooking up the *key_press_handler* to the matplotlib event system. This ensures the same shortcuts and mouse actions across backends.

Other operations

Subclasses will have additional attributes and functions to access additional functionality. This is of course backend-specific. For example, most GUI backends have `window` and `toolbar` attributes that give access to the native GUI widgets of the respective framework.

Attributes

canvas

[*FigureCanvasBase*] The backend-specific canvas instance.

num

[int or str] The figure number.

key_press_handler_id

[int] The default key handler cid, when using the toolmanager. To disable the default key press handling use:

```
figure.canvas.mpl_disconnect(
    figure.canvas.manager.key_press_handler_id)
```

button_press_handler_id

[int] The default mouse button handler cid, when using the toolmanager. To disable the default button press handling use:

```
figure.canvas.mpl_disconnect(
    figure.canvas.manager.button_press_handler_id)
```

`button_press`(*self*, *event*)
The default Matplotlib button actions for extra mouse buttons.

`destroy`(*self*)

`full_screen_toggle`(*self*)

`get_window_title`(*self*)
Return the title text of the window containing the figure, or None if there is no window (e.g., a PS backend).

`key_press`(*self*, *event*)
Implement the default Matplotlib key bindings defined at *Navigation Keyboard Shortcuts*.

`resize`(*self*, *w*, *h*)
For GUI backends, resize the window (in pixels).

`set_window_title`(*self*, *title*)
Set the title text of the window containing the figure.

This has no effect for non-GUI (e.g., PS) backends.

`show`(*self*)
For GUI backends, show the figure window and redraw. For non-GUI backends, raise an exception, unless running headless (i.e. on Linux with an unset DISPLAY); this exception is converted to a warning in *Figure.show*.

`property statusbar`

`class matplotlib.backend_bases.GraphicsContextBase`
Bases: `object`

An abstract base class that provides color, line styles, etc.

`copy_properties`(*self*, *gc*)
Copy properties from *gc* to self.

`get_alpha`(*self*)
Return the alpha value used for blending - not supported on all backends.

`get_antialiased`(*self*)
Return whether the object should try to do antialiased rendering.

get_capstyle(*self*)
Return the capstyle as a string in ('butt', 'round', 'projecting').

get_clip_path(*self*)
Return the clip path in the form (path, transform), where path is a *Path* instance, and transform is an affine transform to apply to the path before clipping.

get_clip_rectangle(*self*)
Return the clip rectangle as a *Bbox* instance.

get_dashes(*self*)
Return the dash style as an (offset, dash-list) pair.

The dash list is a even-length list that gives the ink on, ink off in points. See p. 107 of to PostScript blue book for more info.

Default value is (None, None).

get_forced_alpha(*self*)
Return whether the value given by get_alpha() should be used to override any other alpha-channel values.

get_gid(*self*)
Return the object identifier if one is set, None otherwise.

get_hatch(*self*)
Get the current hatch style.

get_hatch_color(*self*)
Get the hatch color.

get_hatch_linewidth(*self*)
Get the hatch linewidth.

get_hatch_path(*self*, *density=6.0*)
Return a *Path* for the current hatch.

get_joinstyle(*self*)
Return the line join style as one of ('miter', 'round', 'bevel').

get_linewidth(*self*)
Return the line width in points.

get_rgb(*self*)
Return a tuple of three or four floats from 0-1.

get_sketch_params(*self*)
Return the sketch parameters for the artist.

Returns

tuple or `None`

A 3-tuple with the following elements:

- `scale`: The amplitude of the wiggle perpendicular to the source line.
- `length`: The length of the wiggle along the line.
- `randomness`: The scale factor by which the length is shrunken or expanded.

May return `None` if no sketch parameters were set.

`get_snap`(*self*)
Return the snap setting, which can be:

- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

`get_url`(*self*)
Return a url if one is set, None otherwise.

`restore`(*self*)
Restore the graphics context from the stack - needed only for backends that save graphics contexts on a stack.

`set_alpha`(*self*, *alpha*)
Set the alpha value used for blending - not supported on all backends.

If `alpha=None` (the default), the alpha components of the foreground and fill colors will be used to set their respective transparencies (where applicable); otherwise, `alpha` will override them.

`set_antialiased`(*self*, *b*)
Set whether object should be drawn with antialiased rendering.

`set_capstyle`(*self*, *cs*)
Set the capstyle to be one of ('butt', 'round', 'projecting').

`set_clip_path`(*self*, *path*)
Set the clip path and transformation.

Parameters

path

[*TransformedPath* or None]

`set_clip_rectangle`(*self*, *rectangle*)
Set the clip rectangle with sequence (left, bottom, width, height)

`set_dashes`(*self*, *dash_offset*, *dash_list*)
Set the dash style for the gc.

Parameters

dash_offset

[float or None] The offset (usually 0).

dash_list

[array-like or None] The on-off sequence as points.

Notes

`(None, None)` specifies a solid line.

See p. 107 of to PostScript blue book for more info.

`set_foreground`(*self*, *fg*, *isRGBA=False*)
Set the foreground color.

Parameters

fg

[color]

isRGBA

[bool] If *fg* is known to be an `(r, g, b, a)` tuple, *isRGBA* can be set to True to improve performance.

`set_gid`(*self*, *id*)
Set the id.

`set_hatch`(*self*, *hatch*)
Set the hatch style (for fills).

`set_hatch_color`(*self*, *hatch_color*)
Set the hatch color.

`set_joinstyle`(*self*, *js*)
Set the join style to be one of ('miter', 'round', 'bevel').

`set_linewidth`(*self*, *w*)
Set the linewidth in points.

`set_sketch_params`(*self*, *scale=None*, *length=None*, *randomness=None*)
Set the sketch parameters.

Parameters

scale

[float, optional] The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is `None`, or not provided, no sketch filter will be provided.

length

[float, default: 128] The length of the wiggle along the line, in pixels.

randomness

[float, default: 16] The scale factor by which the length is shrunken or expanded.

`set_snap`(*self*, *snap*)
Set the snap setting which may be:

- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

`set_url`(*self*, *url*)
Set the url for links in compatible backends.

class matplotlib.backend_bases.**KeyEvent**(*name*, *canvas*, *key*, *x=0*, *y=0*, *guiEvent=None*)

Bases: *matplotlib.backend_bases.LocationEvent*

A key event (key press, key release).

Attach additional attributes as defined in *FigureCanvasBase.mpl_connect()*.

In addition to the *Event* and *LocationEvent* attributes, the following attributes are defined:

Notes

Modifier keys will be prefixed to the pressed key and will be in the order "ctrl", "alt", "super". The exception to this rule is when the pressed key is itself a modifier key, therefore "ctrl+alt" and "alt+control" can both be valid key values.

Examples

```
def on_key(event):
    print('you pressed', event.key, event.xdata, event.ydata)

cid = fig.canvas.mpl_connect('key_press_event', on_key)
```

Attributes

key

[None or str] the key(s) pressed. Could be **None**, a single case sensitive ascii character ("g", "G", "#", etc.), a special key ("control", "shift", "f1", "up", etc.) or a combination of the above (e.g., "ctrl+alt+g", "ctrl+alt+G").

(*x*, *y*) in figure coords ((0, 0) = bottom left).

class matplotlib.backend_bases.LocationEvent(*name*, *canvas*, *x*, *y*, *guiEvent=None*)

Bases: *matplotlib.backend_bases.Event*

An event that has a screen location.

The following additional attributes are defined and shown with their default values.

In addition to the *Event* attributes, the following event attributes are defined:

Attributes

x

[int] x position - pixels from left of canvas.

y

[int] y position - pixels from bottom of canvas.

inaxes

[*Axes* or None] The *Axes* instance over which the mouse is, if any.

xdata

[float or None] x data coordinate of the mouse.

ydata

[float or None] y data coordinate of the mouse.

(*x*, *y*) in figure coords ((0, 0) = bottom left).

lastevent = None

class matplotlib.backend_bases.MouseButton(*value*)

Bases: enum.IntEnum

An enumeration.

BACK = 8

FORWARD = 9

LEFT = 1

MIDDLE = 2

RIGHT = 3

class matplotlib.backend_bases.MouseEvent(*name*, *canvas*, *x*, *y*, *button=None*, *key=None*, *step=0*, *dblclick=False*, *guiEvent=None*)

Bases: *matplotlib.backend_bases.LocationEvent*

A mouse event ('button_press_event',

'button_release_event', 'scroll_event', 'motion_notify_event').

In addition to the *Event* and *LocationEvent* attributes, the following attributes are defined:

Examples

```
def on_press(event):
    print('you pressed', event.button, event.xdata, event.ydata)

cid = fig.canvas.mpl_connect('button_press_event', on_press)
```

Attributes

button

[None or *MouseButton* or {'up', 'down'}] The button pressed. 'up' and 'down' are used for scroll events. Note that in the nbagg backend, both the middle and right clicks return RIGHT since right clicking will bring up the context menu in some browsers. Note that LEFT and RIGHT actually refer to the "primary" and "secondary" buttons, i.e. if the user inverts their left and right buttons ("left-handed setting") then the LEFT button will be the one physically on the right.

key

[None or str] The key pressed when the mouse event triggered, e.g. 'shift'. See *KeyEvent*.

> **Warning:** This key is currently obtained from the last 'key_press_event' or 'key_release_event' that occurred within the canvas. Thus, if the last change of keyboard state occurred while the canvas did not have focus, this attribute will be wrong.

step

[float] The number of scroll steps (positive for 'up', negative for 'down'). This applies only to 'scroll_event' and defaults to 0 otherwise.

dblclick

[bool] Whether the event is a double-click. This applies only to 'button_press_event' and is False otherwise. In particular, it's not used in 'button_release_event'.

(*x*, *y*) in figure coords ((0, 0) = bottom left) button pressed None, 1, 2, 3, 'up', 'down'

class matplotlib.backend_bases.**NavigationToolbar2**(*canvas*)

Bases: object

Base class for the navigation cursor, version 2.

Backends must implement a canvas that handles connections for 'button_press_event' and 'button_release_event'. See *FigureCanvasBase.mpl_connect()* for more information.

They must also define

save_figure()

save the current figure

set_cursor()

if you want the pointer icon to change

draw_rubberband() **(optional)**

draw the zoom to rect "rubberband" rectangle

set_message() **(optional)**

display message

set_history_buttons() **(optional)**

you can change the history back / forward buttons to indicate disabled / enabled state.

and override `__init__` to set up the toolbar -- without forgetting to call the base-class init. Typically, `__init__` needs to set up toolbar buttons connected to the *home*, *back*, *forward*, *pan*, *zoom*, and *save_figure* methods and using standard icons in the "images" subdirectory of the data path.

That's it, we'll do the rest!

back(*self*, **args*)

Move back up the view lim stack.

For convenience of being directly connected as a GUI callback, which often get passed additional parameters, this method accepts arbitrary parameters, but does not use them.

drag_pan(*self*, *event*)

Callback for dragging in pan/zoom mode.

`drag_zoom`(*self*, *event*)
Callback for dragging in zoom mode.

`draw`(*self*)
[*Deprecated*] Redraw the canvases, update the locators.

Notes

Deprecated since version 3.3.

`draw_rubberband`(*self*, *event*, *x0*, *y0*, *x1*, *y1*)
Draw a rectangle rubberband to indicate zoom limits.

Note that it is not guaranteed that `x0 <= x1` and `y0 <= y1`.

`forward`(*self*, **args*)
Move forward in the view lim stack.

For convenience of being directly connected as a GUI callback, which often get passed additional parameters, this method accepts arbitrary parameters, but does not use them.

`home`(*self*, **args*)
Restore the original view.

For convenience of being directly connected as a GUI callback, which often get passed additional parameters, this method accepts arbitrary parameters, but does not use them.

`mouse_move`(*self*, *event*)

`pan`(*self*, **args*)
Toggle the pan/zoom tool.

Pan with left button, zoom with right.

`press`(*self*, *event*)
[*Deprecated*] Called whenever a mouse button is pressed.

Notes

Deprecated since version 3.3.

`press_pan`(*self*, *event*)
Callback for mouse button press in pan/zoom mode.

`press_zoom`(*self*, *event*)
Callback for mouse button press in zoom to rect mode.

`push_current`(*self*)
Push the current view limits and position onto the stack.

release(*self*, *event*)
[*Deprecated*] Callback for mouse button release.

Notes

Deprecated since version 3.3.

release_pan(*self*, *event*)
Callback for mouse button release in pan/zoom mode.

release_zoom(*self*, *event*)
Callback for mouse button release in zoom to rect mode.

remove_rubberband(*self*)
Remove the rubberband.

save_figure(*self*, **args*)
Save the current figure.

set_cursor(*self*, *cursor*)
Set the current cursor to one of the `Cursors` enums values.

If required by the backend, this method should trigger an update in the backend event loop after the cursor is set, as this method may be called e.g. before a long-running task during which the GUI is not updated.

set_history_buttons(*self*)
Enable or disable the back/forward button.

set_message(*self*, *s*)
Display a message on toolbar or in status bar.

toolitems = (('Home', 'Reset original view', 'home', 'home'), ('Back', 'Back to previous v

update(*self*)
Reset the axes stack.

zoom(*self*, **args*)
Toggle zoom to rect mode.

exception matplotlib.backend_bases.**NonGuiException**
Bases: `Exception`

Raised when trying show a figure in a non-GUI backend.

class matplotlib.backend_bases.**PickEvent**(*name*, *canvas*, *mouseevent*, *artist*, *guiEvent=None*, ***kwargs*)
Bases: *matplotlib.backend_bases.Event*

A pick event, fired when the user picks a location on the canvas sufficiently close to an artist.

Attrs: all the *Event* attributes plus

Examples

Bind a function `on_pick()` to pick events, that prints the coordinates of the picked data point:

```
ax.plot(np.rand(100), 'o', picker=5)  # 5 points tolerance

def on_pick(event):
    line = event.artist
    xdata, ydata = line.get_data()
    ind = event.ind
    print('on pick line:', np.array([xdata[ind], ydata[ind]]).T)

cid = fig.canvas.mpl_connect('pick_event', on_pick)
```

Attributes

mouseevent

[*MouseEvent*] The mouse event that generated the pick.

artist

[*matplotlib.artist.Artist*] The picked artist.

other

Additional attributes may be present depending on the type of the picked object; e.g., a *Line2D* pick may define different extra attributes than a *PatchCollection* pick.

class matplotlib.backend_bases.**RendererBase**

Bases: `object`

An abstract base class to handle drawing/rendering operations.

The following methods must be implemented in the backend for full functionality (though just implementing *draw_path()* alone would give a highly capable backend):

- *draw_path()*
- *draw_image()*
- *draw_gouraud_triangle()*

The following methods *should* be implemented in the backend for optimization reasons:

- *draw_text()*
- *draw_markers()*
- *draw_path_collection()*
- *draw_quad_mesh()*

`close_group`(*self*, *s*)
Close a grouping element with label *s*.

Only used by the SVG renderer.

`draw_gouraud_triangle`(*self*, *gc*, *points*, *colors*, *transform*)
Draw a Gouraud-shaded triangle.

Parameters

gc

[*GraphicsContextBase*] The graphics context.

points

[array-like, shape=(3, 2)] Array of (x, y) points for the triangle.

colors

[array-like, shape=(3, 4)] RGBA colors for each point of the triangle.

transform

[*matplotlib.transforms.Transform*] An affine transform to apply to the points.

`draw_gouraud_triangles`(*self*, *gc*, *triangles_array*, *colors_array*, *transform*)
Draw a series of Gouraud triangles.

Parameters

points

[array-like, shape=(N, 3, 2)] Array of *N* (x, y) points for the triangles.

colors

[array-like, shape=(N, 3, 4)] Array of *N* RGBA colors for each point of the triangles.

transform

[*matplotlib.transforms.Transform*] An affine transform to apply to the points.

`draw_image`(*self*, *gc*, *x*, *y*, *im*, *transform=None*)
Draw an RGBA image.

Parameters

gc

[*GraphicsContextBase*] A graphics context with clipping information.

x

[scalar] The distance in physical units (i.e., dots or pixels) from the left hand side of the canvas.

y

[scalar] The distance in physical units (i.e., dots or pixels) from the bottom side of the canvas.

im

[array-like, shape=(N, M, 4), dtype=np.uint8] An array of RGBA pixels.

transform

[*matplotlib.transforms.Affine2DBase*] If and only if the concrete backend is written such that *option_scale_image()* returns `True`, an affine transformation (i.e., an *Affine2DBase*) *may* be passed to *draw_image()*. The translation vector of the transformation is given in physical units (i.e., dots or pixels). Note that the transformation does not override *x* and *y*, and has to be applied *before* translating the result by *x* and *y* (this can be accomplished by adding *x* and *y* to the translation vector defined by *transform*).

`draw_markers`(*self*, *gc*, *marker_path*, *marker_trans*, *path*, *trans*, *rgbFace=None*)

Draw a marker at each of the vertices in path.

This includes all vertices, including control points on curves. To avoid that behavior, those vertices should be removed before calling this function.

This provides a fallback implementation of draw_markers that makes multiple calls to *draw_path()*. Some backends may want to override this method in order to draw the marker only once and reuse it multiple times.

Parameters

gc

[*GraphicsContextBase*] The graphics context.

marker_trans

[*matplotlib.transforms.Transform*] An affine transform applied to the marker.

trans

[*matplotlib.transforms.Transform*] An affine transform applied to the path.

draw_path(*self*, *gc*, *path*, *transform*, *rgbFace=None*)
Draw a *Path* instance using the given affine transform.

draw_path_collection(*self*, *gc*, *master_transform*, *paths*, *all_transforms*, *offsets*, *offsetTrans*, *facecolors*, *edgecolors*, *linewidths*, *linestyles*, *antialiaseds*, *urls*, *offset_position*)
Draw a collection of paths selecting drawing properties from the lists *facecolors*, *edgecolors*, *linewidths*, *linestyles* and *antialiaseds*. *offsets* is a list of offsets to apply to each of the paths. The offsets in *offsets* are first transformed by *offsetTrans* before being applied.

offset_position may be either "screen" or "data" depending on the space that the offsets are in; "data" is deprecated.

This provides a fallback implementation of *draw_path_collection()* that makes multiple calls to *draw_path()*. Some backends may want to override this in order to render each set of path data only once, and then reference that path multiple times with the different offsets, colors, styles etc. The generator methods `_iter_collection_raw_paths()` and `_iter_collection()` are provided to help with (and standardize) the implementation across backends. It is highly recommended to use those generators, so that changes to the behavior of *draw_path_collection()* can be made globally.

draw_quad_mesh(*self*, *gc*, *master_transform*, *meshWidth*, *meshHeight*, *coordinates*, *offsets*, *offsetTrans*, *facecolors*, *antialiased*, *edgecolors*)
Fallback implementation of *draw_quad_mesh()* that generates paths and then calls *draw_path_collection()*.

draw_tex(*self*, *gc*, *x*, *y*, *s*, *prop*, *angle*, *ismath=<deprecated parameter>*, *mtext=None*)

draw_text(*self*, *gc*, *x*, *y*, *s*, *prop*, *angle*, *ismath=False*, *mtext=None*)
Draw the text instance.

Parameters

gc

[*GraphicsContextBase*] The graphics context.

x

[float] The x location of the text in display coords.

y

[float] The y location of the text baseline in display coords.

s

[str] The text string.

prop

[*matplotlib.font_manager.FontProperties*] The font properties.

angle

[float] The rotation angle in degrees anti-clockwise.

mtext

[*matplotlib.text.Text*] The original text object to be rendered.

Notes

Note for backend implementers:

When you are trying to determine if you have gotten your bounding box right (which is what enables the text layout/alignment to work properly), it helps to change the line in text.py:

```
if 0: bbox_artist(self, renderer)
```

to if 1, and then the actual bounding box will be plotted along with your text.

`flipy`(*self*)
Return whether y values increase from top to bottom.

Note that this only affects drawing of texts and images.

`get_canvas_width_height`(*self*)
Return the canvas width and height in display coords.

`get_image_magnification`(*self*)
Get the factor by which to magnify images passed to *draw_image()*. Allows a backend to have images at a different resolution to other artists.

`get_texmanager`(*self*)
Return the *TexManager* instance.

`get_text_width_height_descent`(*self*, *s*, *prop*, *ismath*)
Get the width, height, and descent (offset from the bottom to the baseline), in display coords, of the string *s* with *FontProperties* *prop*.

`new_gc`(*self*)
Return an instance of a *GraphicsContextBase*.

`open_group`(*self*, *s*, *gid=None*)
Open a grouping element with label *s* and *gid* (if set) as id.

Only used by the SVG renderer.

`option_image_nocomposite`(*self*)
Return whether image composition by Matplotlib should be skipped.

Raster backends should usually return False (letting the C-level rasterizer take care of image composition); vector backends should usually return `not rcParams["image.composite_image"]`.

option_scale_image(*self*)
Return whether arbitrary affine transformations in *draw_image()* are supported (True for most vector backends).

points_to_pixels(*self*, *points*)
Convert points to display units.

You need to override this function (unless your backend doesn't have a dpi, e.g., postscript or svg). Some imaging systems assume some value for pixels per inch:

```
points to pixels = points * pixels_per_inch/72 * dpi/72
```

Parameters

points

[float or array-like] a float or a numpy array of float

Returns

Points converted to pixels

start_filter(*self*)
Switch to a temporary renderer for image filtering effects.

Currently only supported by the agg renderer.

start_rasterizing(*self*)
Switch to the raster renderer.

Used by *MixedModeRenderer*.

stop_filter(*self*, *filter_func*)
Switch back to the original renderer. The contents of the temporary renderer is processed with the *filter_func* and is drawn on the original renderer as an image.

Currently only supported by the agg renderer.

stop_rasterizing(*self*)
Switch back to the vector renderer and draw the contents of the raster renderer as an image on the vector renderer.

Used by *MixedModeRenderer*.

class matplotlib.backend_bases.**ResizeEvent**(*name*, *canvas*)
Bases: *matplotlib.backend_bases.Event*

An event triggered by a canvas resize

In addition to the *Event* attributes, the following event attributes are defined:

Attributes

width

[int] Width of the canvas in pixels.

height

[int] Height of the canvas in pixels.

class matplotlib.backend_bases.**ShowBase**

Bases: matplotlib.backend_bases._Backend

Simple base class to generate a `show()` function in backends.

Subclass must override `mainloop()` method.

class matplotlib.backend_bases.**StatusbarBase**(***kwargs*)

Bases: object

[*Deprecated*] Base class for the statusbar.

Notes

Deprecated since version 3.3.

set_message(*self*, *s*)

Display a message on toolbar or in status bar.

Parameters

s

[str] Message text.

class matplotlib.backend_bases.**TimerBase**(*interval=None*, *callbacks=None*)

Bases: object

A base class for providing timer events, useful for things animations. Backends need to implement a few specific methods in order to use their own timing mechanisms so that the timer events are integrated into their event loops.

Subclasses must override the following methods:

- `_timer_start`: Backend-specific code for starting the timer.
- `_timer_stop`: Backend-specific code for stopping the timer.

Subclasses may additionally override the following methods:

- `_timer_set_single_shot`: Code for setting the timer to single shot operating mode, if supported by the timer object. If not, the `Timer` class itself will store the flag and the `_on_timer` method should be overridden to support such behavior.
- `_timer_set_interval`: Code for setting the interval on the timer, if there is a method for doing so on the timer object.

- `_on_timer`: The internal function that any timer object should call, which will handle the task of running all callbacks that have been set.

  Parameters

  interval

  [int, default: 1000ms] The time between timer events in milliseconds. Will be stored as `timer.interval`.

  callbacks

  [List[Tuple[callable, Tuple, Dict]]] List of (func, args, kwargs) tuples that will be called upon timer events. This list is accessible as `timer.callbacks` and can be manipulated directly, or the functions *add_callback* and *remove_callback* can be used.

`add_callback`(*self*, *func*, **args*, ***kwargs*)
: Register *func* to be called by timer when the event fires. Any additional arguments provided will be passed to *func*.

  This function returns *func*, which makes it possible to use it as a decorator.

`property interval`
: The time between timer events, in milliseconds.

`remove_callback`(*self*, *func*, **args*, ***kwargs*)
: Remove *func* from list of callbacks.

  args and *kwargs* are optional and used to distinguish between copies of the same function registered to be called with different arguments. This behavior is deprecated. In the future, `*args, **kwargs` won't be considered anymore; to keep a specific callback removable by itself, pass it to *add_callback* as a `functools.partial` object.

`property single_shot`
: Whether this timer should stop after a single run.

`start`(*self*, *interval=None*)
: Start the timer object.

  Parameters

  interval

  [int, optional] Timer interval in milliseconds; overrides a previously set interval if provided.

`stop`(*self*)
: Stop the timer.

`class matplotlib.backend_bases.ToolContainerBase`(*toolmanager*)
: Bases: `object`

Base class for all tool containers, e.g. toolbars.

Attributes

toolmanager

[*ToolManager*] The tools with which this `ToolContainer` wants to communicate.

`add_tool`(*self*, *tool*, *group*, *position=- 1*)
Add a tool to this container.

Parameters

tool

[tool_like] The tool to add, see *ToolManager.get_tool*.

group

[str] The name of the group to add this tool to.

position

[int, default: -1] The position within the group to place this tool.

`add_toolitem`(*self*, *name*, *group*, *position*, *image*, *description*, *toggle*)
Add a toolitem to the container.

This method must be implemented per backend.

The callback associated with the button click event, must be *exactly* `self.trigger_tool(name)`.

Parameters

name

[str] Name of the tool to add, this gets used as the tool's ID and as the default label of the buttons.

group

[str] Name of the group that this tool belongs to.

position

[int] Position of the tool within its group, if -1 it goes at the end.

image_file

[str] Filename of the image for the button or `None`.

description

[str] Description of the tool, used for the tooltips.

toggle

[bool]

- `True` : The button is a toggle (change the pressed/unpressed state between consecutive clicks).
- `False` : The button is a normal button (returns to unpressed state after release).

`remove_toolitem`(*self*, *name*)
Remove a toolitem from the `ToolContainer`.

This method must get implemented per backend.

Called when *`ToolManager`* emits a `tool_removed_event`.

Parameters

name

[str] Name of the tool to remove.

`set_message`(*self*, *s*)
Display a message on the toolbar.

Parameters

s

[str] Message text.

`toggle_toolitem`(*self*, *name*, *toggled*)
Toggle the toolitem without firing event.

Parameters

name

[str] Id of the tool to toggle.

toggled

[bool] Whether to set this tool as toggled or not.

`trigger_tool`(*self*, *name*)
Trigger the tool.

Parameters

name

[str] Name (id) of the tool triggered from within the container.

matplotlib.backend_bases.**button_press_handler**(*event*, *canvas=None*, *toolbar=None*)

The default Matplotlib button actions for extra mouse buttons.

Parameters are as for *key_press_handler*, except that *event* is a *MouseEvent*.

matplotlib.backend_bases.**get_registered_canvas_class**(*format*)

Return the registered default canvas for given file format. Handles deferred import of required backend.

matplotlib.backend_bases.**key_press_handler**(*event*, *canvas=None*, *toolbar=None*)

Implement the default Matplotlib key bindings for the canvas and toolbar described at *Navigation Keyboard Shortcuts*.

Parameters

event

[*KeyEvent*] A key press/release event.

canvas

[*FigureCanvasBase*, default: `event.canvas`] The backend-specific canvas instance. This parameter is kept for back-compatibility, but, if set, should always be equal to `event.canvas`.

toolbar

[*NavigationToolbar2*, default: `event.canvas.toolbar`] The navigation cursor toolbar. This parameter is kept for back-compatibility, but, if set, should always be equal to `event.canvas.toolbar`.

matplotlib.backend_bases.**register_backend**(*format*, *backend*, *description=None*)

Register a backend for saving to a given file format.

Parameters

format

[str] File extension

backend

[module string or canvas class] Backend for handling file output

description

[str, default: ""] Description of the file type.

18.8 matplotlib.backend_managers

class matplotlib.backend_managers.**ToolEvent**(*name*, *sender*, *tool*, *data=None*)

Bases: object

Event for tool manipulation (add/remove).

class matplotlib.backend_managers.**ToolManager**(*figure=None*)

Bases: object

Manager for actions triggered by user interactions (key press, toolbar clicks, ...) on a Figure.

Attributes

figure

[*Figure*] Figure that holds the canvas.

keypresslock

[*LockDraw*] *LockDraw* object to know if the *canvas* key_press_event is locked.

messagelock

[*LockDraw*] *LockDraw* object to know if the message is available to write.

property **active_toggle**

Currently toggled tools.

add_tool(*self*, *name*, *tool*, **args*, ***kwargs*)

Add *tool* to *ToolManager*.

If successful, adds a new event `tool_trigger_{name}` where `{name}` is the *name* of the tool; the event is fired every time the tool is triggered.

Parameters

name

[str] Name of the tool, treated as the ID, has to be unique.

tool

[class_like, i.e. str or type] Reference to find the class of the Tool to added.

See also:

matplotlib.backend_tools.ToolBase

The base class for tools.

Notes

args and kwargs get passed directly to the tools constructor.

`property canvas`
Canvas managed by FigureManager.

`property figure`
Figure that holds the canvas.

`get_tool(self, name, warn=True)`
Return the tool object with the given name.

For convenience, this passes tool objects through.

Parameters

name

[str or *ToolBase*] Name of the tool, or the tool itself.

warn

[bool, default: True] Whether a warning should be emitted it no tool with the given name exists.

Returns

ToolBase **or None**

The tool or None if no tool with the given name exists.

`get_tool_keymap(self, name)`
Return the keymap associated with the specified tool.

Parameters

name

[str] Name of the Tool.

Returns

list of str

List of keys associated with the tool.

`message_event(self, message, sender=None)`
Emit a *ToolManagerMessageEvent*.

`remove_tool(self, name)`
Remove tool named *name*.

Parameters

name

[str] Name of the tool.

`set_figure`(*self*, *figure*, *update_tools=True*)
Bind the given figure to the tools.

Parameters

figure

[*Figure*]

update_tools

[bool, default: True] Force tools to update figure.

`toolmanager_connect`(*self*, *s*, *func*)
Connect event with string *s* to *func*.

Parameters

s

[str] The name of the event. The following events are recognized:

- 'tool_message_event'
- 'tool_removed_event'
- 'tool_added_event'

For every tool added a new event is created

- 'tool_trigger_TOOLNAME', where TOOLNAME is the id of the tool.

func

[callable] Callback function for the toolmanager event with signature:

```
def func(event: ToolEvent) -> Any
```

Returns

cid

The callback id for the connection. This can be used in *toolmanager_disconnect*.

`toolmanager_disconnect`(*self*, *cid*)
Disconnect callback id *cid*.

Example usage:

```
cid = toolmanager.toolmanager_connect('tool_trigger_zoom', onpress)
#...later
toolmanager.toolmanager_disconnect(cid)
```

property tools
A dict mapping tool name -> controlled tool.

trigger_tool(*self*, *name*, *sender=None*, *canvasevent=None*, *data=None*)
Trigger a tool and emit the tool_trigger_{name} event.

Parameters

name

[str] Name of the tool.

sender

[object] Object that wishes to trigger the tool.

canvasevent

[Event] Original Canvas event or None.

data

[object] Extra data to pass to the tool when triggering.

update_keymap(*self*, *name*, *key*, **args*)
Set the keymap to associate with the specified tool.

Parameters

name

[str] Name of the Tool.

keys

[str or list of str] Keys to associate with the tool.

class matplotlib.backend_managers.ToolManagerMessageEvent(*name*, *sender*, *message*)
Bases: object

Event carrying messages from toolmanager.

Messages usually get displayed to the user by the toolbar.

class matplotlib.backend_managers.ToolTriggerEvent(*name*, *sender*, *tool*, *canvasevent=None*, *data=None*)
Bases: *matplotlib.backend_managers.ToolEvent*

Event to inform that a tool has been triggered.

18.9 matplotlib.backend_tools

Abstract base classes define the primitives for Tools. These tools are used by *matplotlib.backend_managers.ToolManager*

ToolBase

Simple stateless tool

ToolToggleBase

Tool that has two states, only one Toggle tool can be active at any given time for the same *matplotlib.backend_managers.ToolManager*

class matplotlib.backend_tools.**AxisScaleBase**(**args*, ***kwargs*)

Bases: *matplotlib.backend_tools.ToolToggleBase*

Base Tool to toggle between linear and logarithmic.

disable(*self*, *event*)

Disable the toggle tool.

trigger call this method when `toggled` is True.

This can happen in different circumstances.

- Click on the toolbar tool button.
- Call to *matplotlib.backend_managers.ToolManager.trigger_tool*.
- Another *ToolToggleBase* derived tool is triggered (from the same *ToolManager*).

enable(*self*, *event*)

Enable the toggle tool.

trigger calls this method when `toggled` is False.

trigger(*self*, *sender*, *event*, *data=None*)

Calls *enable* or *disable* based on `toggled` value.

class matplotlib.backend_tools.**ConfigureSubplotsBase**(*toolmanager*, *name*)

Bases: *matplotlib.backend_tools.ToolBase*

Base tool for the configuration of subplots.

description = 'Configure subplots'

image = 'subplots'

class matplotlib.backend_tools.**Cursors**(*value*)

Bases: `enum.IntEnum`

Backend-independent cursor types.

HAND = 0

MOVE = 3

POINTER = 1

SELECT_REGION = 2

WAIT = 4

class matplotlib.backend_tools.**RubberbandBase**(*toolmanager*, *name*)

Bases: *matplotlib.backend_tools.ToolBase*

Draw and remove a rubberband.

draw_rubberband(*self*, **data*)

Draw rubberband.

This method must get implemented per backend.

remove_rubberband(*self*)

Remove rubberband.

This method should get implemented per backend.

trigger(*self*, *sender*, *event*, *data*)

Call *draw_rubberband* or *remove_rubberband* based on data.

class matplotlib.backend_tools.**SaveFigureBase**(*toolmanager*, *name*)

Bases: *matplotlib.backend_tools.ToolBase*

Base tool for figure saving.

default_keymap = ['s', 'ctrl+s']

description = 'Save the figure'

image = 'filesave'

class matplotlib.backend_tools.**SetCursorBase**(**args*, ***kwargs*)

Bases: *matplotlib.backend_tools.ToolBase*

Change to the current cursor while inaxes.

This tool, keeps track of all *ToolToggleBase* derived tools, and calls *set_cursor* when a tool gets triggered.

set_cursor(*self*, *cursor*)

Set the cursor.

This method has to be implemented per backend.

set_figure(*self*, *figure*)

Assign a figure to the tool.

Parameters

figure

[*Figure*]

class matplotlib.backend_tools.ToolBack(*toolmanager*, *name*)

Bases: *matplotlib.backend_tools.ViewsPositionsBase*

Move back up the view limits stack.

default_keymap = ['left', 'c', 'backspace', 'MouseButton.BACK']

description = 'Back to previous view'

image = 'back'

class matplotlib.backend_tools.ToolBase(*toolmanager*, *name*)

Bases: object

Base tool class.

A base tool, only implements *trigger* method or not method at all. The tool is instantiated by *matplotlib.backend_managers.ToolManager*.

Attributes

toolmanager

[*matplotlib.backend_managers.ToolManager*] ToolManager that controls this Tool.

figure

[FigureCanvas] Figure instance that is affected by this Tool.

name

[str] Tool Id.

property canvas

default_keymap = None

Keymap to associate with this tool.

String: List of comma separated keys that will be used to call this tool when the keypress event of `self.figure.canvas` is emitted.

description = None

Description of the Tool.

String: If the Tool is included in the Toolbar this text is used as a Tooltip.

destroy(*self*)

Destroy the tool.

This method is called when the tool is removed by *matplotlib.backend_managers.ToolManager.remove_tool*.

property figure

image = None

Filename of the image.

String: Filename of the image to use in the toolbar. If None, the *name* is used as a label in the toolbar button.

property name
Tool Id.

set_figure(*self*, *figure*)
Assign a figure to the tool.

Parameters

figure

[*Figure*]

property toolmanager

trigger(*self*, *sender*, *event*, *data=None*)
Called when this tool gets used.

This method is called by *matplotlib.backend_managers.ToolManager.trigger_tool*.

Parameters

event

[*Event*] The canvas event that caused this tool to be called.

sender

[object] Object that requested the tool to be triggered.

data

[object] Extra data.

matplotlib.backend_tools.ToolCopyToClipboard
alias of *matplotlib.backend_tools.ToolCopyToClipboardBase*

class matplotlib.backend_tools.ToolCopyToClipboardBase(*toolmanager*, *name*)
Bases: *matplotlib.backend_tools.ToolBase*

Tool to copy the figure to the clipboard.

default_keymap = ['ctrl+c', 'cmd+c']

description = 'Copy the canvas figure to clipboard'

trigger(*self*, **args*, ***kwargs*)
Called when this tool gets used.

This method is called by *matplotlib.backend_managers.ToolManager.trigger_tool*.

Parameters

event

[*Event*] The canvas event that caused this tool to be called.

sender

[object] Object that requested the tool to be triggered.

data

[object] Extra data.

class matplotlib.backend_tools.**ToolCursorPosition**(**args*, ***kwargs*)

Bases: *matplotlib.backend_tools.ToolBase*

Send message with the current pointer position.

This tool runs in the background reporting the position of the cursor.

send_message(*self*, *event*)

Call *matplotlib.backend_managers.ToolManager.message_event*.

set_figure(*self*, *figure*)

Assign a figure to the tool.

Parameters

figure

[*Figure*]

class matplotlib.backend_tools.**ToolEnableAllNavigation**(***kwargs*)

Bases: matplotlib.backend_tools._ToolEnableAllNavigation

[*Deprecated*]

Notes

Deprecated since version 3.3:

class matplotlib.backend_tools.**ToolEnableNavigation**(***kwargs*)

Bases: matplotlib.backend_tools._ToolEnableNavigation

[*Deprecated*]

Notes

Deprecated since version 3.3:

class matplotlib.backend_tools.**ToolForward**(*toolmanager*, *name*)

Bases: *matplotlib.backend_tools.ViewsPositionsBase*

Move forward in the view lim stack.

```
default_keymap = ['right', 'v', 'MouseButton.FORWARD']
```

```
description = 'Forward to next view'
```

```
image = 'forward'
```

class matplotlib.backend_tools.**ToolFullScreen**(**args*, ***kwargs*)

Bases: *matplotlib.backend_tools.ToolToggleBase*

Tool to toggle full screen.

```
default_keymap = ['f', 'ctrl+f']
```

```
description = 'Toggle fullscreen mode'
```

disable(*self*, *event*)

Disable the toggle tool.

`trigger` call this method when `toggled` is True.

This can happen in different circumstances.

- Click on the toolbar tool button.
- Call to *matplotlib.backend_managers.ToolManager.trigger_tool*.
- Another *ToolToggleBase* derived tool is triggered (from the same *ToolManager*).

enable(*self*, *event*)

Enable the toggle tool.

`trigger` calls this method when `toggled` is False.

class matplotlib.backend_tools.**ToolGrid**(*toolmanager*, *name*)

Bases: *matplotlib.backend_tools.ToolBase*

Tool to toggle the major grids of the figure.

```
default_keymap = ['g']
```

```
description = 'Toggle major grids'
```

trigger(*self*, *sender*, *event*, *data=None*)

Called when this tool gets used.

This method is called by *matplotlib.backend_managers.ToolManager.trigger_tool*.

Parameters

event

[*Event*] The canvas event that caused this tool to be called.

sender

[object] Object that requested the tool to be triggered.

data

[object] Extra data.

class matplotlib.backend_tools.ToolHelpBase(*toolmanager*, *name*)

Bases: *matplotlib.backend_tools.ToolBase*

default_keymap = ['f1']

description = 'Print tool list, shortcuts and description'

static format_shortcut(*key_sequence*)

Converts a shortcut string from the notation used in rc config to the standard notation for displaying shortcuts, e.g. 'ctrl+a' -> 'Ctrl+A'.

image = 'help'

class matplotlib.backend_tools.ToolHome(*toolmanager*, *name*)

Bases: *matplotlib.backend_tools.ViewsPositionsBase*

Restore the original view limits.

default_keymap = ['h', 'r', 'home']

description = 'Reset original view'

image = 'home'

class matplotlib.backend_tools.ToolMinorGrid(*toolmanager*, *name*)

Bases: *matplotlib.backend_tools.ToolBase*

Tool to toggle the major and minor grids of the figure.

default_keymap = ['G']

description = 'Toggle major and minor grids'

trigger(*self*, *sender*, *event*, *data=None*)

Called when this tool gets used.

This method is called by *matplotlib.backend_managers.ToolManager.trigger_tool*.

Parameters

event

[*Event*] The canvas event that caused this tool to be called.

sender

[object] Object that requested the tool to be triggered.

data

[object] Extra data.

class matplotlib.backend_tools.**ToolPan**(**args*)

Bases: *matplotlib.backend_tools.ZoomPanBase*

Pan axes with left mouse, zoom with right.

```
cursor = 3
```

```
default_keymap = ['p']
```

```
description = 'Pan axes with left mouse, zoom with right'
```

```
image = 'move'
```

```
radio_group = 'default'
```

class matplotlib.backend_tools.**ToolQuit**(*toolmanager*, *name*)

Bases: *matplotlib.backend_tools.ToolBase*

Tool to call the figure manager destroy method.

```
default_keymap = ['ctrl+w', 'cmd+w', 'q']
```

```
description = 'Quit the figure'
```

trigger(*self*, *sender*, *event*, *data=None*)

Called when this tool gets used.

This method is called by *matplotlib.backend_managers.ToolManager.trigger_tool*.

Parameters

event

[*Event*] The canvas event that caused this tool to be called.

sender

[object] Object that requested the tool to be triggered.

data

[object] Extra data.

class matplotlib.backend_tools.**ToolQuitAll**(*toolmanager*, *name*)

Bases: *matplotlib.backend_tools.ToolBase*

Tool to call the figure manager destroy method.

```
default_keymap = []
```

```
description = 'Quit all figures'
```

`trigger`(*self*, *sender*, *event*, *data=None*)
Called when this tool gets used.

This method is called by *matplotlib.backend_managers.ToolManager.trigger_tool*.

Parameters

event

[*Event*] The canvas event that caused this tool to be called.

sender

[object] Object that requested the tool to be triggered.

data

[object] Extra data.

`class matplotlib.backend_tools.ToolToggleBase`(**args*, ***kwargs*)
Bases: *matplotlib.backend_tools.ToolBase*

Toggleable tool.

Every time it is triggered, it switches between enable and disable.

Parameters

``*args``

Variable length argument to be used by the Tool.

``**kwargs``

toggled if present and True, sets the initial state of the Tool Arbitrary keyword arguments to be consumed by the Tool

`cursor = None`
Cursor to use when the tool is active.

`default_toggled = False`
Default of toggled state.

`disable`(*self*, *event=None*)
Disable the toggle tool.

trigger call this method when *toggled* is True.

This can happen in different circumstances.

- Click on the toolbar tool button.
- Call to *matplotlib.backend_managers.ToolManager.trigger_tool*.
- Another *ToolToggleBase* derived tool is triggered (from the same *ToolManager*).

`enable`(*self*, *event=None*)
Enable the toggle tool.

trigger calls this method when *toggled* is False.

`radio_group = None`
Attribute to group 'radio' like tools (mutually exclusive).

String that identifies the group or **None** if not belonging to a group.

`set_figure`(*self*, *figure*)
Assign a figure to the tool.

Parameters

figure

[*Figure*]

`property toggled`
State of the toggled tool.

`trigger`(*self*, *sender*, *event*, *data=None*)
Calls *enable* or *disable* based on *toggled* value.

`class matplotlib.backend_tools.ToolViewsPositions`(**args*, ***kwargs*)
Bases: *matplotlib.backend_tools.ToolBase*

Auxiliary Tool to handle changes in views and positions.

Runs in the background and should get used by all the tools that need to access the figure's history of views and positions, e.g.

- *ToolZoom*
- *ToolPan*
- *ToolHome*
- *ToolBack*
- *ToolForward*

`add_figure`(*self*, *figure*)
Add the current figure to the stack of views and positions.

`back`(*self*)
Back one step in the stack of views and positions.

`clear`(*self*, *figure*)
Reset the axes stack.

`forward`(*self*)
Forward one step in the stack of views and positions.

`home`(*self*)
Recall the first view and position from the stack.

`push_current`(*self*, *figure=None*)
Push the current view limits and position onto their respective stacks.

`refresh_locators`(*self*)
[*Deprecated*] Redraw the canvases, update the locators.

Notes

Deprecated since version 3.3.

`update_home_views`(*self*, *figure=None*)
Make sure that `self.home_views` has an entry for all axes present in the figure.

`update_view`(*self*)
Update the view limits and position for each axes from the current stack position. If any axes are present in the figure that aren't in the current stack position, use the home view limits for those axes and don't update *any* positions.

class `matplotlib.backend_tools.ToolXScale`(**args*, ***kwargs*)
Bases: *matplotlib.backend_tools.AxisScaleBase*

Tool to toggle between linear and logarithmic scales on the X axis.

`default_keymap = ['k', 'L']`

`description = 'Toggle scale X axis'`

`set_scale`(*self*, *ax*, *scale*)

class `matplotlib.backend_tools.ToolYScale`(**args*, ***kwargs*)
Bases: *matplotlib.backend_tools.AxisScaleBase*

Tool to toggle between linear and logarithmic scales on the Y axis.

`default_keymap = ['l']`

`description = 'Toggle scale Y axis'`

`set_scale`(*self*, *ax*, *scale*)

class `matplotlib.backend_tools.ToolZoom`(**args*)
Bases: *matplotlib.backend_tools.ZoomPanBase*

A Tool for zooming using a rectangle selector.

`cursor = 2`

`default_keymap = ['o']`

`description = 'Zoom to rectangle'`

`image = 'zoom_to_rect'`

`radio_group = 'default'`

class matplotlib.backend_tools.**ViewsPositionsBase**(*toolmanager*, *name*)

Bases: *matplotlib.backend_tools.ToolBase*

Base class for *ToolHome*, *ToolBack* and *ToolForward*.

trigger(*self*, *sender*, *event*, *data=None*)

Called when this tool gets used.

This method is called by *matplotlib.backend_managers.ToolManager.trigger_tool*.

Parameters

event

[*Event*] The canvas event that caused this tool to be called.

sender

[object] Object that requested the tool to be triggered.

data

[object] Extra data.

class matplotlib.backend_tools.**ZoomPanBase**(**args*)

Bases: *matplotlib.backend_tools.ToolToggleBase*

Base class for *ToolZoom* and *ToolPan*.

disable(*self*, *event*)

Release the canvas and disconnect press/release events.

enable(*self*, *event*)

Connect press/release events and lock the canvas.

scroll_zoom(*self*, *event*)

trigger(*self*, *sender*, *event*, *data=None*)

Calls *enable* or *disable* based on `toggled` value.

matplotlib.backend_tools.**add_tools_to_container**(*container*, *tools=[['navigation', ['home', 'back', 'forward']], ['zoompan', ['pan', 'zoom', 'subplots']], ['io', ['save', 'help']]]*)

Add multiple tools to the container.

Parameters

container

[Container] `backend_bases.ToolContainerBase` object that will get the tools added.

tools

[list, optional] List in the form `[[group1, [tool1, tool2 ...]], [group2, [...]]]` where the tools `[tool1, tool2, ...]` will display in group1. See `add_tool` for details.

matplotlib.backend_tools.**add_tools_to_manager**(*toolmanager, tools={'home': <class 'matplotlib.backend_tools.ToolHome'>, 'back': <class 'matplotlib.backend_tools.ToolBack'>, 'forward': <class 'matplotlib.backend_tools.ToolForward'>, 'zoom': <class 'matplotlib.backend_tools.ToolZoom'>, 'pan': <class 'matplotlib.backend_tools.ToolPan'>, 'subplots': 'ToolConfigureSubplots', 'save': 'ToolSaveFigure', 'grid': <class 'matplotlib.backend_tools.ToolGrid'>, 'grid_minor': <class 'matplotlib.backend_tools.ToolMinorGrid'>, 'fullscreen': <class 'matplotlib.backend_tools.ToolFullScreen'>, 'quit': <class 'matplotlib.backend_tools.ToolQuit'>, 'quit_all': <class 'matplotlib.backend_tools.ToolQuitAll'>, 'allnav': <class 'matplotlib.backend_tools._ToolEnableAllNavigation'>, 'nav': <class 'matplotlib.backend_tools._ToolEnableNavigation'>, 'xscale': <class 'matplotlib.backend_tools.ToolXScale'>, 'yscale': <class 'matplotlib.backend_tools.ToolYScale'>, 'position': <class 'matplotlib.backend_tools.ToolCursorPosition'>, 'viewpos': <class 'matplotlib.backend_tools.ToolViewsPositions'>, 'cursor': 'ToolSetCursor', 'rubberband': 'ToolRubberband', 'help': 'ToolHelp', 'copy': 'ToolCopyToClipboard'}*)

Add multiple tools to a *ToolManager*.

Parameters

toolmanager

[*backend_managers.ToolManager*] Manager to which the tools are added.

tools

[{str: class_like}, optional] The tools to add in a {name: tool} dict, see `add_tool` for more info.

matplotlib.backend_tools.**cursors**
alias of *matplotlib.backend_tools.Cursors*

matplotlib.backend_tools.**default_toolbar_tools** = [['navigation', ['home', 'back', 'forward']],
Default tools in the toolbar

matplotlib.backend_tools.**default_tools** = {'allnav': <class 'matplotlib.backend_tools._ToolEnab
Default tools

18.10 matplotlib.backends

18.10.1 matplotlib.backends.backend_mixed

class matplotlib.backends.backend_mixed.**MixedModeRenderer**(*figure*, *width*, *height*, *dpi*, *vector_renderer*, *raster_renderer_class=None*, *bbox_inches_restore=None*)

Bases: `object`

A helper class to implement a renderer that switches between vector and raster drawing. An example may be a PDF writer, where most things are drawn with PDF vector commands, but some very complex objects, such as quad meshes, are rasterised and then output as images.

Parameters

figure

[*matplotlib.figure.Figure*] The figure instance.

width

[scalar] The width of the canvas in logical units

height

[scalar] The height of the canvas in logical units

dpi

[float] The dpi of the canvas

vector_renderer

[*matplotlib.backend_bases.RendererBase*] An instance of a subclass of *RendererBase* that will be used for the vector drawing.

raster_renderer_class

[*matplotlib.backend_bases.RendererBase*] The renderer class to use for the raster drawing. If not provided, this will use the Agg backend (which is currently the only viable option anyway.)

`start_rasterizing`(*self*)
Enter "raster" mode. All subsequent drawing commands (until *stop_rasterizing* is called) will be drawn with the raster backend.

`stop_rasterizing`(*self*)
Exit "raster" mode. All of the drawing that was done since the last *start_rasterizing* call will be copied to the vector backend by calling draw_image.

If *start_rasterizing* has been called multiple times, *stop_rasterizing* must be called the same number of times before "raster" mode is exited.

18.10.2 matplotlib.backends.backend_template

A fully functional, do-nothing backend intended as a template for backend writers. It is fully functional in that you can select it as a backend e.g. with

```
import matplotlib
matplotlib.use("template")
```

and your program will (should!) run without error, though no output is produced. This provides a starting point for backend writers; you can selectively implement drawing methods (*draw_path*, *draw_image*, etc.) and slowly see your figure come to life instead having to have a full blown implementation before getting any results.

Copy this file to a directory outside of the Matplotlib source tree, somewhere where Python can import it (by adding the directory to your `sys.path` or by packaging it as a normal Python package); if the backend is importable as `import my.backend` you can then select it using

```
import matplotlib
matplotlib.use("module://my.backend")
```

If your backend implements support for saving figures (i.e. has a `print_xyz` method), you can register it as the default handler for a given file type:

```
from matplotlib.backend_bases import register_backend
register_backend('xyz', 'my_backend', 'XYZ File Format')
...
plt.savefig("figure.xyz")
```

matplotlib.backends.backend_template.**FigureCanvas**
alias of *matplotlib.backends.backend_template.FigureCanvasTemplate*

class matplotlib.backends.backend_template.**FigureCanvasTemplate**(*figure*)
Bases: *matplotlib.backend_bases.FigureCanvasBase*

The canvas the figure renders into. Calls the draw and print fig methods, creates the renderers, etc.

Note: GUI templates will want to connect events for button presses, mouse movements and key presses to functions that call the base class methods button_press_event, button_release_event, motion_notify_event, key_press_event, and key_release_event. See the implementations of the interactive backends for examples.

Attributes

figure

[*matplotlib.figure.Figure*] A high-level Figure instance

draw(*self*)
Draw the figure using the renderer.

filetypes = {'eps': 'Encapsulated Postscript', 'foo': 'My magic Foo format', 'jpeg': 'Joi

get_default_filetype(*self*)
Return the default savefig file format as specified in `rcParams["savefig.format"]` (default: `'png'`).

The returned string does not include a period. This method is overridden in backends that only support a single file type.

print_foo(*self*, *filename*, **args*, ***kwargs*)
Write out format foo. The dpi, facecolor and edgecolor are restored to their original values after this call, so you don't need to save and restore them.

matplotlib.backends.backend_template.**FigureManager**
alias of *matplotlib.backends.backend_template.FigureManagerTemplate*

class matplotlib.backends.backend_template.**FigureManagerTemplate**(*canvas*, *num*)
Bases: *matplotlib.backend_bases.FigureManagerBase*

Helper class for pyplot mode, wraps everything up into a neat bundle.

For non-interactive backends, the base class is sufficient.

class matplotlib.backends.backend_template.**GraphicsContextTemplate**
Bases: *matplotlib.backend_bases.GraphicsContextBase*

The graphics context provides the color, line styles, etc... See the cairo and postscript backends for examples of mapping the graphics context attributes (cap styles, join styles, line widths, colors) to a particular backend. In cairo this is done by wrapping a cairo.Context object and forwarding the appropriate calls

to it using a dictionary mapping styles to gdk constants. In Postscript, all the work is done by the renderer, mapping line styles to postscript calls.

If it's more appropriate to do the mapping at the renderer level (as in the postscript backend), you don't need to override any of the GC methods. If it's more appropriate to wrap an instance (as in the cairo backend) and do the mapping here, you'll need to override several of the setter methods.

The base GraphicsContext stores colors as a RGB tuple on the unit interval, e.g., (0.5, 0.0, 1.0). You may need to map this to colors appropriate for your backend.

class matplotlib.backends.backend_template.**RendererTemplate**(*dpi*)

Bases: *matplotlib.backend_bases.RendererBase*

The renderer handles drawing/rendering operations.

This is a minimal do-nothing class that can be used to get started when writing a new backend. Refer to `backend_bases.RendererBase` for documentation of the methods.

draw_image(*self*, *gc*, *x*, *y*, *im*)

Draw an RGBA image.

Parameters

gc

[*GraphicsContextBase*] A graphics context with clipping information.

x

[scalar] The distance in physical units (i.e., dots or pixels) from the left hand side of the canvas.

y

[scalar] The distance in physical units (i.e., dots or pixels) from the bottom side of the canvas.

im

[array-like, shape=(N, M, 4), dtype=np.uint8] An array of RGBA pixels.

transform

[*matplotlib.transforms.Affine2DBase*] If and only if the concrete backend is written such that `option_scale_image()` returns `True`, an affine transformation (i.e., an *Affine2DBase*) *may* be passed to *draw_image()*. The translation vector of the transformation is given in physical units (i.e., dots or pixels). Note that the transformation does not override *x* and *y*, and has to be applied *before* translating the result by *x* and *y* (this can be accomplished by adding *x* and *y* to the translation vector defined by *transform*).

`draw_path`(*self*, *gc*, *path*, *transform*, *rgbFace=None*)
Draw a *Path* instance using the given affine transform.

`draw_text`(*self*, *gc*, *x*, *y*, *s*, *prop*, *angle*, *ismath=False*, *mtext=None*)
Draw the text instance.

Parameters

gc

[*GraphicsContextBase*] The graphics context.

x

[float] The x location of the text in display coords.

y

[float] The y location of the text baseline in display coords.

s

[str] The text string.

prop

[*matplotlib.font_manager.FontProperties*] The font properties.

angle

[float] The rotation angle in degrees anti-clockwise.

mtext

[*matplotlib.text.Text*] The original text object to be rendered.

Notes

Note for backend implementers:

When you are trying to determine if you have gotten your bounding box right (which is what enables the text layout/alignment to work properly), it helps to change the line in text.py:

```
if 0: bbox_artist(self, renderer)
```

to if 1, and then the actual bounding box will be plotted along with your text.

`flipy`(*self*)
Return whether y values increase from top to bottom.

Note that this only affects drawing of texts and images.

`get_canvas_width_height`(*self*)
Return the canvas width and height in display coords.

get_text_width_height_descent(*self*, *s*, *prop*, *ismath*)
Get the width, height, and descent (offset from the bottom to the baseline), in display coords, of the string *s* with *FontProperties* *prop*.

new_gc(*self*)
Return an instance of a *GraphicsContextBase*.

points_to_pixels(*self*, *points*)
Convert points to display units.

You need to override this function (unless your backend doesn't have a dpi, e.g., postscript or svg). Some imaging systems assume some value for pixels per inch:

```
points to pixels = points * pixels_per_inch/72 * dpi/72
```

Parameters

points

[float or array-like] a float or a numpy array of float

Returns

Points converted to pixels

matplotlib.backends.backend_template.**draw_if_interactive**()
For image backends - is not required. For GUI backends - this should be overridden if drawing should be done in interactive python mode.

matplotlib.backends.backend_template.**new_figure_manager**(*num*, **args*, *FigureClass=<class 'matplotlib.figure.Figure'>*, ***kwargs*)
Create a new figure manager instance.

matplotlib.backends.backend_template.**new_figure_manager_given_figure**(*num*, *figure*)
Create a new figure manager instance for the given figure.

matplotlib.backends.backend_template.**show**(*, *block=None*)
For image backends - is not required. For GUI backends - show() is usually the last line of a pyplot script and tells the backend that it is time to draw. In interactive mode, this should do nothing.

18.10.3 matplotlib.backends.backend_agg

An agg backend.

Features that are implemented:

- capstyles and join styles
- dashes
- linewidth
- lines, rectangles, ellipses
- clipping to a rectangle
- output to RGBA and Pillow-supported image formats
- alpha blending
- DPI scaling properly - everything scales properly (dashes, linewidths, etc)
- draw polygon
- freetype2 w/ ft2font

TODO:

- integrate screen dpi w/ ppi and text

matplotlib.backends.backend_agg.**FigureCanvas**
alias of *matplotlib.backends.backend_agg.FigureCanvasAgg*

class matplotlib.backends.backend_agg.**FigureCanvasAgg**(*figure*)
Bases: *matplotlib.backend_bases.FigureCanvasBase*

buffer_rgba(*self*)
Get the image as a memoryview to the renderer's buffer.

draw must be called at least once before this function will work and to update the renderer for any subsequent changes to the Figure.

copy_from_bbox(*self*, *bbox*)

draw(*self*)
Render the *Figure*.

get_renderer(*self*, *cleared=False*)

print_jpeg(*self*, *filename_or_obj*, **args*, *dryrun=<deprecated parameter>*, *pil_kwargs=None*, ***kwargs*)
Write the figure to a JPEG file.

Parameters

filename_or_obj

[str or path-like or file-like] The file to write to.

Other Parameters

quality

[int, default: `rcParams["savefig.jpeg_quality"]` (default: `95`)] The image quality, on a scale from 1 (worst) to 95 (best). Values above 95 should be avoided; 100 disables portions of the JPEG compression algorithm, and results in large files with hardly any gain in image quality. This parameter is deprecated.

optimize

[bool, default: False] Whether the encoder should make an extra pass over the image in order to select optimal encoder settings. This parameter is deprecated.

progressive

[bool, default: False] Whether the image should be stored as a progressive JPEG file. This parameter is deprecated.

pil_kwargs

[dict, optional] Additional keyword arguments that are passed to `PIL.Image.Image.save` when saving the figure. These take precedence over *quality*, *optimize* and *progressive*.

`print_jpg`(*self*, *filename_or_obj*, **args*, *dryrun=<deprecated parameter>*, *pil_kwargs=None*, ***kwargs*)

Write the figure to a JPEG file.

Parameters

filename_or_obj

[str or path-like or file-like] The file to write to.

Other Parameters

quality

[int, default: `rcParams["savefig.jpeg_quality"]` (default: `95`)] The image quality, on a scale from 1 (worst) to 95 (best). Values above 95 should be avoided; 100 disables portions of the JPEG compression algorithm, and results in large files with hardly any gain in image quality. This parameter is deprecated.

optimize

[bool, default: False] Whether the encoder should make an extra pass over the image in order to select optimal encoder settings. This parameter is deprecated.

progressive

[bool, default: False] Whether the image should be stored as a progressive JPEG file. This parameter is deprecated.

pil_kwargs

[dict, optional] Additional keyword arguments that are passed to `PIL.Image.Image.save` when saving the figure. These take precedence over *quality*, *optimize* and *progressive*.

`print_png`(*self*, *filename_or_obj*, **args*, *metadata=None*, *pil_kwargs=None*)

Write the figure to a PNG file.

Parameters

filename_or_obj

[str or path-like or file-like] The file to write to.

metadata

[dict, optional] Metadata in the PNG file as key-value pairs of bytes or latin-1 encodable strings. According to the PNG specification, keys must be shorter than 79 chars.

The PNG specification defines some common keywords that may be used as appropriate:

- Title: Short (one line) title or caption for image.
- Author: Name of image's creator.
- Description: Description of image (possibly long).
- Copyright: Copyright notice.
- Creation Time: Time of original image creation (usually RFC 1123 format).
- Software: Software used to create the image.
- Disclaimer: Legal disclaimer.
- Warning: Warning of nature of content.
- Source: Device used to create the image.
- Comment: Miscellaneous comment; conversion from other image format.

Other keywords may be invented for other purposes.

If 'Software' is not given, an autogenerated value for Matplotlib will be used. This can be removed by setting it to *None*.

For more details see the PNG specification.

pil_kwargs

[dict, optional] Keyword arguments passed to `PIL.Image.Image.save`.

If the 'pnginfo' key is present, it completely overrides *metadata*, including the default 'Software' key.

`print_raw`(*self*, *filename_or_obj*, **args*)

`print_rgba`(*self*, *filename_or_obj*, **args*)

`print_tif`(*self*, *filename_or_obj*, ***, *dryrun=<deprecated parameter>*, *pil_kwargs=None*)

`print_tiff`(*self*, *filename_or_obj*, ***, *dryrun=<deprecated parameter>*, *pil_kwargs=None*)

`print_to_buffer`(*self*)

`restore_region`(*self*, *region*, *bbox=None*, *xy=None*)

`tostring_argb`(*self*)
Get the image as ARGB `bytes`.

draw must be called at least once before this function will work and to update the renderer for any subsequent changes to the Figure.

`tostring_rgb`(*self*)
Get the image as RGB `bytes`.

draw must be called at least once before this function will work and to update the renderer for any subsequent changes to the Figure.

class matplotlib.backends.backend_agg.**RendererAgg**(*width*, *height*, *dpi*)
Bases: *matplotlib.backend_bases.RendererBase*

The renderer handles all the drawing primitives using a graphics context instance that controls the colors/styles

`buffer_rgba`(*self*)

`clear`(*self*)

`draw_mathtext`(*self*, *gc*, *x*, *y*, *s*, *prop*, *angle*)
Draw mathtext using *matplotlib.mathtext*.

`draw_path`(*self*, *gc*, *path*, *transform*, *rgbFace=None*)
Draw a *Path* instance using the given affine transform.

`draw_path_collection`(*self*, *gc*, *master_transform*, *paths*, *all_transforms*, *offsets*, *offsetTrans*, *facecolors*, *edgecolors*, *linewidths*, *linestyles*, *antialiaseds*, *urls*, *offset_position*)
Draw a collection of paths selecting drawing properties from the lists *facecolors*, *edgecolors*, *linewidths*, *linestyles* and *antialiaseds*. *offsets* is a list of offsets to apply to each of the paths. The offsets in *offsets* are first transformed by *offsetTrans* before being applied.

offset_position may be either "screen" or "data" depending on the space that the offsets are in; "data" is deprecated.

This provides a fallback implementation of *draw_path_collection()* that makes multiple calls to *draw_path()*. Some backends may want to override this in order to render each set of path data only once, and then reference that path multiple times with the different offsets, colors, styles etc. The generator methods `_iter_collection_raw_paths()` and `_iter_collection()` are provided to help with (and standardize) the implementation across backends. It is highly recommended to use those generators, so that changes to the behavior of *draw_path_collection()* can be made globally.

`draw_tex`(*self*, *gc*, *x*, *y*, *s*, *prop*, *angle*, *ismath=<deprecated parameter>*, *mtext=None*)

`draw_text`(*self*, *gc*, *x*, *y*, *s*, *prop*, *angle*, *ismath=False*, *mtext=None*)

Draw the text instance.

Parameters

gc

[*GraphicsContextBase*] The graphics context.

x

[float] The x location of the text in display coords.

y

[float] The y location of the text baseline in display coords.

s

[str] The text string.

prop

[*matplotlib.font_manager.FontProperties*] The font properties.

angle

[float] The rotation angle in degrees anti-clockwise.

mtext

[*matplotlib.text.Text*] The original text object to be rendered.

Notes

Note for backend implementers:

When you are trying to determine if you have gotten your bounding box right (which is what enables the text layout/alignment to work properly), it helps to change the line in text.py:

```
if 0: bbox_artist(self, renderer)
```

to if 1, and then the actual bounding box will be plotted along with your text.

`get_canvas_width_height`(*self*)
Return the canvas width and height in display coords.

`get_text_width_height_descent`(*self*, *s*, *prop*, *ismath*)
Get the width, height, and descent (offset from the bottom to the baseline), in display coords, of the string *s* with *FontProperties* *prop*.

`lock = <unlocked _thread.RLock object owner=0 count=0>`

`option_image_nocomposite`(*self*)
Return whether image composition by Matplotlib should be skipped.

Raster backends should usually return False (letting the C-level rasterizer take care of image composition); vector backends should usually return `not rcParams["image.composite_image"]`.

`option_scale_image`(*self*)
Return whether arbitrary affine transformations in `draw_image()` are supported (True for most vector backends).

`points_to_pixels`(*self*, *points*)
Convert points to display units.

You need to override this function (unless your backend doesn't have a dpi, e.g., postscript or svg). Some imaging systems assume some value for pixels per inch:

```
points to pixels = points * pixels_per_inch/72 * dpi/72
```

Parameters

points

[float or array-like] a float or a numpy array of float

Returns

Points converted to pixels

restore_region(*self*, *region*, *bbox=None*, *xy=None*)
Restore the saved region. If bbox (instance of BboxBase, or its extents) is given, only the region specified by the bbox will be restored. *xy* (a pair of floats) optionally specifies the new position (the LLC of the original region, not the LLC of the bbox) where the region will be restored.

```
>>> region = renderer.copy_from_bbox()
>>> x1, y1, x2, y2 = region.get_extents()
>>> renderer.restore_region(region, bbox=(x1+dx, y1, x2, y2),
...                         xy=(x1-dx, y1))
```

start_filter(*self*)
Start filtering. It simply create a new canvas (the old one is saved).

stop_filter(*self*, *post_processing*)
Save the plot in the current canvas as a image and apply the *post_processing* function.

def post_processing(image, dpi):

ny, nx, depth = image.shape # image (numpy array) has RGBA channels and has a depth of 4. ... # create a new_image (numpy array of 4 channels, size can be # different). The resulting image may have offsets from # lower-left corner of the original image return new_image, offset_x, offset_y

The saved renderer is restored and the returned image from post_processing is plotted (using draw_image) on it.

tostring_argb(*self*)

tostring_rgb(*self*)

tostring_rgba_minimized(*self*)

matplotlib.backends.backend_agg.get_hinting_flag()

18.10.4 matplotlib.backends.backend_cairo

A Cairo backend for matplotlib

Author

Steve Chaplin and others

This backend depends on cairocffi or pycairo.

matplotlib.backends.backend_cairo.FigureCanvas
alias of *matplotlib.backends.backend_cairo.FigureCanvasCairo*

class matplotlib.backends.backend_cairo.FigureCanvasCairo(*figure*)
Bases: *matplotlib.backend_bases.FigureCanvasBase*

copy_from_bbox(*self*, *bbox*)

print_pdf(*self*, *fobj*, **args*, ***kwargs*)

print_png(*self*, *fobj*)

print_ps(*self*, *fobj*, **args*, ***kwargs*)

print_raw(*self*, *fobj*)

print_rgba(*self*, *fobj*)

print_svg(*self*, *fobj*, **args*, ***kwargs*)

print_svgz(*self*, *fobj*, **args*, ***kwargs*)

restore_region(*self*, *region*)

class matplotlib.backends.backend_cairo.GraphicsContextCairo(*renderer*)

Bases: *matplotlib.backend_bases.GraphicsContextBase*

get_rgb(*self*)

Return a tuple of three or four floats from 0-1.

restore(*self*)

Restore the graphics context from the stack - needed only for backends that save graphics contexts on a stack.

set_alpha(*self*, *alpha*)

Set the alpha value used for blending - not supported on all backends.

If `alpha=None` (the default), the alpha components of the foreground and fill colors will be used to set their respective transparencies (where applicable); otherwise, `alpha` will override them.

set_capstyle(*self*, *cs*)

Set the capstyle to be one of ('butt', 'round', 'projecting').

set_clip_path(*self*, *path*)

Set the clip path and transformation.

Parameters

path

[*TransformedPath* or None]

set_clip_rectangle(*self*, *rectangle*)

Set the clip rectangle with sequence (left, bottom, width, height)

set_dashes(*self*, *offset*, *dashes*)

Set the dash style for the gc.

Parameters

dash_offset

[float or None] The offset (usually 0).

dash_list

[array-like or None] The on-off sequence as points.

Notes

`(None, None)` specifies a solid line.

See p. 107 of to PostScript blue book for more info.

`set_foreground`(*self*, *fg*, *isRGBA=None*)
Set the foreground color.

Parameters

fg

[color]

isRGBA

[bool] If *fg* is known to be an `(r, g, b, a)` tuple, *isRGBA* can be set to True to improve performance.

`set_joinstyle`(*self*, *js*)
Set the join style to be one of ('miter', 'round', 'bevel').

`set_linewidth`(*self*, *w*)
Set the linewidth in points.

`class matplotlib.backends.backend_cairo.RendererCairo`(*dpi*)
Bases: *matplotlib.backend_bases.RendererBase*

`draw_image`(*self*, *gc*, *x*, *y*, *im*)
Draw an RGBA image.

Parameters

gc

[*GraphicsContextBase*] A graphics context with clipping information.

x

[scalar] The distance in physical units (i.e., dots or pixels) from the left hand side of the canvas.

y

[scalar] The distance in physical units (i.e., dots or pixels) from the bottom side of the canvas.

im

[array-like, shape=(N, M, 4), dtype=np.uint8] An array of RGBA pixels.

transform

[*matplotlib.transforms.Affine2DBase*] If and only if the concrete backend is written such that `option_scale_image()` returns `True`, an affine transformation (i.e., an *Affine2DBase*) *may* be passed to *draw_image()*. The translation vector of the transformation is given in physical units (i.e., dots or pixels). Note that the transformation does not override *x* and *y*, and has to be applied *before* translating the result by *x* and *y* (this can be accomplished by adding *x* and *y* to the translation vector defined by *transform*).

`draw_markers`(*self*, *gc*, *marker_path*, *marker_trans*, *path*, *transform*, *rgbFace=None*)

Draw a marker at each of the vertices in path.

This includes all vertices, including control points on curves. To avoid that behavior, those vertices should be removed before calling this function.

This provides a fallback implementation of draw_markers that makes multiple calls to *draw_path()*. Some backends may want to override this method in order to draw the marker only once and reuse it multiple times.

Parameters

gc

[*GraphicsContextBase*] The graphics context.

marker_trans

[*matplotlib.transforms.Transform*] An affine transform applied to the marker.

trans

[*matplotlib.transforms.Transform*] An affine transform applied to the path.

`draw_path`(*self*, *gc*, *path*, *transform*, *rgbFace=None*)

Draw a *Path* instance using the given affine transform.

`draw_text`(*self*, *gc*, *x*, *y*, *s*, *prop*, *angle*, *ismath=False*, *mtext=None*)

Draw the text instance.

Parameters

gc

[*GraphicsContextBase*] The graphics context.

x

[float] The x location of the text in display coords.

y

[float] The y location of the text baseline in display coords.

s

[str] The text string.

prop

[*matplotlib.font_manager.FontProperties*] The font properties.

angle

[float] The rotation angle in degrees anti-clockwise.

mtext

[*matplotlib.text.Text*] The original text object to be rendered.

Notes

Note for backend implementers:

When you are trying to determine if you have gotten your bounding box right (which is what enables the text layout/alignment to work properly), it helps to change the line in text.py:

```
if 0: bbox_artist(self, renderer)
```

to if 1, and then the actual bounding box will be plotted along with your text.

`property fontangles`

`property fontweights`

`get_canvas_width_height`(*self*)
Return the canvas width and height in display coords.

`get_text_width_height_descent`(*self*, *s*, *prop*, *ismath*)
Get the width, height, and descent (offset from the bottom to the baseline), in display coords, of the string *s* with *FontProperties* *prop*.

`new_gc`(*self*)
Return an instance of a *GraphicsContextBase*.

`points_to_pixels`(*self*, *points*)
Convert points to display units.

You need to override this function (unless your backend doesn't have a dpi, e.g., postscript or svg). Some imaging systems assume some value for pixels per inch:

```
points to pixels = points * pixels_per_inch/72 * dpi/72
```

Parameters

points

[float or array-like] a float or a numpy array of float

Returns

Points converted to pixels

`set_ctx_from_surface`(*self*, *surface*)

`set_width_height`(*self*, *width*, *height*)

18.10.5 matplotlib.backends.backend_gtk3agg

TODO We'll add this later, importing the gtk3 backends requires an active X-session, which is not compatible with cron jobs.

18.10.6 matplotlib.backends.backend_gtk3cairo

TODO We'll add this later, importing the gtk3 backends requires an active X-session, which is not compatible with cron jobs.

18.10.7 matplotlib.backends.backend_nbagg

Interactive figures in the IPython notebook

`class matplotlib.backends.backend_nbagg.CommSocket`(*manager*)

Bases: `object`

Manages the Comm connection between IPython and the browser (client).

Comms are 2 way, with the CommSocket being able to publish a message via the send_json method, and handle a message with on_message. On the JS side figure.send_message and figure.ws.onmessage do the sending and receiving respectively.

`is_open`(*self*)

`on_close`(*self*)

`on_message`(*self*, *message*)

`send_binary`(*self*, *blob*)

`send_json`(*self*, *content*)

matplotlib.backends.backend_nbagg.**FigureCanvas**
alias of *matplotlib.backends.backend_nbagg.FigureCanvasNbAgg*

class matplotlib.backends.backend_nbagg.**FigureCanvasNbAgg**(**args*, ***kwargs*)
Bases: matplotlib.backends.backend_webagg_core.FigureCanvasWebAggCore

matplotlib.backends.backend_nbagg.**FigureManager**
alias of *matplotlib.backends.backend_nbagg.FigureManagerNbAgg*

class matplotlib.backends.backend_nbagg.**FigureManagerNbAgg**(*canvas*, *num*)
Bases: matplotlib.backends.backend_webagg_core.FigureManagerWebAgg

ToolbarCls
alias of *NavigationIPy*

clearup_closed(*self*)
Clear up any closed Comms.

property connected

destroy(*self*)

display_js(*self*)

classmethod get_javascript(*stream=None*)

remove_comm(*self*, *comm_id*)

reshow(*self*)
A special method to re-show the figure in the notebook.

show(*self*)
For GUI backends, show the figure window and redraw. For non-GUI backends, raise an exception, unless running headless (i.e. on Linux with an unset DISPLAY); this exception is converted to a warning in *Figure.show*.

class matplotlib.backends.backend_nbagg.**NavigationIPy**(*canvas*)
Bases: matplotlib.backends.backend_webagg_core.NavigationToolbar2WebAgg

toolitems = [('Home', 'Reset original view', 'fa fa-home icon-home', 'home'), ('Back', 'Ba

matplotlib.backends.backend_nbagg.**connection_info**()
Return a string showing the figure and connection status for the backend.

This is intended as a diagnostic tool, and not for general use.

matplotlib.backends.backend_nbagg.**new_figure_manager_given_figure**(*num*, *figure*)
Create a new figure manager instance for the given figure.

matplotlib.backends.backend_nbagg.**show**(*block=None*)
Show all figures.

show blocks by calling `mainloop` if *block* is `True`, or if it is `None` and we are neither in IPython's `%pylab` mode, nor in `interactive` mode.

18.10.8 matplotlib.backends.backend_pdf

A PDF matplotlib backend Author: Jouni K Seppänen <jks@iki.fi>

matplotlib.backends.backend_pdf.**FigureCanvas**
alias of *matplotlib.backends.backend_pdf.FigureCanvasPdf*

class matplotlib.backends.backend_pdf.**FigureCanvasPdf**(*figure*)
Bases: *matplotlib.backend_bases.FigureCanvasBase*

The canvas the figure renders into. Calls the draw and print fig methods, creates the renderers, etc...

Attributes

figure

[*matplotlib.figure.Figure*] A high-level Figure instance

filetypes = {'pdf': 'Portable Document Format'}

fixed_dpi = 72

get_default_filetype(*self*)
Return the default savefig file format as specified in `rcParams["savefig.format"]` (default: `'png'`).

The returned string does not include a period. This method is overridden in backends that only support a single file type.

print_pdf(*self*, *filename*, ***, *dpi=72*, *bbox_inches_restore=None*, *metadata=None*)

class matplotlib.backends.backend_pdf.**GraphicsContextPdf**(*file*)
Bases: *matplotlib.backend_bases.GraphicsContextBase*

alpha_cmd(*self*, *alpha*, *forced*, *effective_alphas*)

capstyle_cmd(*self*, *style*)

capstyles = {'butt': 0, 'projecting': 2, 'round': 1}

clip_cmd(*self*, *cliprect*, *clippath*)
Set clip rectangle. Calls *pop()* and *push()*.

commands = ((('_cliprect', '_clippath'), <function GraphicsContextPdf.clip_cmd>), (('_alph

copy_properties(*self*, *other*)
Copy properties of other into self.

dash_cmd(*self*, *dashes*)

delta(*self*, *other*)
Copy properties of other into self and return PDF commands needed to transform self into other.

fill(*self*, **args*)
Predicate: does the path need to be filled?

An optional argument can be used to specify an alternative _fillcolor, as needed by RendererPdf.draw_markers.

fillcolor_cmd(*self*, *rgb*)

finalize(*self*)
Make sure every pushed graphics state is popped.

hatch_cmd(*self*, *hatch*, *hatch_color*)

joinstyle_cmd(*self*, *style*)

joinstyles = {'bevel': 2, 'miter': 0, 'round': 1}

linewidth_cmd(*self*, *width*)

paint(*self*)
Return the appropriate pdf operator to cause the path to be stroked, filled, or both.

pop(*self*)

push(*self*)

rgb_cmd(*self*, *rgb*)

stroke(*self*)
Predicate: does the path need to be stroked (its outline drawn)? This tests for the various conditions that disable stroking the path, in which case it would presumably be filled.

class matplotlib.backends.backend_pdf.Name(*name*)
Bases: object

PDF name object.

static hexify(*match*)

name

pdfRepr(*self*)

class matplotlib.backends.backend_pdf.Operator(*op*)
Bases: object

PDF operator object.

op

pdfRepr(*self*)

class matplotlib.backends.backend_pdf.PdfFile(*filename*, *metadata=None*)
Bases: object

PDF file object.

Parameters

filename

[str or path-like or file-like] Output target; if a string, a file will be opened for writing.

metadata

[dict from strings to strings and dates] Information dictionary object (see PDF reference section 10.2.1 'Document Information Dictionary'), e.g.: `{'Creator': 'My software', 'Author': 'Me', 'Title': 'Awesome'}`.

The standard keys are 'Title', 'Author', 'Subject', 'Keywords', 'Creator', 'Producer', 'CreationDate', 'ModDate', and 'Trapped'. Values have been predefined for 'Creator', 'Producer' and 'CreationDate'. They can be removed by setting them to `None`.

`addGouraudTriangles`(*self*, *points*, *colors*)

Add a Gouraud triangle shading.

Parameters

points

[np.ndarray] Triangle vertices, shape (n, 3, 2) where n = number of triangles, 3 = vertices, 2 = x, y.

colors

[np.ndarray] Vertex colors, shape (n, 3, 1) or (n, 3, 4) as with points, but last dimension is either (gray,) or (r, g, b, alpha).

Returns

Name, Reference

`alphaState`(*self*, *alpha*)

Return name of an ExtGState that sets alpha to the given value.

`beginStream`(*self*, *id*, *len*, *extra=None*, *png=None*)

`close`(*self*)

Flush all buffers and free all resources.

`createType1Descriptor`(*self*, *t1font*, *fontfile*)

`dviFontName`(*self*, *dvifont*)

Given a dvi font object, return a name suitable for Op.selectfont. This registers the font information in `self.dviFontInfo` if not yet registered.

`embedTTF`(*self*, *filename*, *characters*)

Embed the TTF font from the named file into the document.

`endStream`(*self*)

finalize(*self*)
Write out the various deferred objects and the pdf end matter.

fontName(*self*, *fontprop*)
Select a font based on fontprop and return a name suitable for Op.selectfont. If fontprop is a string, it will be interpreted as the filename of the font.

hatchPattern(*self*, *hatch_style*)

imageObject(*self*, *image*)
Return name of an image XObject representing the given image.

markerObject(*self*, *path*, *trans*, *fill*, *stroke*, *lw*, *joinstyle*, *capstyle*)
Return name of a marker XObject representing the given path.

newPage(*self*, *width*, *height*)

newTextnote(*self*, *text*, *positionRect=[- 100, - 100, 0, 0]*)

output(*self*, **data*)

pathCollectionObject(*self*, *gc*, *path*, *trans*, *padding*, *filled*, *stroked*)

static pathOperations(*path*, *transform*, *clip=None*, *simplify=None*, *sketch=None*)

recordXref(*self*, *id*)

reserveObject(*self*, *name=''*)
Reserve an ID for an indirect object.

The name is used for debugging in case we forget to print out the object with writeObject.

property used_characters

write(*self*, *data*)

writeExtGSTates(*self*)

writeFonts(*self*)

writeGouraudTriangles(*self*)

writeHatches(*self*)

writeImages(*self*)

writeInfoDict(*self*)
Write out the info dictionary, checking it for good form

writeMarkers(*self*)

writeObject(*self*, *object*, *contents*)

writePath(*self*, *path*, *transform*, *clip=False*, *sketch=None*)

writePathCollectionTemplates(*self*)

`writeTrailer`(*self*)
Write out the PDF trailer.

`writeXref`(*self*)
Write out the xref table.

class matplotlib.backends.backend_pdf.`PdfPages`(*filename*, *keep_empty=True*, *metadata=None*)

Bases: `object`

A multi-page PDF file.

Notes

In reality *PdfPages* is a thin wrapper around *PdfFile*, in order to avoid confusion when using *savefig* and forgetting the format argument.

Examples

```
>>> import matplotlib.pyplot as plt
>>> # Initialize:
>>> with PdfPages('foo.pdf') as pdf:
...     # As many times as you like, create a figure fig and save it:
...     fig = plt.figure()
...     pdf.savefig(fig)
...     # When no figure is specified the current figure is saved
...     pdf.savefig()
```

Create a new PdfPages object.

Parameters

filename

[str or path-like or file-like] Plots using *PdfPages.savefig* will be written to a file at this location. The file is opened at once and any older file with the same name is overwritten.

keep_empty

[bool, optional] If set to False, then empty pdf files will be deleted automatically when closed.

metadata

[dict, optional] Information dictionary object (see PDF reference section 10.2.1 'Document Information Dictionary'), e.g.: `{'Creator': 'My software', 'Author': 'Me', 'Title': 'Awesome'}`.

The standard keys are 'Title', 'Author', 'Subject', 'Keywords', 'Creator', 'Producer', 'CreationDate', 'ModDate', and 'Trapped'.

Values have been predefined for 'Creator', 'Producer' and 'CreationDate'. They can be removed by setting them to `None`.

`attach_note`(*self*, *text*, *positionRect=[- 100, - 100, 0, 0]*)
Add a new text note to the page to be saved next. The optional positionRect specifies the position of the new note on the page. It is outside the page per default to make sure it is invisible on printouts.

`close`(*self*)
Finalize this object, making the underlying file a complete PDF file.

`get_pagecount`(*self*)
Return the current number of pages in the multipage pdf file.

`infodict`(*self*)
Return a modifiable information dictionary object (see PDF reference section 10.2.1 'Document Information Dictionary').

`keep_empty`

`savefig`(*self*, *figure=None*, ***kwargs*)
Save a *Figure* to this file as a new page.

Any other keyword arguments are passed to *savefig*.

Parameters

figure

[*Figure* or int, optional] Specifies what figure is saved to file. If not specified, the active figure is saved. If a *Figure* instance is provided, this figure is saved. If an int is specified, the figure instance to save is looked up by number.

class `matplotlib.backends.backend_pdf.Reference`(*id*)
Bases: `object`

PDF reference object.

Use PdfFile.reserveObject() to create References.

`pdfRepr`(*self*)

`write`(*self*, *contents*, *file*)

class `matplotlib.backends.backend_pdf.RendererPdf`(*file*, *image_dpi*, *height*, *width*)
Bases: `matplotlib.backends._backend_pdf_ps.RendererPDFPSBase`

`check_gc`(*self*, *gc*, *fillcolor=None*)

`draw_gouraud_triangle`(*self*, *gc*, *points*, *colors*, *trans*)
Draw a Gouraud-shaded triangle.

Parameters

gc

[*GraphicsContextBase*] The graphics context.

points

[array-like, shape=(3, 2)] Array of (x, y) points for the triangle.

colors

[array-like, shape=(3, 4)] RGBA colors for each point of the triangle.

transform

[*matplotlib.transforms.Transform*] An affine transform to apply to the points.

`draw_gouraud_triangles`(*self*, *gc*, *points*, *colors*, *trans*)

Draw a series of Gouraud triangles.

Parameters

points

[array-like, shape=(N, 3, 2)] Array of *N* (x, y) points for the triangles.

colors

[array-like, shape=(N, 3, 4)] Array of *N* RGBA colors for each point of the triangles.

transform

[*matplotlib.transforms.Transform*] An affine transform to apply to the points.

`draw_image`(*self*, *gc*, *x*, *y*, *im*, *transform=None*)

Draw an RGBA image.

Parameters

gc

[*GraphicsContextBase*] A graphics context with clipping information.

x

[scalar] The distance in physical units (i.e., dots or pixels) from the left hand side of the canvas.

y

[scalar] The distance in physical units (i.e., dots or pixels) from the bottom side of the canvas.

im

[array-like, shape=(N, M, 4), dtype=np.uint8] An array of RGBA pixels.

transform

[*matplotlib.transforms.Affine2DBase*] If and only if the concrete backend is written such that `option_scale_image()` returns `True`, an affine transformation (i.e., an *Affine2DBase*) *may* be passed to *draw_image()*. The translation vector of the transformation is given in physical units (i.e., dots or pixels). Note that the transformation does not override *x* and *y*, and has to be applied *before* translating the result by *x* and *y* (this can be accomplished by adding *x* and *y* to the translation vector defined by *transform*).

`draw_markers`(*self*, *gc*, *marker_path*, *marker_trans*, *path*, *trans*, *rgbFace=None*)

Draw a marker at each of the vertices in path.

This includes all vertices, including control points on curves. To avoid that behavior, those vertices should be removed before calling this function.

This provides a fallback implementation of draw_markers that makes multiple calls to *draw_path()*. Some backends may want to override this method in order to draw the marker only once and reuse it multiple times.

Parameters

gc

[*GraphicsContextBase*] The graphics context.

marker_trans

[*matplotlib.transforms.Transform*] An affine transform applied to the marker.

trans

[*matplotlib.transforms.Transform*] An affine transform applied to the path.

`draw_mathtext`(*self*, *gc*, *x*, *y*, *s*, *prop*, *angle*)

`draw_path`(*self*, *gc*, *path*, *transform*, *rgbFace=None*)

Draw a *Path* instance using the given affine transform.

`draw_path_collection`(*self*, *gc*, *master_transform*, *paths*, *all_transforms*, *offsets*, *offsetTrans*, *facecolors*, *edgecolors*, *linewidths*, *linestyles*, *antialiaseds*, *urls*, *offset_position*)

Draw a collection of paths selecting drawing properties from the lists *facecolors*, *edgecolors*, *linewidths*, *linestyles* and *antialiaseds*. *offsets* is a list of

offsets to apply to each of the paths. The offsets in *offsets* are first transformed by *offsetTrans* before being applied.

offset_position may be either "screen" or "data" depending on the space that the offsets are in; "data" is deprecated.

This provides a fallback implementation of *draw_path_collection()* that makes multiple calls to *draw_path()*. Some backends may want to override this in order to render each set of path data only once, and then reference that path multiple times with the different offsets, colors, styles etc. The generator methods `_iter_collection_raw_paths()` and `_iter_collection()` are provided to help with (and standardize) the implementation across backends. It is highly recommended to use those generators, so that changes to the behavior of *draw_path_collection()* can be made globally.

`draw_tex`(*self*, *gc*, *x*, *y*, *s*, *prop*, *angle*, *ismath=<deprecated parameter>*, *mtext=None*)

`draw_text`(*self*, *gc*, *x*, *y*, *s*, *prop*, *angle*, *ismath=False*, *mtext=None*)

Draw the text instance.

Parameters

gc

[*GraphicsContextBase*] The graphics context.

x

[float] The x location of the text in display coords.

y

[float] The y location of the text baseline in display coords.

s

[str] The text string.

prop

[*matplotlib.font_manager.FontProperties*] The font properties.

angle

[float] The rotation angle in degrees anti-clockwise.

mtext

[*matplotlib.text.Text*] The original text object to be rendered.

Notes

Note for backend implementers:

When you are trying to determine if you have gotten your bounding box right (which is what enables the text layout/alignment to work properly), it helps to change the line in text.py:

```
if 0: bbox_artist(self, renderer)
```

to if 1, and then the actual bounding box will be plotted along with your text.

`encode_string`(*self*, *s*, *fonttype*)

`finalize`(*self*)

`get_image_magnification`(*self*)
Get the factor by which to magnify images passed to *draw_image()*. Allows a backend to have images at a different resolution to other artists.

`merge_used_characters`(*self*, **args*, ***kwargs*)
[*Deprecated*]

Notes

Deprecated since version 3.3:

`new_gc`(*self*)
Return an instance of a *GraphicsContextBase*.

`track_characters`(*self*, **args*, ***kwargs*)
[*Deprecated*] Keep track of which characters are required from each font.

Notes

Deprecated since version 3.3.

class matplotlib.backends.backend_pdf.`Stream`(*id*, *len*, *file*, *extra=None*, *png=None*)

Bases: `object`

PDF stream object.

This has no pdfRepr method. Instead, call begin(), then output the contents of the stream by calling write(), and finally call end().

Parameters

id

[int] Object id of the stream.

len

[Reference or None] An unused Reference object for the length of the stream; None means to use a memory buffer so the length can be inlined.

file

[PdfFile] The underlying object to write the stream to.

extra

[dict from Name to anything, or None] Extra key-value pairs to include in the stream header.

png

[dict or None] If the data is already png encoded, the decode parameters.

`compressobj`

`end`(*self*)
Finalize stream.

`extra`

`file`

`id`

`len`

`pdfFile`

`pos`

`write`(*self*, *data*)
Write some data on the stream.

`class matplotlib.backends.backend_pdf.Verbatim`(*x*)
Bases: `object`

Store verbatim PDF command content for later inclusion in the stream.

`pdfRepr`(*self*)

`matplotlib.backends.backend_pdf.fill`(*strings*, *linelen=75*)
Make one string from sequence of strings, with whitespace in between.

The whitespace is chosen to form lines of at most *linelen* characters, if possible.

`matplotlib.backends.backend_pdf.pdfRepr`(*obj*)
Map Python objects to PDF syntax.

18.10.9 matplotlib.backends.backend_pgf

matplotlib.backends.backend_pgf.**FigureCanvas**
alias of *matplotlib.backends.backend_pgf.FigureCanvasPgf*

class matplotlib.backends.backend_pgf.**FigureCanvasPgf**(*figure*)
Bases: *matplotlib.backend_bases.FigureCanvasBase*

filetypes = {'pdf': 'LaTeX compiled PGF picture', 'pgf': 'LaTeX PGF picture', 'png': 'Por

get_default_filetype(*self*)
Return the default savefig file format as specified in `rcParams["savefig.format"]` (default: `'png'`).

The returned string does not include a period. This method is overridden in backends that only support a single file type.

get_renderer(*self*)

print_pdf(*self*, *fname_or_fh*, **args*, ***kwargs*)
Use LaTeX to compile a Pgf generated figure to PDF.

print_pgf(*self*, *fname_or_fh*, **args*, ***kwargs*)
Output pgf macros for drawing the figure so it can be included and rendered in latex documents.

print_png(*self*, *fname_or_fh*, **args*, ***kwargs*)
Use LaTeX to compile a pgf figure to pdf and convert it to png.

class matplotlib.backends.backend_pgf.**GraphicsContextPgf**(***kwargs*)
Bases: *matplotlib.backend_bases.GraphicsContextBase*

[*Deprecated*]

Notes

Deprecated since version 3.3:

exception matplotlib.backends.backend_pgf.**LatexError**(*message*, *latex_output=''*)
Bases: `Exception`

class matplotlib.backends.backend_pgf.**LatexManager**
Bases: `object`

The LatexManager opens an instance of the LaTeX application for determining the metrics of text elements. The LaTeX environment can be modified by setting fonts and/or a custom preamble in *rcParams*.

get_width_height_descent(*self*, *text*, *prop*)
Get the width, total height and descent for a text typeset by the current LaTeX environment.

latex_stdin_utf8(*self*)
[*Deprecated*]

Notes

Deprecated since version 3.3:

class matplotlib.backends.backend_pgf.PdfPages(*filename*, *, *keep_empty=True*, *metadata=None*)

Bases: object

A multi-page PDF file using the pgf backend

Examples

```
>>> import matplotlib.pyplot as plt
>>> # Initialize:
>>> with PdfPages('foo.pdf') as pdf:
...     # As many times as you like, create a figure fig and save it:
...     fig = plt.figure()
...     pdf.savefig(fig)
...     # When no figure is specified the current figure is saved
...     pdf.savefig()
```

Create a new PdfPages object.

Parameters

filename

[str or path-like] Plots using *PdfPages.savefig* will be written to a file at this location. Any older file with the same name is overwritten.

keep_empty

[bool, default: True] If set to False, then empty pdf files will be deleted automatically when closed.

metadata

[dict, optional] Information dictionary object (see PDF reference section 10.2.1 'Document Information Dictionary'), e.g.: `{'Creator': 'My software', 'Author': 'Me', 'Title': 'Awesome'}`.

The standard keys are 'Title', 'Author', 'Subject', 'Keywords', 'Creator', 'Producer', 'CreationDate', 'ModDate', and 'Trapped'. Values have been predefined for 'Creator', 'Producer' and 'CreationDate'. They can be removed by setting them to None.

close(*self*)
Finalize this object, running LaTeX in a temporary directory and moving the final pdf file to *filename*.

get_pagecount(*self*)
Return the current number of pages in the multipage pdf file.

keep_empty

property metadata

savefig(*self*, *figure=None*, ***kwargs*)
Save a *Figure* to this file as a new page.

Any other keyword arguments are passed to *savefig*.

Parameters

figure

[*Figure* or int, optional] Specifies what figure is saved to file. If not specified, the active figure is saved. If a *Figure* instance is provided, this figure is saved. If an int is specified, the figure instance to save is looked up by number.

class matplotlib.backends.backend_pgf.RendererPgf(*figure*, *fh*, *dummy=<deprecated parameter>*)
Bases: *matplotlib.backend_bases.RendererBase*

Create a new PGF renderer that translates any drawing instruction into text commands to be interpreted in a latex pgfpicture environment.

Attributes

figure

[*matplotlib.figure.Figure*] Matplotlib figure to initialize height, width and dpi from.

fh

[file-like] File handle for the output of the drawing commands.

draw_image(*self*, *gc*, *x*, *y*, *im*, *transform=None*)
Draw an RGBA image.

Parameters

gc

[*GraphicsContextBase*] A graphics context with clipping information.

x

[scalar] The distance in physical units (i.e., dots or pixels) from the left hand side of the canvas.

y

[scalar] The distance in physical units (i.e., dots or pixels) from the bottom side of the canvas.

im

[array-like, shape=(N, M, 4), dtype=np.uint8] An array of RGBA pixels.

transform

[*matplotlib.transforms.Affine2DBase*] If and only if the concrete backend is written such that *option_scale_image()* returns `True`, an affine transformation (i.e., an *Affine2DBase*) *may* be passed to *draw_image()*. The translation vector of the transformation is given in physical units (i.e., dots or pixels). Note that the transformation does not override *x* and *y*, and has to be applied *before* translating the result by *x* and *y* (this can be accomplished by adding *x* and *y* to the translation vector defined by *transform*).

`draw_markers`(*self*, *gc*, *marker_path*, *marker_trans*, *path*, *trans*, *rgbFace=None*)

Draw a marker at each of the vertices in path.

This includes all vertices, including control points on curves. To avoid that behavior, those vertices should be removed before calling this function.

This provides a fallback implementation of draw_markers that makes multiple calls to *draw_path()*. Some backends may want to override this method in order to draw the marker only once and reuse it multiple times.

Parameters

gc

[*GraphicsContextBase*] The graphics context.

marker_trans

[*matplotlib.transforms.Transform*] An affine transform applied to the marker.

trans

[*matplotlib.transforms.Transform*] An affine transform applied to the path.

`draw_path`(*self*, *gc*, *path*, *transform*, *rgbFace=None*)
Draw a *Path* instance using the given affine transform.

`draw_tex`(*self*, *gc*, *x*, *y*, *s*, *prop*, *angle*, *ismath='TeX!'*, *mtext=None*)

`draw_text`(*self*, *gc*, *x*, *y*, *s*, *prop*, *angle*, *ismath=False*, *mtext=None*)
Draw the text instance.

Parameters

gc

[*GraphicsContextBase*] The graphics context.

x

[float] The x location of the text in display coords.

y

[float] The y location of the text baseline in display coords.

s

[str] The text string.

prop

[*matplotlib.font_manager.FontProperties*] The font properties.

angle

[float] The rotation angle in degrees anti-clockwise.

mtext

[*matplotlib.text.Text*] The original text object to be rendered.

Notes

Note for backend implementers:

When you are trying to determine if you have gotten your bounding box right (which is what enables the text layout/alignment to work properly), it helps to change the line in text.py:

```
if 0: bbox_artist(self, renderer)
```

to if 1, and then the actual bounding box will be plotted along with your text.

`flipy`(*self*)
Return whether y values increase from top to bottom.

Note that this only affects drawing of texts and images.

`get_canvas_width_height`(*self*)
Return the canvas width and height in display coords.

get_text_width_height_descent(*self*, *s*, *prop*, *ismath*)
Get the width, height, and descent (offset from the bottom to the baseline), in display coords, of the string *s* with *FontProperties* *prop*.

property latexManager

option_image_nocomposite(*self*)
Return whether image composition by Matplotlib should be skipped.

Raster backends should usually return False (letting the C-level rasterizer take care of image composition); vector backends should usually return `not rcParams["image.composite_image"]`.

option_scale_image(*self*)
Return whether arbitrary affine transformations in *draw_image()* are supported (True for most vector backends).

points_to_pixels(*self*, *points*)
Convert points to display units.

You need to override this function (unless your backend doesn't have a dpi, e.g., postscript or svg). Some imaging systems assume some value for pixels per inch:

```
points to pixels = points * pixels_per_inch/72 * dpi/72
```

Parameters

points

[float or array-like] a float or a numpy array of float

Returns

Points converted to pixels

class matplotlib.backends.backend_pgf.TmpDirCleaner
Bases: object

static add(*tmpdir*)

static cleanup_remaining_tmpdirs()

remaining_tmpdirs = {}

matplotlib.backends.backend_pgf.common_texification(*text*)
Do some necessary and/or useful substitutions for texts to be included in LaTeX documents.

This distinguishes text-mode and math-mode by replacing the math separator `$` with `\(\displaystyle %s\)`. Escaped math separators (`\$`) are ignored.

The following characters are escaped in text segments: `_^$%`

matplotlib.backends.backend_pgf.**get_fontspec**()
Build fontspec preamble from rc.

matplotlib.backends.backend_pgf.**get_preamble**()
Get LaTeX preamble from rc.

matplotlib.backends.backend_pgf.**make_pdf_to_png_converter**()
Return a function that converts a pdf file to a png file.

matplotlib.backends.backend_pgf.**repl_escapetext**(*m*)
[*Deprecated*]

Notes

Deprecated since version 3.2:

matplotlib.backends.backend_pgf.**repl_mathdefault**(*m*)
[*Deprecated*]

Notes

Deprecated since version 3.2:

matplotlib.backends.backend_pgf.**writeln**(*fh*, *line*)

18.10.10 matplotlib.backends.backend_ps

A PostScript backend, which can produce both PostScript .ps and .eps.

matplotlib.backends.backend_ps.**FigureCanvas**
alias of *matplotlib.backends.backend_ps.FigureCanvasPS*

class matplotlib.backends.backend_ps.**FigureCanvasPS**(*figure*)
Bases: *matplotlib.backend_bases.FigureCanvasBase*

filetypes = {'eps': 'Encapsulated Postscript', 'ps': 'Postscript'}

fixed_dpi = 72

get_default_filetype(*self*)
Return the default savefig file format as specified in `rcParams["savefig.format"]` (default: `'png'`).

The returned string does not include a period. This method is overridden in backends that only support a single file type.

print_eps(*self*, *outfile*, **args*, ***kwargs*)

print_ps(*self*, *outfile*, **args*, ***kwargs*)

class matplotlib.backends.backend_ps.**GraphicsContextPS**
Bases: *matplotlib.backend_bases.GraphicsContextBase*

get_capstyle(*self*)
Return the capstyle as a string in ('butt', 'round', 'projecting').

get_joinstyle(*self*)
Return the line join style as one of ('miter', 'round', 'bevel').

class matplotlib.backends.backend_ps.**PsBackendHelper**
Bases: object

class matplotlib.backends.backend_ps.**RendererPS**(*width*, *height*, *pswriter*, *imagedpi=72*)
Bases: matplotlib.backends._backend_pdf_ps.RendererPDFPSBase

The renderer handles all the drawing primitives using a graphics context instance that controls the colors/styles.

create_hatch(*self*, *hatch*)

draw_gouraud_triangle(*self*, *gc*, *points*, *colors*, *trans*)
Draw a Gouraud-shaded triangle.

Parameters

gc

[*GraphicsContextBase*] The graphics context.

points

[array-like, shape=(3, 2)] Array of (x, y) points for the triangle.

colors

[array-like, shape=(3, 4)] RGBA colors for each point of the triangle.

transform

[*matplotlib.transforms.Transform*] An affine transform to apply to the points.

draw_gouraud_triangles(*self*, *gc*, *points*, *colors*, *trans*)
Draw a series of Gouraud triangles.

Parameters

points

[array-like, shape=(N, 3, 2)] Array of *N* (x, y) points for the triangles.

colors

[array-like, shape=(N, 3, 4)] Array of *N* RGBA colors for each point of the triangles.

transform

[*matplotlib.transforms.Transform*] An affine transform to apply to the points.

`draw_image`(*self*, *gc*, *x*, *y*, *im*, *transform=None*)

Draw an RGBA image.

Parameters

gc

[*GraphicsContextBase*] A graphics context with clipping information.

x

[scalar] The distance in physical units (i.e., dots or pixels) from the left hand side of the canvas.

y

[scalar] The distance in physical units (i.e., dots or pixels) from the bottom side of the canvas.

im

[array-like, shape=(N, M, 4), dtype=np.uint8] An array of RGBA pixels.

transform

[*matplotlib.transforms.Affine2DBase*] If and only if the concrete backend is written such that `option_scale_image()` returns `True`, an affine transformation (i.e., an *Affine2DBase*) *may* be passed to *draw_image()*. The translation vector of the transformation is given in physical units (i.e., dots or pixels). Note that the transformation does not override *x* and *y*, and has to be applied *before* translating the result by *x* and *y* (this can be accomplished by adding *x* and *y* to the translation vector defined by *transform*).

`draw_markers`(*self*, *gc*, *marker_path*, *marker_trans*, *path*, *trans*, *rgbFace=None*)

Draw a marker at each of the vertices in path.

This includes all vertices, including control points on curves. To avoid that behavior, those vertices should be removed before calling this function.

This provides a fallback implementation of draw_markers that makes multiple calls to *draw_path()*. Some backends may want to override this method in order to draw the marker only once and reuse it multiple times.

Parameters

gc

[*GraphicsContextBase*] The graphics context.

marker_trans

[*matplotlib.transforms.Transform*] An affine transform applied to the marker.

trans

[*matplotlib.transforms.Transform*] An affine transform applied to the path.

`draw_mathtext`(*self*, *gc*, *x*, *y*, *s*, *prop*, *angle*)
Draw the math text using matplotlib.mathtext.

`draw_path`(*self*, *gc*, *path*, *transform*, *rgbFace=None*)
Draw a *Path* instance using the given affine transform.

`draw_path_collection`(*self*, *gc*, *master_transform*, *paths*, *all_transforms*, *offsets*, *offsetTrans*, *facecolors*, *edgecolors*, *linewidths*, *linestyles*, *antialiaseds*, *urls*, *offset_position*)
Draw a collection of paths selecting drawing properties from the lists *facecolors*, *edgecolors*, *linewidths*, *linestyles* and *antialiaseds*. *offsets* is a list of offsets to apply to each of the paths. The offsets in *offsets* are first transformed by *offsetTrans* before being applied.

offset_position may be either "screen" or "data" depending on the space that the offsets are in; "data" is deprecated.

This provides a fallback implementation of *draw_path_collection()* that makes multiple calls to *draw_path()*. Some backends may want to override this in order to render each set of path data only once, and then reference that path multiple times with the different offsets, colors, styles etc. The generator methods `_iter_collection_raw_paths()` and `_iter_collection()` are provided to help with (and standardize) the implementation across backends. It is highly recommended to use those generators, so that changes to the behavior of *draw_path_collection()* can be made globally.

`draw_tex`(*self*, *gc*, *x*, *y*, *s*, *prop*, *angle*, *ismath=<deprecated parameter>*, *mtext=None*)

`draw_text`(*self*, *gc*, *x*, *y*, *s*, *prop*, *angle*, *ismath=False*, *mtext=None*)
Draw the text instance.

Parameters

gc

[*GraphicsContextBase*] The graphics context.

x

[float] The x location of the text in display coords.

y

[float] The y location of the text baseline in display coords.

s

[str] The text string.

prop

[*matplotlib.font_manager.FontProperties*] The font properties.

angle

[float] The rotation angle in degrees anti-clockwise.

mtext

[*matplotlib.text.Text*] The original text object to be rendered.

Notes

Note for backend implementers:

When you are trying to determine if you have gotten your bounding box right (which is what enables the text layout/alignment to work properly), it helps to change the line in text.py:

```
if 0: bbox_artist(self, renderer)
```

to if 1, and then the actual bounding box will be plotted along with your text.

`get_image_magnification`(*self*)
Get the factor by which to magnify images passed to draw_image. Allows a backend to have images at a different resolution to other artists.

`merge_used_characters`(*self*, **args*, ***kwargs*)
[*Deprecated*]

Notes

Deprecated since version 3.3:

`new_gc`(*self*)
Return an instance of a *GraphicsContextBase*.

`set_color`(*self*, *r*, *g*, *b*, *store=True*)

`set_font`(*self*, *fontname*, *fontsize*, *store=True*)

`set_linecap`(*self*, *linecap*, *store=True*)

`set_linedash`(*self*, *offset*, *seq*, *store=True*)

`set_linejoin`(*self*, *linejoin*, *store=True*)

set_linewidth(*self*, *linewidth*, *store=True*)

track_characters(*self*, **args*, ***kwargs*)

[*Deprecated*] Keep track of which characters are required from each font.

Notes

Deprecated since version 3.3.

property used_characters

matplotlib.backends.backend_ps.**convert_psfrags**(*tmpfile*, *psfrags*, *font_preamble*, *custom_preamble*, *paper_width*, *paper_height*, *orientation*)

When we want to use the LaTeX backend with postscript, we write PSFrag tags to a temporary postscript file, each one marking a position for LaTeX to render some text. convert_psfrags generates a LaTeX document containing the commands to convert those tags to text. LaTeX/dvips produces the postscript file that includes the actual text.

matplotlib.backends.backend_ps.**get_bbox_header**(*lbrt*, *rotated=False*)

Return a postscript header string for the given bbox lbrt=(l, b, r, t). Optionally, return rotate command.

matplotlib.backends.backend_ps.**gs_distill**(*tmpfile*, *eps=False*, *ptype='letter'*, *bbox=None*, *rotated=False*)

Use ghostscript's pswrite or epswrite device to distill a file. This yields smaller files without illegal encapsulated postscript operators. The output is low-level, converting text to outlines.

matplotlib.backends.backend_ps.**pstoeps**(*tmpfile*, *bbox=None*, *rotated=False*)

Convert the postscript to encapsulated postscript. The bbox of the eps file will be replaced with the given *bbox* argument. If None, original bbox will be used.

matplotlib.backends.backend_ps.**quote_ps_string**(*s*)

Quote dangerous characters of S for use in a PostScript string constant.

matplotlib.backends.backend_ps.**xpdf_distill**(*tmpfile*, *eps=False*, *ptype='letter'*, *bbox=None*, *rotated=False*)

Use ghostscript's ps2pdf and xpdf's/poppler's pdftops to distill a file. This yields smaller files without illegal encapsulated postscript operators. This distiller is preferred, generating high-level postscript output that treats text as text.

18.10.11 matplotlib.backends.backend_qt4agg

NOTE Not included, to avoid adding a dependency to building the docs.

18.10.12 matplotlib.backends.backend_qt4cairo

NOTE Not included, to avoid adding a dependency to building the docs.

18.10.13 matplotlib.backends.backend_qt5agg

NOTE Not included, to avoid adding a dependency to building the docs.

18.10.14 matplotlib.backends.backend_qt5cairo

NOTE Not included, to avoid adding a dependency to building the docs.

18.10.15 matplotlib.backends.backend_svg

matplotlib.backends.backend_svg.**FigureCanvas**
alias of *matplotlib.backends.backend_svg.FigureCanvasSVG*

class matplotlib.backends.backend_svg.**FigureCanvasSVG**(*figure*)
Bases: *matplotlib.backend_bases.FigureCanvasBase*

filetypes = {'svg': 'Scalable Vector Graphics', 'svgz': 'Scalable Vector Graphics'}

fixed_dpi = 72

get_default_filetype(*self*)
Return the default savefig file format as specified in `rcParams["savefig.format"]` (default: `'png'`).

The returned string does not include a period. This method is overridden in backends that only support a single file type.

print_svg(*self*, *filename*, **args*, ***kwargs*)

Parameters

filename

[str or path-like or file-like] Output target; if a string, a file will be opened for writing.

metadata

[Dict[str, Any], optional] Metadata in the SVG file defined as key-value pairs of strings, datetimes, or lists

of strings, e.g., `{'Creator': 'My software', 'Contributor': ['Me', 'My Friend'], 'Title': 'Awesome'}`.

The standard keys and their value types are:

- *str*: `'Coverage'`, `'Description'`, `'Format'`, `'Identifier'`, `'Language'`, `'Relation'`, `'Source'`, `'Title'`, and `'Type'`.
- *str* or *list of str*: `'Contributor'`, `'Creator'`, `'Keywords'`, `'Publisher'`, and `'Rights'`.
- *str*, *date*, *datetime*, or *tuple* of same: `'Date'`. If a non-*str*, then it will be formatted as ISO 8601.

Values have been predefined for `'Creator'`, `'Date'`, `'Format'`, and `'Type'`. They can be removed by setting them to `None`.

Information is encoded as Dublin Core Metadata.

`print_svgz`(*self*, *filename*, **args*, ***kwargs*)

class matplotlib.backends.backend_svg.**RendererSVG**(*width*, *height*, *svgwriter*, *basename=None*, *image_dpi=72*, ***, *metadata=None*)

Bases: *matplotlib.backend_bases.RendererBase*

`close_group`(*self*, *s*)

Close a grouping element with label *s*.

Only used by the SVG renderer.

`draw_gouraud_triangle`(*self*, *gc*, *points*, *colors*, *trans*)

Draw a Gouraud-shaded triangle.

Parameters

gc

[*GraphicsContextBase*] The graphics context.

points

[array-like, shape=(3, 2)] Array of (x, y) points for the triangle.

colors

[array-like, shape=(3, 4)] RGBA colors for each point of the triangle.

transform

[*matplotlib.transforms.Transform*] An affine transform to apply to the points.

`draw_gouraud_triangles`(*self*, *gc*, *triangles_array*, *colors_array*, *transform*)

Draw a series of Gouraud triangles.

Parameters

points

[array-like, shape=(N, 3, 2)] Array of *N* (x, y) points for the triangles.

colors

[array-like, shape=(N, 3, 4)] Array of *N* RGBA colors for each point of the triangles.

transform

[*matplotlib.transforms.Transform*] An affine transform to apply to the points.

`draw_image`(*self*, *gc*, *x*, *y*, *im*, *transform=None*)

Draw an RGBA image.

Parameters

gc

[*GraphicsContextBase*] A graphics context with clipping information.

x

[scalar] The distance in physical units (i.e., dots or pixels) from the left hand side of the canvas.

y

[scalar] The distance in physical units (i.e., dots or pixels) from the bottom side of the canvas.

im

[array-like, shape=(N, M, 4), dtype=np.uint8] An array of RGBA pixels.

transform

[*matplotlib.transforms.Affine2DBase*] If and only if the concrete backend is written such that *option_scale_image()* returns `True`, an affine transformation (i.e., an *Affine2DBase*) *may* be passed to *draw_image()*. The translation vector of the transformation is given in physical units (i.e., dots or pixels). Note that the transformation does not override *x* and *y*, and has to be applied *before* translating the result by *x* and *y* (this can be accomplished by adding *x* and *y* to the translation vector defined by *transform*).

`draw_markers`(*self*, *gc*, *marker_path*, *marker_trans*, *path*, *trans*, *rgbFace=None*)

Draw a marker at each of the vertices in path.

This includes all vertices, including control points on curves. To avoid that behavior, those vertices should be removed before calling this function.

This provides a fallback implementation of draw_markers that makes multiple calls to *draw_path()*. Some backends may want to override this method in order to draw the marker only once and reuse it multiple times.

Parameters

gc

[*GraphicsContextBase*] The graphics context.

marker_trans

[*matplotlib.transforms.Transform*] An affine transform applied to the marker.

trans

[*matplotlib.transforms.Transform*] An affine transform applied to the path.

`draw_path`(*self*, *gc*, *path*, *transform*, *rgbFace=None*)

Draw a *Path* instance using the given affine transform.

`draw_path_collection`(*self*, *gc*, *master_transform*, *paths*, *all_transforms*, *offsets*, *offsetTrans*, *facecolors*, *edgecolors*, *linewidths*, *linestyles*, *antialiaseds*, *urls*, *offset_position*)

Draw a collection of paths selecting drawing properties from the lists *facecolors*, *edgecolors*, *linewidths*, *linestyles* and *antialiaseds*. *offsets* is a list of offsets to apply to each of the paths. The offsets in *offsets* are first transformed by *offsetTrans* before being applied.

offset_position may be either "screen" or "data" depending on the space that the offsets are in; "data" is deprecated.

This provides a fallback implementation of *draw_path_collection()* that makes multiple calls to *draw_path()*. Some backends may want to override this in order to render each set of path data only once, and then reference that path multiple times with the different offsets, colors, styles etc. The generator methods `_iter_collection_raw_paths()` and `_iter_collection()` are provided to help with (and standardize) the implementation across backends. It is highly recommended to use those generators, so that changes to the behavior of *draw_path_collection()* can be made globally.

`draw_tex`(*self*, *gc*, *x*, *y*, *s*, *prop*, *angle*, *ismath=<deprecated parameter>*, *mtext=None*)

`draw_text`(*self*, *gc*, *x*, *y*, *s*, *prop*, *angle*, *ismath=False*, *mtext=None*)

Draw the text instance.

Parameters

gc

[*GraphicsContextBase*] The graphics context.

x

[float] The x location of the text in display coords.

y

[float] The y location of the text baseline in display coords.

s

[str] The text string.

prop

[*matplotlib.font_manager.FontProperties*] The font properties.

angle

[float] The rotation angle in degrees anti-clockwise.

mtext

[*matplotlib.text.Text*] The original text object to be rendered.

Notes

Note for backend implementers:

When you are trying to determine if you have gotten your bounding box right (which is what enables the text layout/alignment to work properly), it helps to change the line in text.py:

```
if 0: bbox_artist(self, renderer)
```

to if 1, and then the actual bounding box will be plotted along with your text.

`finalize`(*self*)

`flipy`(*self*)
Return whether y values increase from top to bottom.

Note that this only affects drawing of texts and images.

`get_canvas_width_height`(*self*)
Return the canvas width and height in display coords.

`get_image_magnification`(*self*)
Get the factor by which to magnify images passed to *draw_image()*. Allows a backend to have images at a different resolution to other artists.

`get_text_width_height_descent`(*self*, *s*, *prop*, *ismath*)
Get the width, height, and descent (offset from the bottom to the baseline), in display coords, of the string *s* with *FontProperties* *prop*.

`open_group`(*self*, *s*, *gid=None*)
Open a grouping element with label *s* and *gid* (if set) as id.

Only used by the SVG renderer.

`option_image_nocomposite`(*self*)
Return whether image composition by Matplotlib should be skipped.

Raster backends should usually return False (letting the C-level rasterizer take care of image composition); vector backends should usually return `not rcParams["image.composite_image"]`.

`option_scale_image`(*self*)
Return whether arbitrary affine transformations in *draw_image()* are supported (True for most vector backends).

`class matplotlib.backends.backend_svg.XMLWriter`(*file*)
Bases: `object`

Parameters

file

[writable text file-like object]

`close`(*self*, *id*)
Close open elements, up to (and including) the element identified by the given identifier.

Parameters

id

Element identifier, as returned by the *start()* method.

`comment`(*self*, *comment*)
Add a comment to the output stream.

Parameters

comment

[str] Comment text.

`data`(*self*, *text*)
Add character data to the output stream.

Parameters

text

[str] Character data.

element(*self*, *tag*, *text=None*, *attrib={}*, ***extra*)
Add an entire element. This is the same as calling *start()*, *data()*, and *end()* in sequence. The *text* argument can be omitted.

end(*self*, *tag=None*, *indent=True*)
Close the current element (opened by the most recent call to *start()*).

Parameters

tag

Element tag. If given, the tag must match the start tag. If omitted, the current element is closed.

flush(*self*)
Flush the output stream.

start(*self*, *tag*, *attrib={}*, ***extra*)
Open a new element. Attributes can be given as keyword arguments, or as a string/string dictionary. The method returns an opaque identifier that can be passed to the *close()* method, to close all open elements up to and including this one.

Parameters

tag

Element tag.

attrib

Attribute dictionary. Alternatively, attributes can be given as keyword arguments.

Returns

An element identifier.

matplotlib.backends.backend_svg.**escape_attrib**(*s*)

matplotlib.backends.backend_svg.**escape_cdata**(*s*)

matplotlib.backends.backend_svg.**escape_comment**(*s*)

matplotlib.backends.backend_svg.**generate_css**(*attrib={}*)

matplotlib.backends.backend_svg.**generate_transform**(*transform_list=[]*)

matplotlib.backends.backend_svg.**short_float_fmt**(*x*)
Create a short string representation of a float, which is %f formatting with trailing zeros and the decimal point removed.

18.10.16 matplotlib.backends.backend_tkagg

matplotlib.backends.backend_tkagg.**FigureCanvas**
alias of *matplotlib.backends.backend_tkagg.FigureCanvasTkAgg*

class matplotlib.backends.backend_tkagg.**FigureCanvasTkAgg**(*figure*, *master=None*, *resize_callback=None*)
Bases: *matplotlib.backends.backend_agg.FigureCanvasAgg*, matplotlib.backends._backend_tk.FigureCanvasTk

blit(*self*, *bbox=None*)
Blit the canvas in bbox (default entire canvas).

draw(*self*)
Render the *Figure*.

18.10.17 matplotlib.backends.backend_webagg

Note: The WebAgg backend is not documented here, in order to avoid adding Tornado to the doc build requirements.

18.10.18 matplotlib.backends.backend_wxagg

NOTE Not included, to avoid adding a dependency to building the docs.

18.11 matplotlib.bezier

A module providing some utility functions regarding Bezier path manipulation.

class matplotlib.bezier.**BezierSegment**(*control_points*)
Bases: object

A d-dimensional Bezier segment.

Parameters

control_points

[(N, d) array] Location of the *N* control points.

axis_aligned_extrema(*self*)
Return the dimension and location of the curve's interior extrema.

The extrema are the points along the curve where one of its partial derivatives is zero.

Returns

dims

[int, array_like] Index i of the partial derivative which is zero at each interior extrema.

dzeros

[float, array_like] Of same size as dims. The t such that $d/dx_i B(t) = 0$

property control_points
The control points of the curve.

property degree
Degree of the polynomial. One less the number of control points.

property dimension
The dimension of the curve.

point_at_t(*self*, *t*)
Evaluate curve at a single point *t*. Returns a Tuple[float*d].

property polynomial_coefficients
The polynomial coefficients of the Bezier curve.

Warning: Follows opposite convention from `numpy.polyval`.

Returns

float, (n+1, d) array_like

Coefficients after expanding in polynomial basis, where n is the degree of the bezier curve and d its dimension. These are the numbers (C_j) such that the curve can be written $\sum_{j=0}^{n} C_j t^j$.

Notes

The coefficients are calculated as

$$\binom{n}{j} \sum_{i=0}^{j} (-1)^{i+j} \binom{j}{i} P_i$$

where P_i are the control points of the curve.

exception matplotlib.bezier.NonIntersectingPathException
Bases: `ValueError`

matplotlib.bezier.**check_if_parallel**(*dx1*, *dy1*, *dx2*, *dy2*, *tolerance=1e-05*)

Check if two lines are parallel.

Parameters

dx1, dy1, dx2, dy2

[float] The gradients *dy*/*dx* of the two lines.

tolerance

[float] The angular tolerance in radians up to which the lines are considered parallel.

Returns

is_parallel

- 1 if two lines are parallel in same direction.
- -1 if two lines are parallel in opposite direction.
- False otherwise.

matplotlib.bezier.**concatenate_paths**(*paths*)

[*Deprecated*] Concatenate a list of paths into a single path.

Notes

Deprecated since version 3.3.

matplotlib.bezier.**find_bezier_t_intersecting_with_closedpath**(*bezier_point_at_t*, *inside_closedpath*, *t0=0.0*, *t1=1.0*, *tolerance=0.01*)

Find the intersection of the Bezier curve with a closed path.

The intersection point *t* is approximated by two parameters *t0*, *t1* such that *t0* <= *t* <= *t1*.

Search starts from *t0* and *t1* and uses a simple bisecting algorithm therefore one of the end points must be inside the path while the other doesn't. The search stops when the distance of the points parametrized by *t0* and *t1* gets smaller than the given *tolerance*.

Parameters

bezier_point_at_t

[callable] A function returning x, y coordinates of the Bezier at parameter *t*. It must have the signature:

```
bezier_point_at_t(t: float) -> Tuple[float, float]
```

inside_closedpath

[callable] A function returning True if a given point (x, y) is inside the closed path. It must have the signature:

```
inside_closedpath(point: Tuple[float, float]) -> bool
```

t0, t1

[float] Start parameters for the search.

tolerance

[float] Maximal allowed distance between the final points.

Returns

t0, t1

[float] The Bezier path parameters.

matplotlib.bezier.**find_control_points**(*c1x*, *c1y*, *mmx*, *mmy*, *c2x*, *c2y*)
Find control points of the Bezier curve passing through (*c1x*, *c1y*), (*mmx*, *mmy*), and (*c2x*, *c2y*), at parametric values 0, 0.5, and 1.

matplotlib.bezier.**get_cos_sin**(*x0*, *y0*, *x1*, *y1*)

matplotlib.bezier.**get_intersection**(*cx1*, *cy1*, *cos_t1*, *sin_t1*, *cx2*, *cy2*, *cos_t2*, *sin_t2*)
Return the intersection between the line through (*cx1*, *cy1*) at angle *t1* and the line through (*cx2*, *cy2*) at angle *t2*.

matplotlib.bezier.**get_normal_points**(*cx*, *cy*, *cos_t*, *sin_t*, *length*)
For a line passing through (*cx*, *cy*) and having an angle *t*, return locations of the two points located along its perpendicular line at the distance of *length*.

matplotlib.bezier.**get_parallels**(*bezier2*, *width*)
Given the quadratic Bezier control points *bezier2*, returns control points of quadratic Bezier lines roughly parallel to given one separated by *width*.

matplotlib.bezier.**inside_circle**(*cx*, *cy*, *r*)
Return a function that checks whether a point is in a circle with center (*cx*, *cy*) and radius *r*.

The returned function has the signature:

```
f(xy: Tuple[float, float]) -> bool
```

matplotlib.bezier.**make_path_regular**(*p*)

[*Deprecated*] If the `codes` attribute of *Path* *p* is None, return a copy of *p* with `codes` set to (MOVETO, LINETO, LINETO, ..., LINETO); otherwise return *p* itself.

Notes

Deprecated since version 3.3.

matplotlib.bezier.**make_wedged_bezier2**(*bezier2*, *width*, *w1=1.0*, *wm=0.5*, *w2=0.0*)

Being similar to get_parallels, returns control points of two quadratic Bezier lines having a width roughly parallel to given one separated by *width*.

matplotlib.bezier.**split_bezier_intersecting_with_closedpath**(*bezier*, *inside_closedpath*, *tolerance=0.01*)

Split a Bezier curve into two at the intersection with a closed path.

Parameters

bezier

[array-like(N, 2)] Control points of the Bezier segment. See *BezierSegment*.

inside_closedpath

[callable] A function returning True if a given point (x, y) is inside the closed path. See also *find_bezier_t_intersecting_with_closedpath*.

tolerance

[float] The tolerance for the intersection. See also *find_bezier_t_intersecting_with_closedpath*.

Returns

left, right

Lists of control points for the two Bezier segments.

matplotlib.bezier.**split_de_casteljau**(*beta*, *t*)

Split a Bezier segment defined by its control points *beta* into two separate segments divided at *t* and return their control points.

matplotlib.bezier.**split_path_inout**(*path*, *inside*, *tolerance=0.01*, *reorder_inout=False*)

Divide a path into two segments at the point where `inside(x, y)` becomes False.

18.12 matplotlib.blocking_input

Classes used for blocking interaction with figure windows:

BlockingInput

Creates a callable object to retrieve events in a blocking way for interactive sessions. Base class of the other classes listed here.

BlockingKeyMouseInput

Creates a callable object to retrieve key or mouse clicks in a blocking way for interactive sessions. Used by *waitforbuttonpress*.

BlockingMouseInput

Creates a callable object to retrieve mouse clicks in a blocking way for interactive sessions. Used by *ginput*.

BlockingContourLabeler

Creates a callable object to retrieve mouse clicks in a blocking way that will then be used to place labels on a *ContourSet*. Used by *clabel*.

class matplotlib.blocking_input.**BlockingContourLabeler**(*cs*)

Bases: *matplotlib.blocking_input.BlockingMouseInput*

Callable for retrieving mouse clicks and key presses in a blocking way.

Used to place contour labels.

add_click(*self*, *event*)

Add the coordinates of an event to the list of clicks.

Parameters

event

[*MouseEvent*]

button1(*self*, *event*)

Process an button-1 event (add a label to a contour).

Parameters

event

[*MouseEvent*]

button3(*self*, *event*)

Process an button-3 event (remove a label if not in inline mode).

Unfortunately, if one is doing inline labels, then there is currently no way to fix the broken contour - once humpty-dumpty is broken, he can't be put back together. In inline mode, this does nothing.

Parameters

event

[*MouseEvent*]

`pop_click`(*self*, *event*, *index=- 1*)
Remove a click (by default, the last) from the list of clicks.

Parameters

event

[*MouseEvent*]

`class matplotlib.blocking_input.BlockingInput`(*fig*, *eventslist=()*)
Bases: `object`

Callable for retrieving events in a blocking way.

`add_event`(*self*, *event*)
For base class, this just appends an event to events.

`cleanup`(*self*)
Disconnect all callbacks.

`on_event`(*self*, *event*)
Event handler; will be passed to the current figure to retrieve events.

`pop`(*self*, *index=- 1*)
Remove an event from the event list -- by default, the last.

Note that this does not check that there are events, much like the normal pop method. If no events exist, this will throw an exception.

`pop_event`(*self*, *index=- 1*)
Remove an event from the event list -- by default, the last.

Note that this does not check that there are events, much like the normal pop method. If no events exist, this will throw an exception.

`post_event`(*self*)
For baseclass, do nothing but collect events.

`class matplotlib.blocking_input.BlockingKeyMouseInput`(*fig*)
Bases: *matplotlib.blocking_input.BlockingInput*

Callable for retrieving mouse clicks and key presses in a blocking way.

`post_event`(*self*)
Determine if it is a key event.

class matplotlib.blocking_input.BlockingMouseInput(*fig*, *mouse_add=<MouseButton.LEFT: 1>*, *mouse_pop=<MouseButton.RIGHT: 3>*, *mouse_stop=<MouseButton.MIDDLE: 2>*)

Bases: *matplotlib.blocking_input.BlockingInput*

Callable for retrieving mouse clicks in a blocking way.

This class will also retrieve keypresses and map them to mouse clicks: delete and backspace are a right click, enter is like a middle click, and all others are like a left click.

add_click(*self*, *event*)

Add the coordinates of an event to the list of clicks.

Parameters

event

[*MouseEvent*]

button_add = 1

button_pop = 3

button_stop = 2

cleanup(*self*, *event=None*)

Parameters

event

[*MouseEvent*, optional] Not used

key_event(*self*)

Process a key press event, mapping keys to appropriate mouse clicks.

mouse_event(*self*)

Process a mouse click event.

mouse_event_add(*self*, *event*)

Process an button-1 event (add a click if inside axes).

Parameters

event

[*MouseEvent*]

mouse_event_pop(*self*, *event*)

Process an button-3 event (remove the last click).

Parameters

event

[*MouseEvent*]

`mouse_event_stop`(*self*, *event*)
Process an button-2 event (end blocking input).

Parameters

event

[*MouseEvent*]

`pop`(*self*, *event*, *index=- 1*)
Remove a click and the associated event from the list of clicks.

Defaults to the last click.

`pop_click`(*self*, *event*, *index=- 1*)
Remove a click (by default, the last) from the list of clicks.

Parameters

event

[*MouseEvent*]

`post_event`(*self*)
Process an event.

18.13 matplotlib.category

Plotting of string "category" data: `plot(['d', 'f', 'a'], [1, 2, 3])` will plot three points with x-axis values of 'd', 'f', 'a'.

See /gallery/lines_bars_and_markers/categorical_variables for an example.

The module uses Matplotlib's *matplotlib.units* mechanism to convert from strings to integers and provides a tick locator, a tick formatter, and the *UnitData* class that creates and stores the string-to-integer mapping.

class matplotlib.category.**StrCategoryConverter**
Bases: *matplotlib.units.ConversionInterface*

`static axisinfo`(*unit*, *axis*)
Set the default axis ticks and labels.

Parameters

unit

[*UnitData*] object string unit information for value

axis

[*Axis*] axis for which information is being set

Returns

AxisInfo

Information to support default tick labeling

`static convert(`*value*, *unit*, *axis*`)`

Convert strings in *value* to floats using mapping information stored in the *unit* object.

Parameters

value

[str or iterable] Value or list of values to be converted.

unit

[*UnitData*] An object mapping strings to integers.

axis

[*Axis*] The axis on which the converted value is plotted.

Note: *axis* is unused.

Returns

float or ndarray[float]

`static default_units(`*data*, *axis*`)`

Set and update the *Axis* units.

Parameters

data

[str or iterable of str]

axis

[*Axis*] axis on which the data is plotted

Returns

UnitData

object storing string to integer mapping

class matplotlib.category.**StrCategoryFormatter**(*units_mapping*)

Bases: *matplotlib.ticker.Formatter*

String representation of the data at every tick.

Parameters

units_mapping

[dict] Mapping of category names (str) to indices (int).

format_ticks(*self*, *values*)

Return the tick labels for all the ticks at once.

class matplotlib.category.**StrCategoryLocator**(*units_mapping*)

Bases: *matplotlib.ticker.Locator*

Tick at every integer mapping of the string data.

Parameters

units_mapping

[dict] Mapping of category names (str) to indices (int).

tick_values(*self*, *vmin*, *vmax*)

Return the values of the located ticks given **vmin** and **vmax**.

Note: To get tick locations with the vmin and vmax values defined automatically for the associated `axis` simply call the Locator instance:

```
>>> print(type(loc))
<type 'Locator'>
>>> print(loc())
[1, 2, 3, 4]
```

---

class matplotlib.category.**UnitData**(*data=None*)

Bases: object

Create mapping between unique categorical values and integer ids.

Parameters

data

[iterable] sequence of string values

update(*self*, *data*)

Map new values to integer identifiers.

Parameters

data

[iterable of str or bytes]

Raises

TypeError

If elements in *data* are neither str nor bytes.

18.14 matplotlib.cbook

A collection of utility functions and classes. Originally, many (but not all) were from the Python Cookbook -- hence the name cbook.

This module is safe to import from anywhere within Matplotlib; it imports Matplotlib only at runtime.

class matplotlib.cbook.CallbackRegistry(*exception_handler=<function _exception_printer>*)

Bases: object

Handle registering and disconnecting for a set of signals and callbacks:

```
>>> def oneat(x):
...     print('eat', x)
>>> def ondrink(x):
...     print('drink', x)
```

```
>>> from matplotlib.cbook import CallbackRegistry
>>> callbacks = CallbackRegistry()
```

```
>>> id_eat = callbacks.connect('eat', oneat)
>>> id_drink = callbacks.connect('drink', ondrink)
```

```
>>> callbacks.process('drink', 123)
drink 123
>>> callbacks.process('eat', 456)
eat 456
>>> callbacks.process('be merry', 456) # nothing will be called
>>> callbacks.disconnect(id_eat)
>>> callbacks.process('eat', 456)      # nothing will be called
```

In practice, one should always disconnect all callbacks when they are no longer needed to avoid dangling references (and thus memory leaks). However, real

code in Matplotlib rarely does so, and due to its design, it is rather difficult to place this kind of code. To get around this, and prevent this class of memory leaks, we instead store weak references to bound methods only, so when the destination object needs to die, the CallbackRegistry won't keep it alive.

Parameters

exception_handler

[callable, optional] If provided must have signature

```
def handler(exc: Exception) -> None:
```

If not None this function will be called with any `Exception` subclass raised by the callbacks in *CallbackRegistry.process*. The handler may either consume the exception or re-raise.

The callable must be pickle-able.

The default handler is

```
def h(exc):
    traceback.print_exc()
```

`connect`(*self*, *s*, *func*)
Register *func* to be called when signal *s* is generated.

`disconnect`(*self*, *cid*)
Disconnect the callback registered with callback id *cid*.

`process`(*self*, *s*, **args*, ***kwargs*)
Process signal *s*.

All of the functions registered to receive callbacks on *s* will be called with `*args` and `**kwargs`.

`class matplotlib.cbook.Grouper`(*init=()*)
Bases: `object`

A disjoint-set data structure.

Objects can be joined using *join()*, tested for connectedness using *joined()*, and all disjoint sets can be retrieved by using the object as an iterator.

The objects being joined must be hashable and weak-referenceable.

Examples

```
>>> from matplotlib.cbook import Grouper
>>> class Foo:
...     def __init__(self, s):
...         self.s = s
...     def __repr__(self):
...         return self.s
...
>>> a, b, c, d, e, f = [Foo(x) for x in 'abcdef']
>>> grp = Grouper()
>>> grp.join(a, b)
>>> grp.join(b, c)
>>> grp.join(d, e)
>>> sorted(map(tuple, grp))
[(a, b, c), (d, e)]
>>> grp.joined(a, b)
True
>>> grp.joined(a, c)
True
>>> grp.joined(a, d)
False
```

clean(*self*)
Clean dead weak references from the dictionary.

get_siblings(*self*, *a*)
Return all of the items joined with *a*, including itself.

join(*self*, *a*, **args*)
Join given arguments into the same set. Accepts one or more arguments.

joined(*self*, *a*, *b*)
Return whether *a* and *b* are members of the same set.

remove(*self*, *a*)

exception matplotlib.cbook.**IgnoredKeywordWarning**(**args*, ***kwargs*)
Bases: UserWarning

[*Deprecated*] A class for issuing warnings about keyword arguments that will be ignored by Matplotlib.

Notes

Deprecated since version 3.3.

class matplotlib.cbook.**Stack**(*default=None*)
Bases: object

Stack of elements with a movable cursor.

Mimics home/back/forward in a web browser.

`back`(*self*)
Move the position back and return the current element.

`bubble`(*self*, *o*)
Raise all references of *o* to the top of the stack, and return it.

Raises

ValueError

If *o* is not in the stack.

`clear`(*self*)
Empty the stack.

`empty`(*self*)
Return whether the stack is empty.

`forward`(*self*)
Move the position forward and return the current element.

`home`(*self*)
Push the first element onto the top of the stack.

The first element is returned.

`push`(*self*, *o*)
Push *o* to the stack at current position. Discard all later elements.

o is returned.

`remove`(*self*, *o*)
Remove *o* from the stack.

Raises

ValueError

If *o* is not in the stack.

`matplotlib.cbook.boxplot_stats`(*X*, *whis=1.5*, *bootstrap=None*, *labels=None*, *autorange=False*)
Return a list of dictionaries of statistics used to draw a series of box and whisker plots using *bxp*.

Parameters

X

[array-like] Data that will be represented in the boxplots. Should have 2 or fewer dimensions.

whis

[float or (float, float), default: 1.5] The position of the whiskers.

If a float, the lower whisker is at the lowest datum above `Q1 - whis*(Q3-Q1)`, and the upper whisker at the highest datum below `Q3 + whis*(Q3-Q1)`, where Q1 and Q3 are the first and third quartiles. The default value of `whis = 1.5` corresponds to Tukey's original definition of boxplots.

If a pair of floats, they indicate the percentiles at which to draw the whiskers (e.g., (5, 95)). In particular, setting this to (0, 100) results in whiskers covering the whole range of the data. "range" is a deprecated synonym for (0, 100).

In the edge case where `Q1 == Q3`, *whis* is automatically set to (0, 100) (cover the whole range of the data) if *autorange* is True.

Beyond the whiskers, data are considered outliers and are plotted as individual points.

bootstrap

[int, optional] Number of times the confidence intervals around the median should be bootstrapped (percentile method).

labels

[array-like, optional] Labels for each dataset. Length must be compatible with dimensions of *X*.

autorange

[bool, optional (False)] When `True` and the data are distributed such that the 25th and 75th percentiles are equal, `whis` is set to (0, 100) such that the whisker ends are at the minimum and maximum of the data.

Returns

list of dict

A list of dictionaries containing the results for each column of data. Keys of each dictionary are the following:

Key	Value Description
label	tick label for the boxplot
mean	arithmetic mean value
med	50th percentile
q1	first quartile (25th percentile)
q3	third quartile (75th percentile)
cilo	lower notch around the median
cihi	upper notch around the median
whislo	end of the lower whisker
whishi	end of the upper whisker
fliers	outliers

Notes

Non-bootstrapping approach to confidence interval uses Gaussian-based asymptotic approximation:

$$\mathrm{med} \pm 1.57 \times \frac{\mathrm{iqr}}{\sqrt{N}}$$

General approach from: McGill, R., Tukey, J.W., and Larsen, W.A. (1978) "Variations of Boxplots", The American Statistician, 32:12-16.

`matplotlib.cbook.`**`contiguous_regions`**(*mask*)

Return a list of (ind0, ind1) such that `mask[ind0:ind1].all()` is True and we cover all such regions.

`matplotlib.cbook.`**`delete_masked_points`**(**args*)

Find all masked and/or non-finite points in a set of arguments, and return the arguments with only the unmasked points remaining.

Arguments can be in any of 5 categories:

1) 1-D masked arrays
2) 1-D ndarrays
3) ndarrays with more than one dimension
4) other non-string iterables
5) anything else

The first argument must be in one of the first four categories; any argument with a length differing from that of the first argument (and hence anything in category 5) then will be passed through unchanged.

Masks are obtained from all arguments of the correct length in categories 1, 2, and 4; a point is bad if masked in a masked array or if it is a nan or inf. No attempt is made to extract a mask from categories 2, 3, and 4 if `numpy.isfinite` does not yield a Boolean array.

All input arguments that are not passed unchanged are returned as ndarrays after removing the points or rows corresponding to masks in any of the arguments.

A vastly simpler version of this function was originally written as a helper for Axes.scatter().

`matplotlib.cbook.`**`file_requires_unicode`**(*x*)

Return whether the given writable file-like object requires Unicode to be written to it.

`matplotlib.cbook.`**`flatten`**(*seq*, *scalarp=<function is_scalar_or_string at 0x7f543929d1f0>*)

Return a generator of flattened nested containers.

For example:

```
>>> from matplotlib.cbook import flatten
>>> l = (('John', ['Hunter']), (1, 23), [[([42, (5, 23)], )]])
>>> print(list(flatten(l)))
['John', 'Hunter', 1, 23, 42, 5, 23]
```

By: Composite of Holger Krekel and Luther Blissett From: https://code.activestate.com/recipes/121294/ and Recipe 1.12 in cookbook

matplotlib.cbook.**get_realpath_and_stat**(*path*)

[*Deprecated*]

Notes

Deprecated since version 3.3:

matplotlib.cbook.**get_sample_data**(*fname*, *asfileobj=True*, ***, *np_load=False*)

Return a sample data file. *fname* is a path relative to the `mpl-data/sample_data` directory. If *asfileobj* is `True` return a file object, otherwise just a file path.

Sample data files are stored in the 'mpl-data/sample_data' directory within the Matplotlib package.

If the filename ends in .gz, the file is implicitly ungzipped. If the filename ends with .npy or .npz, *asfileobj* is True, and *np_load* is True, the file is loaded with `numpy.load`. *np_load* currently defaults to False but will default to True in a future release.

matplotlib.cbook.**index_of**(*y*)

A helper function to create reasonable x values for the given *y*.

This is used for plotting (x, y) if x values are not explicitly given.

First try `y.index` (assuming *y* is a `pandas.Series`), if that fails, use `range(len(y))`.

This will be extended in the future to deal with more types of labeled data.

Parameters

y

[float or array-like]

Returns

x, y

[ndarray] The x and y values to plot.

matplotlib.cbook.**is_math_text**(*s*)

Return whether the string *s* contains math expressions.

This is done by checking whether *s* contains an even number of non-escaped dollar signs.

matplotlib.cbook.**is_scalar_or_string**(*val*)
Return whether the given object is a scalar or string like.

matplotlib.cbook.**is_writable_file_like**(*obj*)
Return whether *obj* looks like a file object with a *write* method.

matplotlib.cbook.**local_over_kwdict**(*local_var*, *kwargs*, **keys*)
[*Deprecated*] Enforces the priority of a local variable over potentially conflicting argument(s) from a kwargs dict. The following possible output values are considered in order of priority:

```
local_var > kwargs[keys[0]] > ... > kwargs[keys[-1]]
```

The first of these whose value is not None will be returned. If all are None then None will be returned. Each key in keys will be removed from the kwargs dict in place.

Parameters

local_var

[any object] The local variable (highest priority).

kwargs

[dict] Dictionary of keyword arguments; modified in place.

keys

[str(s)] Name(s) of keyword arguments to process, in descending order of priority.

Returns

any object

Either local_var or one of kwargs[key] for key in keys.

Raises

IgnoredKeywordWarning

For each key in keys that is removed from kwargs but not used as the output value.

Notes

Deprecated since version 3.3.

class matplotlib.cbook.**maxdict**(*maxsize*)

Bases: dict

A dictionary with a maximum size.

Notes

This doesn't override all the relevant methods to constrain the size, just `__setitem__`, so use with caution.

matplotlib.cbook.**normalize_kwargs**(*kw*, *alias_mapping=None*, *required=<deprecated parameter>*, *forbidden=<deprecated parameter>*, *allowed=<deprecated parameter>*)

Helper function to normalize kwarg inputs.

The order they are resolved are:

1. aliasing
2. required
3. forbidden
4. allowed

This order means that only the canonical names need appear in *allowed*, *forbidden*, *required*.

Parameters

kw

[dict] A dict of keyword arguments.

alias_mapping

[dict or Artist subclass or Artist instance, optional] A mapping between a canonical name to a list of aliases, in order of precedence from lowest to highest.

If the canonical value is not in the list it is assumed to have the highest priority.

If an Artist subclass or instance is passed, use its properties alias mapping.

required

[list of str, optional] A list of keys that must be in *kws*. This parameter is deprecated.

forbidden

[list of str, optional] A list of keys which may not be in *kw*. This parameter is deprecated.

allowed

[list of str, optional] A list of allowed fields. If this not None, then raise if *kw* contains any keys not in the union of *required* and *allowed*. To allow only the required fields pass in an empty tuple `allowed=()`. This parameter is deprecated.

Raises

TypeError

To match what python raises if invalid args/kwargs are passed to a callable.

`matplotlib.cbook.`**open_file_cm**(*path_or_file*, *mode='r'*, *encoding=None*)

Pass through file objects and context-manage path-likes.

`matplotlib.cbook.`**print_cycles**(*objects*, *outstream=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*, *show_progress=False*)

Print loops of cyclic references in the given *objects*.

It is often useful to pass in `gc.garbage` to find the cycles that are preventing some objects from being garbage collected.

Parameters

objects

A list of objects to find cycles in.

outstream

The stream for output.

show_progress

[bool] If True, print the number of objects reached as they are found.

`matplotlib.cbook.`**pts_to_midstep**(*x*, **args*)

Convert continuous line to mid-steps.

Given a set of `N` points convert to `2N` points which when connected linearly give a step function which changes values at the middle of the intervals.

Parameters

x

[array] The x location of the steps. May be empty.

y1, ..., yp

[array] y arrays to be turned into steps; all must be the same length as `x`.

Returns

array

The x and y values converted to steps in the same order as the input; can be unpacked as `x_out, y1_out, ..., yp_out`. If the input is length `N`, each of these arrays will be length `2N`.

Examples

```
>>> x_s, y1_s, y2_s = pts_to_midstep(x, y1, y2)
```

matplotlib.cbook.**pts_to_poststep**(*x*, **args*)

Convert continuous line to post-steps.

Given a set of `N` points convert to `2N + 1` points, which when connected linearly give a step function which changes values at the end of the intervals.

Parameters

x

[array] The x location of the steps. May be empty.

y1, ..., yp

[array] y arrays to be turned into steps; all must be the same length as `x`.

Returns

array

The x and y values converted to steps in the same order as the input; can be unpacked as `x_out, y1_out, ..., yp_out`. If the input is length `N`, each of these arrays will be length `2N + 1`. For `N=0`, the length will be 0.

Examples

```
>>> x_s, y1_s, y2_s = pts_to_poststep(x, y1, y2)
```

matplotlib.cbook.**pts_to_prestep**(*x*, **args*)

Convert continuous line to pre-steps.

Given a set of `N` points, convert to `2N - 1` points, which when connected linearly give a step function which changes values at the beginning of the intervals.

Parameters

x

[array] The x location of the steps. May be empty.

y1, ..., yp

[array] y arrays to be turned into steps; all must be the same length as `x`.

Returns

array

The x and y values converted to steps in the same order as the input; can be unpacked as `x_out, y1_out, ..., yp_out`. If the input is length `N`, each of these arrays will be length `2N + 1`. For `N=0`, the length will be 0.

Examples

```
>>> x_s, y1_s, y2_s = pts_to_prestep(x, y1, y2)
```

matplotlib.cbook.**report_memory**(*i=0*)

Return the memory consumed by the process.

matplotlib.cbook.**safe_first_element**(*obj*)

Return the first element in *obj*.

This is an type-independent way of obtaining the first element, supporting both index access and the iterator protocol.

matplotlib.cbook.**safe_masked_invalid**(*x*, *copy=False*)

matplotlib.cbook.**sanitize_sequence**(*data*)

Convert dictview objects to list. Other inputs are returned unchanged.

class matplotlib.cbook.**silent_list**(*type*, *seq=None*)

Bases: `list`

A list with a short `repr()`.

This is meant to be used for a homogeneous list of artists, so that they don't cause long, meaningless output.

Instead of

```
[<matplotlib.lines.Line2D object at 0x7f5749fed3c8>,
 <matplotlib.lines.Line2D object at 0x7f5749fed4e0>,
 <matplotlib.lines.Line2D object at 0x7f5758016550>]
```

one will get

```
<a list of 3 Line2D objects>
```

matplotlib.cbook.**simple_linear_interpolation**(*a*, *steps*)

Resample an array with `steps - 1` points between original point pairs.

Along each column of *a*, `(steps - 1)` points are introduced between each original values; the values are linearly interpolated.

Parameters

a

[array, shape (n, ...)]

steps

[int]

Returns

array

shape `((n - 1) * steps + 1, ...)`

matplotlib.cbook.**strip_math**(*s*)

Remove latex formatting from mathtext.

Only handles fully math and fully non-math strings.

matplotlib.cbook.**to_filehandle**(*fname*, *flag='r'*, *return_opened=False*, *encoding=None*)

Convert a path to an open file handle or pass-through a file-like object.

Consider using *open_file_cm* instead, as it allows one to properly close newly created file objects more easily.

Parameters

fname

[str or path-like or file-like] If `str` or `os.PathLike`, the file is opened using the flags specified by *flag* and *encoding*. If a file-like object, it is passed through.

flag

[str, default 'r'] Passed as the *mode* argument to `open` when *fname* is `str` or `os.PathLike`; ignored if *fname* is file-like.

return_opened

[bool, default False] If True, return both the file object and a boolean indicating whether this was a new file (that the caller needs to close). If False, return only the new file.

encoding

[str or None, default None] Passed as the *mode* argument to `open` when *fname* is `str` or `os.PathLike`; ignored if *fname* is file-like.

Returns

fh

[file-like]

opened

[bool] *opened* is only returned if *return_opened* is True.

`matplotlib.cbook.violin_stats`(*X*, *method*, *points=100*, *quantiles=None*)

Return a list of dictionaries of data which can be used to draw a series of violin plots.

See the `Returns` section below to view the required keys of the dictionary.

Users can skip this function and pass a user-defined set of dictionaries with the same keys to *violinplot* instead of using Matplotlib to do the calculations. See the *Returns* section below for the keys that must be present in the dictionaries.

Parameters

X

[array-like] Sample data that will be used to produce the gaussian kernel density estimates. Must have 2 or fewer dimensions.

method

[callable] The method used to calculate the kernel density estimate for each column of data. When called via `method(v, coords)`, it should return a vector of the values of the KDE evaluated at the values specified in coords.

points

[int, default: 100] Defines the number of points to evaluate each of the gaussian kernel density estimates at.

quantiles

[array-like, default: None] Defines (if not None) a list of floats in interval [0, 1] for each column of data, which represents the quantiles that will be rendered for that column of data. Must have 2 or fewer dimensions. 1D array will be treated as a singleton list containing them.

Returns

list of dict

A list of dictionaries containing the results for each column of data. The dictionaries contain at least the following:

- coords: A list of scalars containing the coordinates this particular kernel density estimate was evaluated at.
- vals: A list of scalars containing the values of the kernel density estimate at each of the coordinates given in *coords*.
- mean: The mean value for this column of data.
- median: The median value for this column of data.
- min: The minimum value for this column of data.
- max: The maximum value for this column of data.
- quantiles: The quantile values for this column of data.

exception matplotlib.cbook.deprecation.**MatplotlibDeprecationWarning**

Bases: UserWarning

A class for issuing deprecation warnings for Matplotlib users.

In light of the fact that Python builtin DeprecationWarnings are ignored by default as of Python 2.7 (see link below), this class was put in to allow for the signaling of deprecation, but via UserWarnings which are not ignored by default.

https://docs.python.org/dev/whatsnew/2.7.html#the-future-for-python-2-x

matplotlib.cbook.deprecation.**deprecated**(*since*, *, *message=''*, *name=''*, *alternative=''*, *pending=False*, *obj_type=None*, *addendum=''*, *removal=''*)

Decorator to mark a function, a class, or a property as deprecated.

When deprecating a classmethod, a staticmethod, or a property, the `@deprecated` decorator should go *under* `@classmethod` and `@staticmethod` (i.e., *deprecated* should directly decorate the underlying callable), but *over* `@property`.

Parameters

since

[str] The release at which this API became deprecated.

message

[str, optional] Override the default deprecation message. The `%(since)s`, `%(name)s`, `%(alternative)s`, `%(obj_type)s`, `%(addendum)s`, and `%(removal)s` format specifiers will be replaced by the values of the respective arguments passed to this function.

name

[str, optional] The name used in the deprecation message; if not provided, the name is automatically determined from the deprecated object.

alternative

[str, optional] An alternative API that the user may use in place of the deprecated API. The deprecation warning will tell the user about this alternative if provided.

pending

[bool, optional] If True, uses a PendingDeprecationWarning instead of a DeprecationWarning. Cannot be used together with *removal*.

obj_type

[str, optional] The object type being deprecated; by default, 'class' if decorating a class, 'attribute' if decorating a property, 'function' otherwise.

addendum

[str, optional] Additional text appended directly to the final message.

removal

[str, optional] The expected removal version. With the default (an empty string), a removal version is automatically computed from *since*. Set to other Falsy values to not schedule a removal date. Cannot be used together with *pending*.

Examples

Basic example:

```
@deprecated('1.4.0')
def the_function_to_deprecate():
    pass
```

matplotlib.cbook.deprecation.**mplDeprecation**

alias of *matplotlib.cbook.deprecation.MatplotlibDeprecationWarning*

matplotlib.cbook.deprecation.**warn_deprecated**(*since*, ***, *message=''*, *name=''*, *alternative=''*, *pending=False*, *obj_type=''*, *addendum=''*, *removal=''*)

Display a standardized deprecation.

Parameters

since

[str] The release at which this API became deprecated.

message

[str, optional] Override the default deprecation message. The `%(since)s`, `%(name)s`, `%(alternative)s`, `%(obj_type)s`, `%(addendum)s`, and `%(removal)s` format specifiers will be replaced by the values of the respective arguments passed to this function.

name

[str, optional] The name of the deprecated object.

alternative

[str, optional] An alternative API that the user may use in place of the deprecated API. The deprecation warning will tell the user about this alternative if provided.

pending

[bool, optional] If True, uses a PendingDeprecationWarning instead of a DeprecationWarning. Cannot be used together with *removal*.

obj_type

[str, optional] The object type being deprecated.

addendum

[str, optional] Additional text appended directly to the final message.

removal

[str, optional] The expected removal version. With the default (an empty string), a removal version is automatically computed from *since*. Set to other Falsy values to not schedule a removal date. Cannot be used together with *pending*.

Examples

Basic example:

```
# To warn of the deprecation of "matplotlib.name_of_module"
warn_deprecated('1.4.0', name='matplotlib.name_of_module',
                obj_type='module')
```

18.15 matplotlib.cm

Builtin colormaps, colormap handling utilities, and the *ScalarMappable* mixin.

See also:

/gallery/color/colormap_reference for a list of builtin colormaps.

Creating Colormaps in Matplotlib for examples of how to make colormaps.

Choosing Colormaps in Matplotlib an in-depth discussion of choosing colormaps.

Colormap Normalization for more details about data normalization.

class matplotlib.cm.**ScalarMappable**(*norm=None*, *cmap=None*)

Bases: object

A mixin class to map scalar data to RGBA.

The ScalarMappable applies data normalization before returning RGBA colors from the given colormap.

Parameters

norm

[*matplotlib.colors.Normalize* (or subclass thereof)] The normalizing object which scales data, typically into the interval `[0, 1]`. If *None*, *norm* defaults to a *colors.Normalize* object which initializes its scaling based on the first data processed.

cmap

[str or *Colormap*] The colormap used to map normalized data values to RGBA colors.

`add_checker`(*self*, *checker*)
[*Deprecated*]

Notes

Deprecated since version 3.3:

`autoscale`(*self*)
Autoscale the scalar limits on the norm instance using the current array

`autoscale_None`(*self*)
Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

`changed`(*self*)
Call this whenever the mappable is changed to notify all the callbackSM listeners to the 'changed' signal.

`check_update`(*self*, *checker*)
[*Deprecated*]

Notes

Deprecated since version 3.3:

`colorbar`
The last colorbar associated with this ScalarMappable. May be None.

`get_alpha`(*self*)

Returns

float

Always returns 1.

`get_array`(*self*)
Return the data array.

`get_clim`(*self*)
Return the values (min, max) that are mapped to the colormap limits.

`get_cmap`(*self*)
Return the *Colormap* instance.

`set_array`(*self*, *A*)
Set the image array from numpy array *A*.

Parameters

A

[ndarray]

`set_clim`(*self*, *vmin=None*, *vmax=None*)
Set the norm limits for image scaling.

Parameters

vmin, vmax

[float] The limits.

The limits may also be passed as a tuple (*vmin*, *vmax*) as a single positional argument.

`set_cmap`(*self*, *cmap*)
Set the colormap for luminance data.

Parameters

cmap

[*Colormap* or str or None]

`set_norm`(*self*, *norm*)
Set the normalization instance.

Parameters

norm

[*Normalize* or None]

Notes

If there are any colorbars using the mappable for this norm, setting the norm of the mappable will reset the norm, locator, and formatters on the colorbar to default.

`to_rgba`(*self*, *x*, *alpha=None*, *bytes=False*, *norm=True*)
Return a normalized rgba array corresponding to *x*.

In the normal case, *x* is a 1-D or 2-D sequence of scalars, and the corresponding ndarray of rgba values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already rgb or rgba, such as might have been read from an image file. If *x* is an ndarray with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an rgb or rgba array, and no mapping will be done. The array can be uint8, or it can be floating point with values in the 0-1 range; otherwise a

ValueError will be raised. If it is a masked array, the mask will be ignored. If the last dimension is 3, the *alpha* kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the *alpha* kwarg is ignored; it does not replace the pre-existing alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if *bytes* is *False* (default), the rgba array will be floats in the 0-1 range; if it is *True*, the returned rgba array will be uint8 in the 0 to 255 range.

If norm is False, no normalization of the input data is performed, and it is assumed to be in the range (0-1).

property update_dict

matplotlib.cm.**get_cmap**(*name=None*, *lut=None*)

Get a colormap instance, defaulting to rc values if *name* is None.

Colormaps added with *register_cmap()* take precedence over built-in colormaps.

Parameters

name

[*matplotlib.colors.Colormap* or str or None, default: None] If a *Colormap* instance, it will be returned. Otherwise, the name of a colormap known to Matplotlib, which will be resampled by *lut*. The default, None, means `rcParams["image.cmap"]` (default: `'viridis'`).

lut

[int or None, default: None] If *name* is not already a Colormap instance and *lut* is not None, the colormap will be resampled to have *lut* entries in the lookup table.

Notes

Currently, this returns the global colormap object, which is deprecated. In Matplotlib 3.5, you will no longer be able to modify the global colormaps in-place.

matplotlib.cm.**register_cmap**(*name=None*, *cmap=None*, *data=None*, *lut=None*)

Add a colormap to the set recognized by *get_cmap()*.

It can be used in two ways:

```
register_cmap(name='swirly', cmap=swirly_cmap)

register_cmap(name='choppy', data=choppydata, lut=128)
```

In the first case, *cmap* must be a *matplotlib.colors.Colormap* instance. The *name* is optional; if absent, the name will be the `name` attribute of the *cmap*.

The second case is deprecated. Here, the three arguments are passed to the *LinearSegmentedColormap* initializer, and the resulting colormap is registered. Instead of this implicit colormap creation, create a *LinearSegmentedColormap* and use the first case: `register_cmap(cmap=LinearSegmentedColormap(name, data, lut))`.

Notes

Registering a colormap stores a reference to the colormap object which can currently be modified and inadvertantly change the global colormap state. This behavior is deprecated and in Matplotlib 3.5 the registered colormap will be immutable.

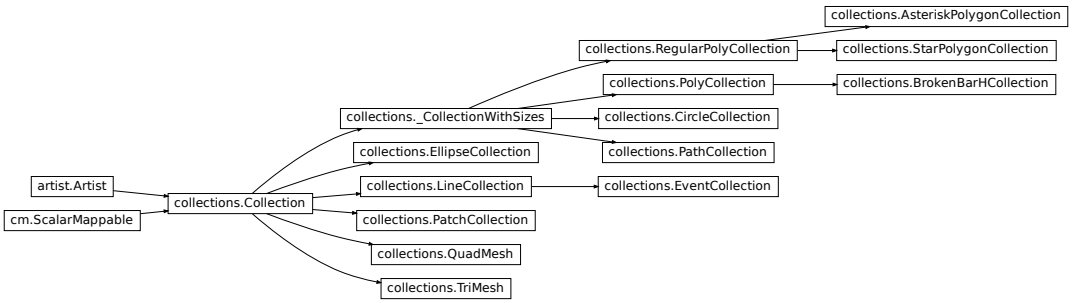
`matplotlib.cm.`**`revcmap`**(*data*)

[*Deprecated*] Can only handle specification *data* in dictionary format.

Notes

Deprecated since version 3.2.

18.16 matplotlib.collections



Classes for the efficient drawing of large collections of objects that share most properties, e.g., a large number of line segments or polygons.

The classes are not meant to be as flexible as their single element counterparts (e.g., you may not be able to select all line styles) but they are meant to be fast for common use cases (e.g., a large set of solid line segments).

class matplotlib.collections.**AsteriskPolygonCollection**(*numsides*, *rotation=0*, *sizes=1*, ***kwargs*)

Bases: *matplotlib.collections.RegularPolyCollection*

Draw a collection of regular asterisks with *numsides* points.

Parameters

numsides

[int] The number of sides of the polygon.

rotation

[float] The rotation of the polygon in radians.

sizes

[tuple of float] The area of the circle circumscribing the polygon in points^2.

****kwargs**

Forwarded to *Collection*.

Examples

See /gallery/event_handling/lasso_demo for a complete example:

```
offsets = np.random.rand(20, 2)
facecolors = [cm.jet(x) for x in np.random.rand(20)]

collection = RegularPolyCollection(
    numsides=5, # a pentagon
    rotation=0, sizes=(50,),
    facecolors=facecolors,
    edgecolors=("black",),
    linewidths=(1,),
    offsets=offsets,
    transOffset=ax.transData,
    )
```

`add_callback`(*self, func*)

Add a callback function that will be called whenever one of the *Artist*'s properties changes.

Parameters

func

[callable] The callback function. It must have the signature:

```
def func(artist: Artist) -> Any
```

where *artist* is the calling *Artist*. Return values may exist but are ignored.

Returns

int

The observer id associated with the callback. This id can be used for removing the callback with *remove_callback* later.

See also:

remove_callback

`add_checker`(*self*, *checker*)
[*Deprecated*]

Notes

Deprecated since version 3.3:

`autoscale`(*self*)
Autoscale the scalar limits on the norm instance using the current array

`autoscale_None`(*self*)
Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

`property axes`
The *Axes* instance the artist resides in, or *None*.

`changed`(*self*)
Call this whenever the mappable is changed to notify all the callbackSM listeners to the 'changed' signal.

`check_update`(*self*, *checker*)
[*Deprecated*]

Notes

Deprecated since version 3.3:

`contains`(*self*, *mouseevent*)
Test whether the mouse event occurred in the collection.

Returns `bool, dict(ind=itemlist)`, where every item in itemlist contains the event.

`convert_xunits`(*self*, *x*)
Convert *x* using the unit type of the xaxis.

If the artist is not in contained in an Axes or if the xaxis does not have units, *x* itself is returned.

`convert_yunits`(*self*, *y*)
Convert *y* using the unit type of the yaxis.

If the artist is not in contained in an Axes or if the yaxis does not have units, *y* itself is returned.

`draw`(*self*, *renderer*)
Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

Parameters

renderer

[*RendererBase* subclass.]

Notes

This method is overridden in the Artist subclasses.

`findobj`(*self*, *match=None*, *include_self=True*)
Find artist objects.

Recursively find all *Artist* instances contained in the artist.

Parameters

match

A filter criterion for the matches. This can be

- *None*: Return all objects contained in artist.
- A function with signature `def match(artist: Artist) -> bool`. The result will only contain artists for which the function returns *True*.
- A class instance: e.g., *Line2D*. The result will only contain artists of this class or its subclasses (`isinstance` check).

include_self

[bool] Include *self* in the list to be checked for a match.

Returns

list of *Artist*

`format_cursor_data`(*self*, *data*)
Return a string representation of *data*.

Note: This method is intended to be overridden by artist subclasses. As an end-user of Matplotlib you will most likely not call this method yourself.

The default implementation converts ints and floats and arrays of ints and floats into a comma-separated string enclosed in square brackets.

See also:

get_cursor_data

`get_agg_filter`(*self*)
Return filter function to be used for agg filter.

`get_alpha`(*self*)
Return the alpha value used for blending - not supported on all backends.

`get_animated`(*self*)
Return whether the artist is animated.

`get_array`(*self*)
Return the data array.

`get_capstyle`(*self*)

`get_children`(*self*)
Return a list of the child *Artist*s of this *Artist*.

`get_clim`(*self*)
Return the values (min, max) that are mapped to the colormap limits.

`get_clip_box`(*self*)
Return the clipbox.

`get_clip_on`(*self*)
Return whether the artist uses clipping.

`get_clip_path`(*self*)
Return the clip path.

`get_cmap`(*self*)
Return the *Colormap* instance.

`get_contains`(*self*)
[*Deprecated*] Return the custom contains function of the artist if set, or *None*.

See also:

set_contains

Notes

Deprecated since version 3.3.

get_cursor_data(*self*, *event*)
Return the cursor data for a given event.

Note: This method is intended to be overridden by artist subclasses. As an end-user of Matplotlib you will most likely not call this method yourself.

Cursor data can be used by Artists to provide additional context information for a given event. The default implementation just returns *None*.

Subclasses can override the method and return arbitrary data. However, when doing so, they must ensure that *format_cursor_data* can convert the data to a string representation.

The only current use case is displaying the z-value of an *AxesImage* in the status bar of a plot window, while moving the mouse.

Parameters

event

[*matplotlib.backend_bases.MouseEvent*]

See also:

format_cursor_data

get_dashes(*self*)
Alias for *get_linestyle*.

get_datalim(*self*, *transData*)

get_ec(*self*)
Alias for *get_edgecolor*.

get_edgecolor(*self*)

get_edgecolors(*self*)
Alias for *get_edgecolor*.

get_facecolor(*self*)

get_facecolors(*self*)
Alias for *get_facecolor*.

get_fc(*self*)
Alias for *get_facecolor*.

get_figure(*self*)
Return the *Figure* instance the artist belongs to.

`get_fill`(*self*)
Return whether fill is set.

`get_gid`(*self*)
Return the group id.

`get_hatch`(*self*)
Return the current hatching pattern.

`get_in_layout`(*self*)
Return boolean flag, `True` if artist is included in layout calculations.

E.g. *Constrained Layout Guide*, *Figure.tight_layout()*, and `fig.savefig(fname, bbox_inches='tight')`.

`get_joinstyle`(*self*)

`get_label`(*self*)
Return the label used for this artist in the legend.

`get_linestyle`(*self*)

`get_linestyles`(*self*)
Alias for *get_linestyle*.

`get_linewidth`(*self*)

`get_linewidths`(*self*)
Alias for *get_linewidth*.

`get_ls`(*self*)
Alias for *get_linestyle*.

`get_lw`(*self*)
Alias for *get_linewidth*.

`get_numsides`(*self*)

`get_offset_position`(*self*)
[*Deprecated*] Return how offsets are applied for the collection. If *offset_position* is 'screen', the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is 'data', the offset is applied before the master transform, i.e., the offsets are in data coordinates.

Notes

Deprecated since version 3.3.

`get_offset_transform`(*self*)

`get_offsets`(*self*)
Return the offsets for the collection.

`get_path_effects`(*self*)

get_paths(*self*)

get_picker(*self*)
Return the picking behavior of the artist.

The possible values are described in *set_picker*.

See also:

set_picker, *pickable*, *pick*

get_pickradius(*self*)

get_rasterized(*self*)
Return whether the artist is to be rasterized.

get_rotation(*self*)

get_sizes(*self*)
Return the sizes ('areas') of the elements in the collection.

Returns

array

The 'area' of each element.

get_sketch_params(*self*)
Return the sketch parameters for the artist.

Returns

tuple or None

A 3-tuple with the following elements:

- *scale*: The amplitude of the wiggle perpendicular to the source line.
- *length*: The length of the wiggle along the line.
- *randomness*: The scale factor by which the length is shrunken or expanded.

Returns *None* if no sketch parameters were set.

get_snap(*self*)
Return the snap setting.

See *set_snap* for details.

get_tightbbox(*self*, *renderer*)
Like *Artist.get_window_extent*, but includes any clipping.

Parameters

renderer

[*RendererBase* subclass] renderer that will be used to draw the figures (i.e. `fig.canvas.get_renderer()`)

Returns

Bbox

The enclosing bounding box (in figure pixel coordinates).

`get_transform`(*self*)
Return the *Transform* instance used by this artist.

`get_transformed_clip_path_and_affine`(*self*)
Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

`get_transforms`(*self*)

`get_url`(*self*)
Return the url.

`get_urls`(*self*)
Return a list of URLs, one for each element of the collection.

The list contains *None* for elements without a URL. See /gallery/misc/hyperlinks_sgskip for an example.

`get_visible`(*self*)
Return the visibility.

`get_window_extent`(*self*, *renderer*)
Get the axes bounding box in display space.

The bounding box' width and height are nonnegative.

Subclasses should override for inclusion in the bounding box "tight" calculation. Default is to return an empty bounding box at 0, 0.

Be careful when using this function, the results will not update if the artist window extent of the artist changes. The extent can change due to any changes in the transform stack, such as changing the axes limits, the figure size, or the canvas used (as is done when saving a figure). This can lead to unexpected behavior where interactive figures will look fine on the screen, but will save incorrectly.

`get_zorder`(*self*)
Return the artist's zorder.

`have_units`(*self*)
Return *True* if units are set on any axis.

`is_transform_set`(*self*)
Return whether the Artist has an explicitly set transform.

This is *True* after *set_transform* has been called.

property mouseover
: If this property is set to *True*, the artist will be queried for custom context information when the mouse cursor moves over it.

  See also *get_cursor_data()*, *ToolCursorPosition* and *NavigationToolbar2*.

pchanged(*self*)
: Call all of the registered callbacks.

  This function is triggered internally when a property is changed.

  See also:

  add_callback

  remove_callback

pick(*self, mouseevent*)
: Process a pick event.

  Each child artist will fire a pick event if *mouseevent* is over the artist and the artist has picker set.

  See also:

  set_picker, *get_picker*, *pickable*

pickable(*self*)
: Return whether the artist is pickable.

  See also:

  set_picker, *get_picker*, *pick*

properties(*self*)
: Return a dictionary of all the properties of the artist.

remove(*self*)
: Remove the artist from the figure if possible.

  The effect will not be visible until the figure is redrawn, e.g., with *FigureCanvasBase.draw_idle*. Call *relim* to update the axes limits if desired.

  Note: *relim* will not see collections even if the collection was added to the axes with *autolim* = True.

  Note: there is no support for removing the artist's legend entry.

remove_callback(*self, oid*)
: Remove a callback based on its observer id.

  See also:

add_callback

set(*self*, ***kwargs*)
A property batch setter. Pass *kwargs* to set properties.

set_aa(*self*, *aa*)
Alias for *set_antialiased*.

set_agg_filter(*self*, *filter_func*)
Set the agg filter.

Parameters

filter_func

[callable] A filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array.

set_alpha(*self*, *alpha*)
Set the alpha value used for blending - not supported on all backends.

Parameters

alpha

[float or None]

set_animated(*self*, *b*)
Set the artist's animation state.

Parameters

b

[bool]

set_antialiased(*self*, *aa*)
Set the antialiasing state for rendering.

Parameters

aa

[bool or list of bools]

set_antialiaseds(*self*, *aa*)
Alias for *set_antialiased*.

set_array(*self*, *A*)
Set the image array from numpy array *A*.

Parameters

A

[ndarray]

set_capstyle(*self*, *cs*)
Set the capstyle for the collection (for all its elements).

Parameters

cs

[{'butt', 'round', 'projecting'}] The capstyle.

set_clim(*self*, *vmin=None*, *vmax=None*)
Set the norm limits for image scaling.

Parameters

vmin, vmax

[float] The limits.

The limits may also be passed as a tuple (*vmin*, *vmax*) as a single positional argument.

set_clip_box(*self*, *clipbox*)
Set the artist's clip *Bbox*.

Parameters

clipbox

[*Bbox*]

set_clip_on(*self*, *b*)
Set whether the artist uses clipping.

When False artists will be visible outside of the axes which can lead to unexpected results.

Parameters

b

[bool]

set_clip_path(*self*, *path*, *transform=None*)
Set the artist's clip path.

Parameters

path

[*Patch* or *Path* or *TransformedPath* or None] The clip path. If given a *Path*, *transform* must be provided as well. If *None*, a previously set clip path is removed.

transform

[*Transform*, optional] Only used if *path* is a *Path*, in which case the given *Path* is converted to a *TransformedPath* using *transform*.

Notes

For efficiency, if *path* is a *Rectangle* this method will set the clipping box to the corresponding rectangle and set the clipping path to `None`.

For technical reasons (support of *set*), a tuple (*path*, *transform*) is also accepted as a single positional parameter.

`set_cmap`(*self*, *cmap*)
Set the colormap for luminance data.

Parameters

cmap

[*Colormap* or str or None]

`set_color`(*self*, *c*)
Set both the edgecolor and the facecolor.

Parameters

c

[color or list of rgba tuples]

See also:

Collection.set_facecolor, *Collection.set_edgecolor*

For setting the edge or face color individually.

`set_contains`(*self*, *picker*)
[*Deprecated*] Define a custom contains test for the artist.

The provided callable replaces the default *contains* method of the artist.

Parameters

picker

[callable] A custom picker function to evaluate if an event is within the artist. The function must have the signature:

```
def contains(artist: Artist, event: MouseEvent) -> bool, dict
```

that returns:

- a bool indicating if the event is within the artist
- a dict of additional information. The dict should at least return the same information as the default `contains()` implementation of the respective artist, but may provide additional information.

Notes

Deprecated since version 3.3.

`set_dashes`(*self*, *ls*)
Alias for *set_linestyle*.

`set_ec`(*self*, *c*)
Alias for *set_edgecolor*.

`set_edgecolor`(*self*, *c*)
Set the edgecolor(s) of the collection.

Parameters

c

[color or list of colors or 'face'] The collection edgecolor(s). If a sequence, the patches cycle through it. If 'face', match the facecolor.

`set_edgecolors`(*self*, *c*)
Alias for *set_edgecolor*.

`set_facecolor`(*self*, *c*)
Set the facecolor(s) of the collection. *c* can be a color (all patches have same color), or a sequence of colors; if it is a sequence the patches will cycle through the sequence.

If *c* is 'none', the patch will not be filled.

Parameters

c

[color or list of colors]

`set_facecolors`(*self*, *c*)
Alias for *set_facecolor*.

set_fc(*self*, *c*)
Alias for *set_facecolor*.

set_figure(*self*, *fig*)
Set the *Figure* instance the artist belongs to.

Parameters

fig

[*Figure*]

set_gid(*self*, *gid*)
Set the (group) id for the artist.

Parameters

gid

[str]

set_hatch(*self*, *hatch*)
Set the hatching pattern

hatch can be one of:

```
/   - diagonal hatching
\   - back diagonal
|   - vertical
-   - horizontal
+   - crossed
x   - crossed diagonal
o   - small circle
O   - large circle
.   - dots
*   - stars
```

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Unlike other properties such as linewidth and colors, hatching can only be specified for the collection as a whole, not separately for each member.

Parameters

hatch

[{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}]

set_in_layout(*self*, *in_layout*)
Set if artist is to be included in layout calculations, E.g. *Constrained Layout Guide*, *Figure.tight_layout()*, and `fig.savefig(fname, bbox_inches='tight')`.

Parameters

in_layout

[bool]

`set_joinstyle`(*self*, *js*)
Set the joinstyle for the collection (for all its elements).

Parameters

js

[{'miter', 'round', 'bevel'}] The joinstyle.

`set_label`(*self*, *s*)
Set a label that will be displayed in the legend.

Parameters

s

[object] *s* will be converted to a string by calling `str`.

`set_linestyle`(*self*, *ls*)
Set the linestyle(s) for the collection.

linestyle	description
`'-'` or `'solid'`	solid line
`'--'` or `'dashed'`	dashed line
`'-.'` or `'dashdot'`	dash-dotted line
`':'` or `'dotted'`	dotted line

Alternatively a dash tuple of the following form can be provided:

```
(offset, onoffseq),
```

where `onoffseq` is an even length tuple of on and off ink in points.

Parameters

ls

[str or tuple or list thereof] Valid values for individual linestyles include {'-', '--', '-.', ':', '', (offset, on-off-seq)}. See *Line2D.set_linestyle* for a complete description.

`set_linestyles`(*self*, *ls*)
Alias for *set_linestyle*.

`set_linewidth`(*self*, *lw*)
Set the linewidth(s) for the collection. *lw* can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence

Parameters

lw

[float or list of floats]

`set_linewidths`(*self*, *lw*)
Alias for *set_linewidth*.

`set_ls`(*self*, *ls*)
Alias for *set_linestyle*.

`set_lw`(*self*, *lw*)
Alias for *set_linewidth*.

`set_norm`(*self*, *norm*)
Set the normalization instance.

Parameters

norm

[*Normalize* or None]

Notes

If there are any colorbars using the mappable for this norm, setting the norm of the mappable will reset the norm, locator, and formatters on the colorbar to default.

`set_offset_position`(*self*, *offset_position*)
[*Deprecated*] Set how offsets are applied. If *offset_position* is 'screen' (default) the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is 'data', the offset is applied before the master transform, i.e., the offsets are in data coordinates.

Parameters

offset_position

[{'screen', 'data'}]

Notes

Deprecated since version 3.3.

`set_offsets`(*self*, *offsets*)
Set the offsets for the collection.

Parameters

offsets

[array-like (N, 2) or (2,)]

`set_path_effects`(*self*, *path_effects*)
Set the path effects.

Parameters

path_effects

[*AbstractPathEffect*]

`set_paths`(*self*)

`set_picker`(*self*, *picker*)
Define the picking behavior of the artist.

Parameters

picker

[None or bool or callable] This can be one of the following:

- *None*: Picking is disabled for this artist (default).
- A boolean: If *True* then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist.
- A function: If picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

to determine the hit test. if the mouse event is over the artist, return *hit=True* and props is a dictionary of properties you want added to the PickEvent attributes.

- *deprecated*: For *Line2D* only, *picker* can also be a float that sets the tolerance for checking whether an event occurred "on" the line; this is deprecated. Use *Line2D.set_pickradius* instead.

`set_pickradius`(*self*, *pr*)
Set the pick radius used for containment tests.

Parameters

d

[float] Pick radius, in points.

`set_rasterized`(*self*, *rasterized*)
Force rasterized (bitmap) drawing in vector backend output.

Defaults to None, which implies the backend's default behavior.

Parameters

rasterized

[bool or None]

`set_sizes`(*self*, *sizes*, *dpi=72.0*)
Set the sizes of each member of the collection.

Parameters

sizes

[ndarray or None] The size to set for each element of the collection. The value is the 'area' of the element.

dpi

[float, default: 72] The dpi of the canvas.

`set_sketch_params`(*self*, *scale=None*, *length=None*, *randomness=None*)
Sets the sketch parameters.

Parameters

scale

[float, optional] The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is `None`, or not provided, no sketch filter will be provided.

length

[float, optional] The length of the wiggle along the line, in pixels (default 128.0)

randomness

[float, optional] The scale factor by which the length is shrunken or expanded (default 16.0)

set_snap(*self*, *snap*)
Set the snapping behavior.

Snapping aligns positions with the pixel grid, which results in clearer images. For example, if a black line of 1px width was defined at a position in between two pixels, the resulting image would contain the interpolated value of that line in the pixel grid, which would be a grey value on both adjacent pixel positions. In contrast, snapping will move the line to the nearest integer pixel value, so that the resulting image will really contain a 1px wide black line.

Snapping is currently only supported by the Agg and MacOSX backends.

Parameters

snap

[bool or None] Possible values:

- *True*: Snap vertices to the nearest pixel center.
- *False*: Do not modify vertex positions.
- *None*: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center.

set_transform(*self*, *t*)
Set the artist transform.

Parameters

t

[*Transform*]

set_url(*self*, *url*)
Set the url for the artist.

Parameters

url

[str]

set_urls(*self*, *urls*)

Parameters

urls

[list of str or None]

Notes

URLs are currently only implemented by the SVG backend. They are ignored by all other backends.

`set_visible`(*self*, *b*)
Set the artist's visibility.

Parameters

b

[bool]

`set_zorder`(*self*, *level*)
Set the zorder for the artist. Artists with lower zorder values are drawn first.

Parameters

level

[float]

`property stale`
Whether the artist is 'stale' and needs to be re-drawn for the output to match the internal state of the artist.

`property sticky_edges`
`x` and `y` sticky edge lists for autoscaling.

When performing autoscaling, if a data limit coincides with a value in the corresponding sticky_edges list, then no margin will be added--the view limit "sticks" to the edge. A typical use case is histograms, where one usually expects no margin on the bottom edge (0) of the histogram.

This attribute cannot be assigned to; however, the `x` and `y` lists can be modified in place as needed.

Examples

```
>>> artist.sticky_edges.x[:] = (xmin, xmax)
>>> artist.sticky_edges.y[:] = (ymin, ymax)
```

`to_rgba`(*self*, *x*, *alpha=None*, *bytes=False*, *norm=True*)
Return a normalized rgba array corresponding to *x*.

In the normal case, *x* is a 1-D or 2-D sequence of scalars, and the corresponding ndarray of rgba values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already rgb or rgba, such as might have been read from an image file. If *x* is an ndarray with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an rgb or rgba array, and no mapping will be done. The array can be uint8, or it can be floating point with values in the 0-1 range; otherwise a ValueError will be raised. If it is a masked array, the mask will be ignored. If the last dimension is 3, the *alpha* kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the *alpha* kwarg is ignored; it does not replace the pre-existing alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if *bytes* is *False* (default), the rgba array will be floats in the 0-1 range; if it is *True*, the returned rgba array will be uint8 in the 0 to 255 range.

If norm is False, no normalization of the input data is performed, and it is assumed to be in the range (0-1).

`update`(*self*, *props*)
Update this artist's properties from the dict *props*.

Parameters

props

[dict]

`property update_dict`

`update_from`(*self*, *other*)
Copy properties from other to self.

`update_scalarmappable`(*self*)
Update colors from the scalar mappable array, if it is not None.

`zorder = 0`

`class matplotlib.collections.BrokenBarHCollection`(*xranges*, *yrange*, ***kwargs*)
Bases: *matplotlib.collections.PolyCollection*

A collection of horizontal bars spanning *yrange* with a sequence of *xranges*.

Parameters

xranges

[list of (float, float)] The sequence of (left-edge-position, width) pairs for each bar.

yrange

[(float, float)] The (lower-edge, height) common to all bars.

****kwargs**

Forwarded to *Collection*.

`add_callback`(*self*, *func*)
Add a callback function that will be called whenever one of the *Artist*'s properties changes.

Parameters

func

[callable] The callback function. It must have the signature:

```
def func(artist: Artist) -> Any
```

where *artist* is the calling *Artist*. Return values may exist but are ignored.

Returns

int

The observer id associated with the callback. This id can be used for removing the callback with *remove_callback* later.

See also:

remove_callback

`add_checker`(*self*, *checker*)
[*Deprecated*]

Notes

Deprecated since version 3.3:

`autoscale`(*self*)
Autoscale the scalar limits on the norm instance using the current array

`autoscale_None`(*self*)
Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

`property axes`
The *Axes* instance the artist resides in, or *None*.

`changed`(*self*)
Call this whenever the mappable is changed to notify all the callbackSM listeners to the 'changed' signal.

`check_update`(*self*, *checker*)
[*Deprecated*]

Notes

Deprecated since version 3.3:

`contains`(*self*, *mouseevent*)
Test whether the mouse event occurred in the collection.

Returns `bool, dict(ind=itemlist)`, where every item in itemlist contains the event.

`convert_xunits`(*self*, *x*)
Convert *x* using the unit type of the xaxis.

If the artist is not in contained in an Axes or if the xaxis does not have units, *x* itself is returned.

`convert_yunits`(*self*, *y*)
Convert *y* using the unit type of the yaxis.

If the artist is not in contained in an Axes or if the yaxis does not have units, *y* itself is returned.

`draw`(*self*, *renderer*)
Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

Parameters

renderer

[*RendererBase* subclass.]

Notes

This method is overridden in the Artist subclasses.

`findobj`(*self*, *match=None*, *include_self=True*)
Find artist objects.

Recursively find all *Artist* instances contained in the artist.

Parameters

match

A filter criterion for the matches. This can be

- *None*: Return all objects contained in artist.
- A function with signature `def match(artist: Artist) -> bool`. The result will only contain artists for which the function returns *True*.

- A class instance: e.g., *Line2D*. The result will only contain artists of this class or its subclasses (`isinstance` check).

include_self

[bool] Include *self* in the list to be checked for a match.

Returns

list of *Artist*

`format_cursor_data`(*self*, *data*)
Return a string representation of *data*.

Note: This method is intended to be overridden by artist subclasses. As an end-user of Matplotlib you will most likely not call this method yourself.

The default implementation converts ints and floats and arrays of ints and floats into a comma-separated string enclosed in square brackets.

See also:

get_cursor_data

`get_agg_filter`(*self*)
Return filter function to be used for agg filter.

`get_alpha`(*self*)
Return the alpha value used for blending - not supported on all backends.

`get_animated`(*self*)
Return whether the artist is animated.

`get_array`(*self*)
Return the data array.

`get_capstyle`(*self*)

`get_children`(*self*)
Return a list of the child *Artist*s of this *Artist*.

`get_clim`(*self*)
Return the values (min, max) that are mapped to the colormap limits.

`get_clip_box`(*self*)
Return the clipbox.

`get_clip_on`(*self*)
Return whether the artist uses clipping.

`get_clip_path`(*self*)
Return the clip path.

get_cmap(*self*)
Return the *Colormap* instance.

get_contains(*self*)
[*Deprecated*] Return the custom contains function of the artist if set, or *None*.

See also:

set_contains

Notes

Deprecated since version 3.3.

get_cursor_data(*self*, *event*)
Return the cursor data for a given event.

Note: This method is intended to be overridden by artist subclasses. As an end-user of Matplotlib you will most likely not call this method yourself.

Cursor data can be used by Artists to provide additional context information for a given event. The default implementation just returns *None*.

Subclasses can override the method and return arbitrary data. However, when doing so, they must ensure that *format_cursor_data* can convert the data to a string representation.

The only current use case is displaying the z-value of an *AxesImage* in the status bar of a plot window, while moving the mouse.

Parameters

event

[*matplotlib.backend_bases.MouseEvent*]

See also:

format_cursor_data

get_dashes(*self*)
Alias for *get_linestyle*.

get_datalim(*self*, *transData*)

get_ec(*self*)
Alias for *get_edgecolor*.

get_edgecolor(*self*)

get_edgecolors(*self*)
Alias for *get_edgecolor*.

get_facecolor(*self*)

get_facecolors(*self*)
Alias for *get_facecolor*.

get_fc(*self*)
Alias for *get_facecolor*.

get_figure(*self*)
Return the *Figure* instance the artist belongs to.

get_fill(*self*)
Return whether fill is set.

get_gid(*self*)
Return the group id.

get_hatch(*self*)
Return the current hatching pattern.

get_in_layout(*self*)
Return boolean flag, `True` if artist is included in layout calculations.

E.g. *Constrained Layout Guide*, *Figure.tight_layout()*, and `fig.savefig(fname, bbox_inches='tight')`.

get_joinstyle(*self*)

get_label(*self*)
Return the label used for this artist in the legend.

get_linestyle(*self*)

get_linestyles(*self*)
Alias for *get_linestyle*.

get_linewidth(*self*)

get_linewidths(*self*)
Alias for *get_linewidth*.

get_ls(*self*)
Alias for *get_linestyle*.

get_lw(*self*)
Alias for *get_linewidth*.

get_offset_position(*self*)
[*Deprecated*] Return how offsets are applied for the collection. If *offset_position* is 'screen', the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is 'data', the offset is applied before the master transform, i.e., the offsets are in data coordinates.

Notes

Deprecated since version 3.3.

`get_offset_transform`(*self*)

`get_offsets`(*self*)
Return the offsets for the collection.

`get_path_effects`(*self*)

`get_paths`(*self*)

`get_picker`(*self*)
Return the picking behavior of the artist.

The possible values are described in *set_picker*.

See also:

set_picker, *pickable*, *pick*

`get_pickradius`(*self*)

`get_rasterized`(*self*)
Return whether the artist is to be rasterized.

`get_sizes`(*self*)
Return the sizes ('areas') of the elements in the collection.

Returns

array

The 'area' of each element.

`get_sketch_params`(*self*)
Return the sketch parameters for the artist.

Returns

tuple or None

A 3-tuple with the following elements:

- *scale*: The amplitude of the wiggle perpendicular to the source line.
- *length*: The length of the wiggle along the line.
- *randomness*: The scale factor by which the length is shrunken or expanded.

Returns *None* if no sketch parameters were set.

get_snap(*self*)
Return the snap setting.

See *set_snap* for details.

get_tightbbox(*self*, *renderer*)
Like *Artist.get_window_extent*, but includes any clipping.

Parameters

renderer

[*RendererBase* subclass] renderer that will be used to draw the figures (i.e. `fig.canvas.get_renderer()`)

Returns

Bbox

The enclosing bounding box (in figure pixel coordinates).

get_transform(*self*)
Return the *Transform* instance used by this artist.

get_transformed_clip_path_and_affine(*self*)
Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

get_transforms(*self*)

get_url(*self*)
Return the url.

get_urls(*self*)
Return a list of URLs, one for each element of the collection.

The list contains *None* for elements without a URL. See /gallery/misc/hyperlinks_sgskip for an example.

get_visible(*self*)
Return the visibility.

get_window_extent(*self*, *renderer*)
Get the axes bounding box in display space.

The bounding box' width and height are nonnegative.

Subclasses should override for inclusion in the bounding box "tight" calculation. Default is to return an empty bounding box at 0, 0.

Be careful when using this function, the results will not update if the artist window extent of the artist changes. The extent can change due to any changes in the transform stack, such as changing the axes limits, the figure size, or the canvas used (as is done when saving a figure). This can lead to

unexpected behavior where interactive figures will look fine on the screen, but will save incorrectly.

`get_zorder`(*self*)
Return the artist's zorder.

`have_units`(*self*)
Return *True* if units are set on any axis.

`is_transform_set`(*self*)
Return whether the Artist has an explicitly set transform.

This is *True* after *set_transform* has been called.

`property mouseover`
If this property is set to *True*, the artist will be queried for custom context information when the mouse cursor moves over it.

See also *get_cursor_data()*, *ToolCursorPosition* and *NavigationToolbar2*.

`pchanged`(*self*)
Call all of the registered callbacks.

This function is triggered internally when a property is changed.

See also:

add_callback

remove_callback

`pick`(*self*, *mouseevent*)
Process a pick event.

Each child artist will fire a pick event if *mouseevent* is over the artist and the artist has picker set.

See also:

set_picker, *get_picker*, *pickable*

`pickable`(*self*)
Return whether the artist is pickable.

See also:

set_picker, *get_picker*, *pick*

`properties`(*self*)
Return a dictionary of all the properties of the artist.

`remove`(*self*)
Remove the artist from the figure if possible.

The effect will not be visible until the figure is redrawn, e.g., with *FigureCanvasBase.draw_idle*. Call *relim* to update the axes limits if desired.

Note: *relim* will not see collections even if the collection was added to the axes with *autolim* = True.

Note: there is no support for removing the artist's legend entry.

`remove_callback`(*self*, *oid*)
Remove a callback based on its observer id.

See also:

add_callback

`set`(*self*, ***kwargs*)
A property batch setter. Pass *kwargs* to set properties.

`set_aa`(*self*, *aa*)
Alias for *set_antialiased*.

`set_agg_filter`(*self*, *filter_func*)
Set the agg filter.

Parameters

filter_func

[callable] A filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array.

`set_alpha`(*self*, *alpha*)
Set the alpha value used for blending - not supported on all backends.

Parameters

alpha

[float or None]

`set_animated`(*self*, *b*)
Set the artist's animation state.

Parameters

b

[bool]

`set_antialiased`(*self*, *aa*)
Set the antialiasing state for rendering.

Parameters

aa

[bool or list of bools]

`set_antialiaseds`(*self*, *aa*)
Alias for *set_antialiased*.

`set_array`(*self*, *A*)
Set the image array from numpy array *A*.

Parameters

A

[ndarray]

`set_capstyle`(*self*, *cs*)
Set the capstyle for the collection (for all its elements).

Parameters

cs

[{'butt', 'round', 'projecting'}] The capstyle.

`set_clim`(*self*, *vmin=None*, *vmax=None*)
Set the norm limits for image scaling.

Parameters

vmin, vmax

[float] The limits.

The limits may also be passed as a tuple (*vmin*, *vmax*) as a single positional argument.

`set_clip_box`(*self*, *clipbox*)
Set the artist's clip *Bbox*.

Parameters

clipbox

[*Bbox*]

`set_clip_on`(*self*, *b*)
Set whether the artist uses clipping.

When False artists will be visible outside of the axes which can lead to unexpected results.

Parameters

b

[bool]

`set_clip_path`(*self*, *path*, *transform=None*)
Set the artist's clip path.

Parameters

path

[*Patch* or *Path* or *TransformedPath* or None] The clip path. If given a *Path*, *transform* must be provided as well. If *None*, a previously set clip path is removed.

transform

[*Transform*, optional] Only used if *path* is a *Path*, in which case the given *Path* is converted to a *TransformedPath* using *transform*.

Notes

For efficiency, if *path* is a *Rectangle* this method will set the clipping box to the corresponding rectangle and set the clipping path to `None`.

For technical reasons (support of *set*), a tuple (*path*, *transform*) is also accepted as a single positional parameter.

`set_cmap`(*self*, *cmap*)
Set the colormap for luminance data.

Parameters

cmap

[*Colormap* or str or None]

`set_color`(*self*, *c*)
Set both the edgecolor and the facecolor.

Parameters

c

[color or list of rgba tuples]

See also:

Collection.set_facecolor, *Collection.set_edgecolor*

For setting the edge or face color individually.

set_contains(*self*, *picker*)
[*Deprecated*] Define a custom contains test for the artist.

The provided callable replaces the default *contains* method of the artist.

Parameters

picker

[callable] A custom picker function to evaluate if an event is within the artist. The function must have the signature:

```
def contains(artist: Artist, event: MouseEvent) -> bool, dict
```

that returns:

- a bool indicating if the event is within the artist
- a dict of additional information. The dict should at least return the same information as the default `contains()` implementation of the respective artist, but may provide additional information.

Notes

Deprecated since version 3.3.

set_dashes(*self*, *ls*)
Alias for *set_linestyle*.

set_ec(*self*, *c*)
Alias for *set_edgecolor*.

set_edgecolor(*self*, *c*)
Set the edgecolor(s) of the collection.

Parameters

c

[color or list of colors or 'face'] The collection edgecolor(s). If a sequence, the patches cycle through it. If 'face', match the facecolor.

set_edgecolors(*self*, *c*)
Alias for *set_edgecolor*.

set_facecolor(*self*, *c*)
Set the facecolor(s) of the collection. *c* can be a color (all patches have same color), or a sequence of colors; if it is a sequence the patches will cycle through the sequence.

If *c* is 'none', the patch will not be filled.

Parameters

c

[color or list of colors]

`set_facecolors`(*self*, *c*)
Alias for *set_facecolor*.

`set_fc`(*self*, *c*)
Alias for *set_facecolor*.

`set_figure`(*self*, *fig*)
Set the *Figure* instance the artist belongs to.

Parameters

fig

[*Figure*]

`set_gid`(*self*, *gid*)
Set the (group) id for the artist.

Parameters

gid

[str]

`set_hatch`(*self*, *hatch*)
Set the hatching pattern

hatch can be one of:

```
/   - diagonal hatching
\   - back diagonal
|   - vertical
-   - horizontal
+   - crossed
x   - crossed diagonal
o   - small circle
O   - large circle
.   - dots
*   - stars
```

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Unlike other properties such as linewidth and colors, hatching can only be specified for the collection as a whole, not separately for each member.

Parameters

hatch

[{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}]

`set_in_layout`(*self*, *in_layout*)

Set if artist is to be included in layout calculations, E.g. *Constrained Layout Guide*, *`Figure.tight_layout()`*, and `fig.savefig(fname, bbox_inches='tight')`.

Parameters

in_layout

[bool]

`set_joinstyle`(*self*, *js*)

Set the joinstyle for the collection (for all its elements).

Parameters

js

[{'miter', 'round', 'bevel'}] The joinstyle.

`set_label`(*self*, *s*)

Set a label that will be displayed in the legend.

Parameters

s

[object] *s* will be converted to a string by calling `str`.

`set_linestyle`(*self*, *ls*)

Set the linestyle(s) for the collection.

linestyle	description
`'-'` or `'solid'`	solid line
`'--'` or `'dashed'`	dashed line
`'-.'` or `'dashdot'`	dash-dotted line
`':'` or `'dotted'`	dotted line

Alternatively a dash tuple of the following form can be provided:

```
(offset, onoffseq),
```

where `onoffseq` is an even length tuple of on and off ink in points.

Parameters

ls

[str or tuple or list thereof] Valid values for individual linestyles include {'-', '--', '-.', ':', '', (offset, on-off-seq)}. See *Line2D.set_linestyle* for a complete description.

`set_linestyles`(*self*, *ls*)
Alias for *set_linestyle*.

`set_linewidth`(*self*, *lw*)
Set the linewidth(s) for the collection. *lw* can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence

Parameters

lw

[float or list of floats]

`set_linewidths`(*self*, *lw*)
Alias for *set_linewidth*.

`set_ls`(*self*, *ls*)
Alias for *set_linestyle*.

`set_lw`(*self*, *lw*)
Alias for *set_linewidth*.

`set_norm`(*self*, *norm*)
Set the normalization instance.

Parameters

norm

[*Normalize* or None]

Notes

If there are any colorbars using the mappable for this norm, setting the norm of the mappable will reset the norm, locator, and formatters on the colorbar to default.

`set_offset_position`(*self*, *offset_position*)
[*Deprecated*] Set how offsets are applied. If *offset_position* is 'screen' (default) the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is 'data', the offset is applied before the master transform, i.e., the offsets are in data coordinates.

Parameters

offset_position

[{'screen', 'data'}]

Notes

Deprecated since version 3.3.

`set_offsets`(*self*, *offsets*)
Set the offsets for the collection.

Parameters

offsets

[array-like (N, 2) or (2,)]

`set_path_effects`(*self*, *path_effects*)
Set the path effects.

Parameters

path_effects

[*AbstractPathEffect*]

`set_paths`(*self*, *verts*, *closed=True*)
Set the vertices of the polygons.

Parameters

verts

[list of array-like] The sequence of polygons [*verts0*, *verts1*, ...] where each element *verts_i* defines the vertices of polygon *i* as a 2D array-like of shape (M, 2).

closed

[bool, default: True] Whether the polygon should be closed by adding a CLOSEPOLY connection at the end.

`set_picker`(*self*, *picker*)
Define the picking behavior of the artist.

Parameters

picker

[None or bool or callable] This can be one of the following:

- *None*: Picking is disabled for this artist (default).

- A boolean: If *True* then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist.
- A function: If picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

to determine the hit test. if the mouse event is over the artist, return *hit=True* and props is a dictionary of properties you want added to the PickEvent attributes.

- *deprecated*: For *Line2D* only, *picker* can also be a float that sets the tolerance for checking whether an event occurred "on" the line; this is deprecated. Use *Line2D.set_pickradius* instead.

`set_pickradius`(*self*, *pr*)

Set the pick radius used for containment tests.

Parameters

d

[float] Pick radius, in points.

`set_rasterized`(*self*, *rasterized*)

Force rasterized (bitmap) drawing in vector backend output.

Defaults to None, which implies the backend's default behavior.

Parameters

rasterized

[bool or None]

`set_sizes`(*self*, *sizes*, *dpi=72.0*)

Set the sizes of each member of the collection.

Parameters

sizes

[ndarray or None] The size to set for each element of the collection. The value is the 'area' of the element.

dpi

[float, default: 72] The dpi of the canvas.

`set_sketch_params`(*self*, *scale=None*, *length=None*, *randomness=None*)
Sets the sketch parameters.

Parameters

scale

[float, optional] The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is `None`, or not provided, no sketch filter will be provided.

length

[float, optional] The length of the wiggle along the line, in pixels (default 128.0)

randomness

[float, optional] The scale factor by which the length is shrunken or expanded (default 16.0)

`set_snap`(*self*, *snap*)
Set the snapping behavior.

Snapping aligns positions with the pixel grid, which results in clearer images. For example, if a black line of 1px width was defined at a position in between two pixels, the resulting image would contain the interpolated value of that line in the pixel grid, which would be a grey value on both adjacent pixel positions. In contrast, snapping will move the line to the nearest integer pixel value, so that the resulting image will really contain a 1px wide black line.

Snapping is currently only supported by the Agg and MacOSX backends.

Parameters

snap

[bool or None] Possible values:

- *True*: Snap vertices to the nearest pixel center.
- *False*: Do not modify vertex positions.
- *None*: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center.

`set_transform`(*self*, *t*)
Set the artist transform.

Parameters

t

[*Transform*]

`set_url`(*self*, *url*)
Set the url for the artist.

Parameters

url

[str]

`set_urls`(*self*, *urls*)

Parameters

urls

[list of str or None]

Notes

URLs are currently only implemented by the SVG backend. They are ignored by all other backends.

`set_verts`(*self*, *verts*, *closed=True*)
Set the vertices of the polygons.

Parameters

verts

[list of array-like] The sequence of polygons [*verts0*, *verts1*, ...] where each element *verts_i* defines the vertices of polygon *i* as a 2D array-like of shape (M, 2).

closed

[bool, default: True] Whether the polygon should be closed by adding a CLOSEPOLY connection at the end.

`set_verts_and_codes`(*self*, *verts*, *codes*)
Initialize vertices with path codes.

`set_visible`(*self*, *b*)
Set the artist's visibility.

Parameters

b

[bool]

`set_zorder`(*self*, *level*)
Set the zorder for the artist. Artists with lower zorder values are drawn first.

Parameters

level

[float]

classmethod **span_where**(*x*, *ymin*, *ymax*, *where*, ***kwargs*)

Return a *BrokenBarHCollection* that plots horizontal bars from over the regions in *x* where *where* is True. The bars range on the y-axis from *ymin* to *ymax*

kwargs are passed on to the collection.

property **stale**

Whether the artist is 'stale' and needs to be re-drawn for the output to match the internal state of the artist.

property **sticky_edges**

`x` and `y` sticky edge lists for autoscaling.

When performing autoscaling, if a data limit coincides with a value in the corresponding sticky_edges list, then no margin will be added--the view limit "sticks" to the edge. A typical use case is histograms, where one usually expects no margin on the bottom edge (0) of the histogram.

This attribute cannot be assigned to; however, the `x` and `y` lists can be modified in place as needed.

Examples

```
>>> artist.sticky_edges.x[:] = (xmin, xmax)
>>> artist.sticky_edges.y[:] = (ymin, ymax)
```

to_rgba(*self*, *x*, *alpha=None*, *bytes=False*, *norm=True*)

Return a normalized rgba array corresponding to *x*.

In the normal case, *x* is a 1-D or 2-D sequence of scalars, and the corresponding ndarray of rgba values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already rgb or rgba, such as might have been read from an image file. If *x* is an ndarray with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an rgb or rgba array, and no mapping will be done. The array can be uint8, or it can be floating point with values in the 0-1 range; otherwise a ValueError will be raised. If it is a masked array, the mask will be ignored. If the last dimension is 3, the *alpha* kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the *alpha* kwarg is ignored; it does not replace the pre-existing alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if *bytes* is *False* (default), the rgba array will be floats in the 0-1 range; if it is *True*, the returned rgba array will be uint8 in the 0 to 255 range.

If norm is False, no normalization of the input data is performed, and it is assumed to be in the range (0-1).

`update`(*self*, *props*)
Update this artist's properties from the dict *props*.

Parameters

props

[dict]

`property update_dict`

`update_from`(*self*, *other*)
Copy properties from other to self.

`update_scalarmappable`(*self*)
Update colors from the scalar mappable array, if it is not None.

`zorder = 0`

`class matplotlib.collections.CircleCollection`(*sizes*, ***kwargs*)
Bases: `matplotlib.collections._CollectionWithSizes`

A collection of circles, drawn using splines.

Parameters

sizes

[float or array-like] The area of each circle in points^2.

****kwargs**

Forwarded to *Collection*.

`add_callback`(*self*, *func*)
Add a callback function that will be called whenever one of the *Artist*'s properties changes.

Parameters

func

[callable] The callback function. It must have the signature:

```
def func(artist: Artist) -> Any
```

where *artist* is the calling *Artist*. Return values may exist but are ignored.

Returns

int

The observer id associated with the callback. This id can be used for removing the callback with *remove_callback* later.

See also:

remove_callback

`add_checker`(*self*, *checker*)
[*Deprecated*]

Notes

Deprecated since version 3.3:

`autoscale`(*self*)
Autoscale the scalar limits on the norm instance using the current array

`autoscale_None`(*self*)
Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

`property axes`
The *Axes* instance the artist resides in, or *None*.

`changed`(*self*)
Call this whenever the mappable is changed to notify all the callbackSM listeners to the 'changed' signal.

`check_update`(*self*, *checker*)
[*Deprecated*]

Notes

Deprecated since version 3.3:

`contains`(*self*, *mouseevent*)
Test whether the mouse event occurred in the collection.

Returns `bool, dict(ind=itemlist)`, where every item in itemlist contains the event.

`convert_xunits`(*self*, *x*)
Convert *x* using the unit type of the xaxis.

If the artist is not in contained in an Axes or if the xaxis does not have units, *x* itself is returned.

`convert_yunits`(*self*, *y*)
Convert *y* using the unit type of the yaxis.

If the artist is not in contained in an Axes or if the yaxis does not have units, *y* itself is returned.

`draw`(*self*, *renderer*)
Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

Parameters

renderer

[*RendererBase* subclass.]

Notes

This method is overridden in the Artist subclasses.

`findobj`(*self*, *match=None*, *include_self=True*)
Find artist objects.

Recursively find all *Artist* instances contained in the artist.

Parameters

match

A filter criterion for the matches. This can be

- *None*: Return all objects contained in artist.
- A function with signature `def match(artist: Artist) -> bool`. The result will only contain artists for which the function returns *True*.
- A class instance: e.g., *Line2D*. The result will only contain artists of this class or its subclasses (`isinstance` check).

include_self

[bool] Include *self* in the list to be checked for a match.

Returns

list of *Artist*

`format_cursor_data`(*self*, *data*)
Return a string representation of *data*.

Note: This method is intended to be overridden by artist subclasses. As an end-user of Matplotlib you will most likely not call this method yourself.

The default implementation converts ints and floats and arrays of ints and floats into a comma-separated string enclosed in square brackets.

See also:

get_cursor_data

`get_agg_filter`(*self*)
Return filter function to be used for agg filter.

`get_alpha`(*self*)
Return the alpha value used for blending - not supported on all backends.

`get_animated`(*self*)
Return whether the artist is animated.

`get_array`(*self*)
Return the data array.

`get_capstyle`(*self*)

`get_children`(*self*)
Return a list of the child *Artist*s of this *Artist*.

`get_clim`(*self*)
Return the values (min, max) that are mapped to the colormap limits.

`get_clip_box`(*self*)
Return the clipbox.

`get_clip_on`(*self*)
Return whether the artist uses clipping.

`get_clip_path`(*self*)
Return the clip path.

`get_cmap`(*self*)
Return the *Colormap* instance.

`get_contains`(*self*)
[*Deprecated*] Return the custom contains function of the artist if set, or *None*.

See also:

set_contains

Notes

Deprecated since version 3.3.

get_cursor_data(*self*, *event*)
Return the cursor data for a given event.

Note: This method is intended to be overridden by artist subclasses. As an end-user of Matplotlib you will most likely not call this method yourself.

Cursor data can be used by Artists to provide additional context information for a given event. The default implementation just returns *None*.

Subclasses can override the method and return arbitrary data. However, when doing so, they must ensure that *format_cursor_data* can convert the data to a string representation.

The only current use case is displaying the z-value of an *AxesImage* in the status bar of a plot window, while moving the mouse.

Parameters

event

[*matplotlib.backend_bases.MouseEvent*]

See also:

format_cursor_data

get_dashes(*self*)
Alias for *get_linestyle*.

get_datalim(*self*, *transData*)

get_ec(*self*)
Alias for *get_edgecolor*.

get_edgecolor(*self*)

get_edgecolors(*self*)
Alias for *get_edgecolor*.

get_facecolor(*self*)

get_facecolors(*self*)
Alias for *get_facecolor*.

get_fc(*self*)
Alias for *get_facecolor*.

get_figure(*self*)
Return the *Figure* instance the artist belongs to.

`get_fill`(*self*)
Return whether fill is set.

`get_gid`(*self*)
Return the group id.

`get_hatch`(*self*)
Return the current hatching pattern.

`get_in_layout`(*self*)
Return boolean flag, `True` if artist is included in layout calculations.

E.g. *Constrained Layout Guide*, *Figure.tight_layout()*, and `fig.savefig(fname, bbox_inches='tight')`.

`get_joinstyle`(*self*)

`get_label`(*self*)
Return the label used for this artist in the legend.

`get_linestyle`(*self*)

`get_linestyles`(*self*)
Alias for *get_linestyle*.

`get_linewidth`(*self*)

`get_linewidths`(*self*)
Alias for *get_linewidth*.

`get_ls`(*self*)
Alias for *get_linestyle*.

`get_lw`(*self*)
Alias for *get_linewidth*.

`get_offset_position`(*self*)
[*Deprecated*] Return how offsets are applied for the collection. If *offset_position* is 'screen', the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is 'data', the offset is applied before the master transform, i.e., the offsets are in data coordinates.

Notes

Deprecated since version 3.3.

`get_offset_transform`(*self*)

`get_offsets`(*self*)
Return the offsets for the collection.

`get_path_effects`(*self*)

`get_paths`(*self*)

get_picker(*self*)
Return the picking behavior of the artist.

The possible values are described in *set_picker*.

See also:

set_picker, *pickable*, *pick*

get_pickradius(*self*)

get_rasterized(*self*)
Return whether the artist is to be rasterized.

get_sizes(*self*)
Return the sizes ('areas') of the elements in the collection.

Returns

array

The 'area' of each element.

get_sketch_params(*self*)
Return the sketch parameters for the artist.

Returns

tuple or None

A 3-tuple with the following elements:

- *scale*: The amplitude of the wiggle perpendicular to the source line.
- *length*: The length of the wiggle along the line.
- *randomness*: The scale factor by which the length is shrunken or expanded.

Returns *None* if no sketch parameters were set.

get_snap(*self*)
Return the snap setting.

See *set_snap* for details.

get_tightbbox(*self*, *renderer*)
Like *Artist.get_window_extent*, but includes any clipping.

Parameters

renderer

[*RendererBase* subclass] renderer that will be used to draw the figures (i.e. `fig.canvas.get_renderer()`)

Returns

Bbox

The enclosing bounding box (in figure pixel coordinates).

`get_transform`(*self*)
Return the *Transform* instance used by this artist.

`get_transformed_clip_path_and_affine`(*self*)
Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

`get_transforms`(*self*)

`get_url`(*self*)
Return the url.

`get_urls`(*self*)
Return a list of URLs, one for each element of the collection.

The list contains *None* for elements without a URL. See /gallery/misc/hyperlinks_sgskip for an example.

`get_visible`(*self*)
Return the visibility.

`get_window_extent`(*self*, *renderer*)
Get the axes bounding box in display space.

The bounding box' width and height are nonnegative.

Subclasses should override for inclusion in the bounding box "tight" calculation. Default is to return an empty bounding box at 0, 0.

Be careful when using this function, the results will not update if the artist window extent of the artist changes. The extent can change due to any changes in the transform stack, such as changing the axes limits, the figure size, or the canvas used (as is done when saving a figure). This can lead to unexpected behavior where interactive figures will look fine on the screen, but will save incorrectly.

`get_zorder`(*self*)
Return the artist's zorder.

`have_units`(*self*)
Return *True* if units are set on any axis.

`is_transform_set`(*self*)
Return whether the Artist has an explicitly set transform.

This is *True* after *set_transform* has been called.

`property mouseover`
If this property is set to *True*, the artist will be queried for custom context information when the mouse cursor moves over it.

See also *get_cursor_data()*, *ToolCursorPosition* and *NavigationToolbar2*.

`pchanged`(*self*)
Call all of the registered callbacks.

This function is triggered internally when a property is changed.

See also:

add_callback

remove_callback

`pick`(*self, mouseevent*)
Process a pick event.

Each child artist will fire a pick event if *mouseevent* is over the artist and the artist has picker set.

See also:

set_picker, *get_picker*, *pickable*

`pickable`(*self*)
Return whether the artist is pickable.

See also:

set_picker, *get_picker*, *pick*

`properties`(*self*)
Return a dictionary of all the properties of the artist.

`remove`(*self*)
Remove the artist from the figure if possible.

The effect will not be visible until the figure is redrawn, e.g., with *FigureCanvasBase.draw_idle*. Call *relim* to update the axes limits if desired.

Note: *relim* will not see collections even if the collection was added to the axes with *autolim* = True.

Note: there is no support for removing the artist's legend entry.

`remove_callback`(*self, oid*)
Remove a callback based on its observer id.

See also:

add_callback

`set`(*self, **kwargs*)
A property batch setter. Pass *kwargs* to set properties.

set_aa(*self*, *aa*)
Alias for *set_antialiased*.

set_agg_filter(*self*, *filter_func*)
Set the agg filter.

Parameters

filter_func

[callable] A filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array.

set_alpha(*self*, *alpha*)
Set the alpha value used for blending - not supported on all backends.

Parameters

alpha

[float or None]

set_animated(*self*, *b*)
Set the artist's animation state.

Parameters

b

[bool]

set_antialiased(*self*, *aa*)
Set the antialiasing state for rendering.

Parameters

aa

[bool or list of bools]

set_antialiaseds(*self*, *aa*)
Alias for *set_antialiased*.

set_array(*self*, *A*)
Set the image array from numpy array *A*.

Parameters

A

[ndarray]

set_capstyle(*self*, *cs*)
Set the capstyle for the collection (for all its elements).

Parameters

cs

[{'butt', 'round', 'projecting'}] The capstyle.

`set_clim`(*self*, *vmin=None*, *vmax=None*)
Set the norm limits for image scaling.

Parameters

vmin, vmax

[float] The limits.

The limits may also be passed as a tuple (*vmin*, *vmax*) as a single positional argument.

`set_clip_box`(*self*, *clipbox*)
Set the artist's clip *Bbox*.

Parameters

clipbox

[*Bbox*]

`set_clip_on`(*self*, *b*)
Set whether the artist uses clipping.

When False artists will be visible outside of the axes which can lead to unexpected results.

Parameters

b

[bool]

`set_clip_path`(*self*, *path*, *transform=None*)
Set the artist's clip path.

Parameters

path

[*Patch* or *Path* or *TransformedPath* or None] The clip path. If given a *Path*, *transform* must be provided as well. If *None*, a previously set clip path is removed.

transform

[*Transform*, optional] Only used if *path* is a *Path*, in which case the given *Path* is converted to a *TransformedPath* using *transform*.

Notes

For efficiency, if *path* is a *Rectangle* this method will set the clipping box to the corresponding rectangle and set the clipping path to `None`.

For technical reasons (support of *set*), a tuple (*path*, *transform*) is also accepted as a single positional parameter.

`set_cmap`(*self*, *cmap*)
Set the colormap for luminance data.

Parameters

cmap

[*Colormap* or str or None]

`set_color`(*self*, *c*)
Set both the edgecolor and the facecolor.

Parameters

c

[color or list of rgba tuples]

See also:

Collection.set_facecolor, *Collection.set_edgecolor*

For setting the edge or face color individually.

`set_contains`(*self*, *picker*)
[*Deprecated*] Define a custom contains test for the artist.

The provided callable replaces the default *contains* method of the artist.

Parameters

picker

[callable] A custom picker function to evaluate if an event is within the artist. The function must have the signature:

```
def contains(artist: Artist, event: MouseEvent) -> bool, dict
```

that returns:

- a bool indicating if the event is within the artist
- a dict of additional information. The dict should at least return the same information as the default `contains()` implementation of the respective artist, but may provide additional information.

Notes

Deprecated since version 3.3.

`set_dashes`(*self*, *ls*)
Alias for *set_linestyle*.

`set_ec`(*self*, *c*)
Alias for *set_edgecolor*.

`set_edgecolor`(*self*, *c*)
Set the edgecolor(s) of the collection.

Parameters

c

[color or list of colors or 'face'] The collection edgecolor(s). If a sequence, the patches cycle through it. If 'face', match the facecolor.

`set_edgecolors`(*self*, *c*)
Alias for *set_edgecolor*.

`set_facecolor`(*self*, *c*)
Set the facecolor(s) of the collection. *c* can be a color (all patches have same color), or a sequence of colors; if it is a sequence the patches will cycle through the sequence.

If *c* is 'none', the patch will not be filled.

Parameters

c

[color or list of colors]

`set_facecolors`(*self*, *c*)
Alias for *set_facecolor*.

`set_fc`(*self*, *c*)
Alias for *set_facecolor*.

`set_figure`(*self*, *fig*)
Set the *Figure* instance the artist belongs to.

Parameters

fig

[*Figure*]

`set_gid`(*self*, *gid*)
Set the (group) id for the artist.

Parameters

gid

[str]

`set_hatch`(*self*, *hatch*)
Set the hatching pattern

hatch can be one of:

```
/   - diagonal hatching
\   - back diagonal
|   - vertical
-   - horizontal
+   - crossed
x   - crossed diagonal
o   - small circle
O   - large circle
.   - dots
*   - stars
```

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Unlike other properties such as linewidth and colors, hatching can only be specified for the collection as a whole, not separately for each member.

Parameters

hatch

[{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}]

`set_in_layout`(*self*, *in_layout*)
Set if artist is to be included in layout calculations, E.g. *Constrained Layout Guide*, *`Figure.tight_layout()`*, and `fig.savefig(fname, bbox_inches='tight')`.

Parameters

in_layout

[bool]

`set_joinstyle`(*self*, *js*)
Set the joinstyle for the collection (for all its elements).

Parameters

js

[{'miter', 'round', 'bevel'}] The joinstyle.

`set_label`(*self*, *s*)
Set a label that will be displayed in the legend.

Parameters

s

[object] *s* will be converted to a string by calling `str`.

`set_linestyle`(*self*, *ls*)
Set the linestyle(s) for the collection.

linestyle	description
`'-'` or `'solid'`	solid line
`'--'` or `'dashed'`	dashed line
`'-.'` or `'dashdot'`	dash-dotted line
`':'` or `'dotted'`	dotted line

Alternatively a dash tuple of the following form can be provided:

```
(offset, onoffseq),
```

where `onoffseq` is an even length tuple of on and off ink in points.

Parameters

ls

[str or tuple or list thereof] Valid values for individual linestyles include {'-', '--', '-.', ':', '', (offset, on-off-seq)}. See *Line2D.set_linestyle* for a complete description.

`set_linestyles`(*self*, *ls*)
Alias for *set_linestyle*.

`set_linewidth`(*self*, *lw*)
Set the linewidth(s) for the collection. *lw* can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence

Parameters

lw

[float or list of floats]

`set_linewidths`(*self*, *lw*)
Alias for *set_linewidth*.

`set_ls`(*self*, *ls*)
Alias for *set_linestyle*.

`set_lw`(*self*, *lw*)
Alias for *set_linewidth*.

`set_norm`(*self*, *norm*)
Set the normalization instance.

Parameters

norm

[*Normalize* or None]

Notes

If there are any colorbars using the mappable for this norm, setting the norm of the mappable will reset the norm, locator, and formatters on the colorbar to default.

`set_offset_position`(*self*, *offset_position*)
[*Deprecated*] Set how offsets are applied. If *offset_position* is 'screen' (default) the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is 'data', the offset is applied before the master transform, i.e., the offsets are in data coordinates.

Parameters

offset_position

[{'screen', 'data'}]

Notes

Deprecated since version 3.3.

`set_offsets`(*self*, *offsets*)
Set the offsets for the collection.

Parameters

offsets

[array-like (N, 2) or (2,)]

`set_path_effects`(*self*, *path_effects*)
Set the path effects.

Parameters

path_effects

[*AbstractPathEffect*]

`set_paths`(*self*)

`set_picker`(*self*, *picker*)
Define the picking behavior of the artist.

Parameters

picker

[None or bool or callable] This can be one of the following:

- *None*: Picking is disabled for this artist (default).
- A boolean: If *True* then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist.
- A function: If picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

to determine the hit test. if the mouse event is over the artist, return *hit=True* and props is a dictionary of properties you want added to the PickEvent attributes.

- *deprecated*: For *Line2D* only, *picker* can also be a float that sets the tolerance for checking whether an event occurred "on" the line; this is deprecated. Use *Line2D.set_pickradius* instead.

`set_pickradius`(*self*, *pr*)
Set the pick radius used for containment tests.

Parameters

d

[float] Pick radius, in points.

set_rasterized(*self*, *rasterized*)
Force rasterized (bitmap) drawing in vector backend output.

Defaults to None, which implies the backend's default behavior.

Parameters

rasterized

[bool or None]

set_sizes(*self*, *sizes*, *dpi=72.0*)
Set the sizes of each member of the collection.

Parameters

sizes

[ndarray or None] The size to set for each element of the collection. The value is the 'area' of the element.

dpi

[float, default: 72] The dpi of the canvas.

set_sketch_params(*self*, *scale=None*, *length=None*, *randomness=None*)
Sets the sketch parameters.

Parameters

scale

[float, optional] The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is `None`, or not provided, no sketch filter will be provided.

length

[float, optional] The length of the wiggle along the line, in pixels (default 128.0)

randomness

[float, optional] The scale factor by which the length is shrunken or expanded (default 16.0)

set_snap(*self*, *snap*)
Set the snapping behavior.

Snapping aligns positions with the pixel grid, which results in clearer images. For example, if a black line of 1px width was defined at a position in between two pixels, the resulting image would contain the interpolated value of that line in the pixel grid, which would be a grey value on both adjacent pixel positions. In contrast, snapping will move the line to the nearest

integer pixel value, so that the resulting image will really contain a 1px wide black line.

Snapping is currently only supported by the Agg and MacOSX backends.

Parameters

snap

[bool or None] Possible values:

- *True*: Snap vertices to the nearest pixel center.
- *False*: Do not modify vertex positions.
- *None*: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center.

`set_transform`(*self*, *t*)

Set the artist transform.

Parameters

t

[*Transform*]

`set_url`(*self*, *url*)

Set the url for the artist.

Parameters

url

[str]

`set_urls`(*self*, *urls*)

Parameters

urls

[list of str or None]

Notes

URLs are currently only implemented by the SVG backend. They are ignored by all other backends.

`set_visible`(*self*, *b*)

Set the artist's visibility.

Parameters

b

[bool]

`set_zorder`(*self*, *level*)

Set the zorder for the artist. Artists with lower zorder values are drawn first.

Parameters

level

[float]

`property stale`

Whether the artist is 'stale' and needs to be re-drawn for the output to match the internal state of the artist.

`property sticky_edges`

`x` and `y` sticky edge lists for autoscaling.

When performing autoscaling, if a data limit coincides with a value in the corresponding sticky_edges list, then no margin will be added--the view limit "sticks" to the edge. A typical use case is histograms, where one usually expects no margin on the bottom edge (0) of the histogram.

This attribute cannot be assigned to; however, the `x` and `y` lists can be modified in place as needed.

Examples

```
>>> artist.sticky_edges.x[:] = (xmin, xmax)
>>> artist.sticky_edges.y[:] = (ymin, ymax)
```

`to_rgba`(*self*, *x*, *alpha=None*, *bytes=False*, *norm=True*)

Return a normalized rgba array corresponding to *x*.

In the normal case, *x* is a 1-D or 2-D sequence of scalars, and the corresponding ndarray of rgba values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already rgb or rgba, such as might have been read from an image file. If *x* is an ndarray with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an rgb or rgba array, and no mapping will be done. The array can be uint8, or it can be floating point with values in the 0-1 range; otherwise a ValueError will be raised. If it is a masked array, the mask will be ignored. If the last dimension is 3, the *alpha* kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the *alpha* kwarg is ignored; it does not replace the pre-existing alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if *bytes* is *False* (default), the rgba array will be floats in the 0-1 range; if it is *True*, the returned rgba array will be uint8 in the 0 to 255 range.

If norm is False, no normalization of the input data is performed, and it is assumed to be in the range (0-1).

`update`(*self*, *props*)
Update this artist's properties from the dict *props*.

Parameters

props

[dict]

`property update_dict`

`update_from`(*self*, *other*)
Copy properties from other to self.

`update_scalarmappable`(*self*)
Update colors from the scalar mappable array, if it is not None.

`zorder = 0`

`class matplotlib.collections.Collection`(*edgecolors=None*, *facecolors=None*, *linewidths=None*, *linestyles='solid'*, *capstyle=None*, *joinstyle=None*, *antialiaseds=None*, *offsets=None*, *transOffset=None*, *norm=None*, *cmap=None*, *pickradius=5.0*, *hatch=None*, *urls=None*, *offset_position=<deprecated parameter>*, *zorder=1*, ***kwargs*)
Bases: *matplotlib.artist.Artist*, *matplotlib.cm.ScalarMappable*

Base class for Collections. Must be subclassed to be usable.

A Collection represents a sequence of *Patch*es that can be drawn more efficiently together than individually. For example, when a single path is being drawn repeatedly at different offsets, the renderer can typically execute a `draw_marker()` call much more efficiently than a series of repeated calls to `draw_path()` with the offsets put in one-by-one.

Most properties of a collection can be configured per-element. Therefore, Collections have "plural" versions of many of the properties of a *Patch* (e.g. *Collection.get_paths* instead of *Patch.get_path*). Exceptions are the *zorder*, *hatch*, *pickradius*, *capstyle* and *joinstyle* properties, which can only be set globally for the whole collection.

Besides these exceptions, all properties can be specified as single values (applying to all elements) or sequences of values. The property of the `i`th element of the collection is:

```
prop[i % len(prop)]
```

Each Collection can optionally be used as its own *ScalarMappable* by passing the *norm* and *cmap* parameters to its constructor. If the Collection's *ScalarMappable* matrix `_A` has been set (via a call to *Collection.set_array*), then at draw time this internal scalar mappable will be used to set the `facecolors` and `edgecolors`, ignoring those that were manually passed in.

Parameters

edgecolors

[color or list of colors, default: `rcParams["patch.edgecolor"]` (default: `'black'`)] Edge color for each patch making up the collection. The special value 'face' can be passed to make the edgecolor match the facecolor.

facecolors

[color or list of colors, default: `rcParams["patch.facecolor"]` (default: `'C0'`)] Face color for each patch making up the collection.

linewidths

[float or list of floats, default: `rcParams["patch.linewidth"]` (default: `1.0`)] Line width for each patch making up the collection.

linestyles

[str or tuple or list thereof, default: 'solid'] Valid strings are ['solid', 'dashed', 'dashdot', 'dotted', '-', '--', '-.', ':']. Dash tuples should be of the form:

```
(offset, onoffseq),
```

where *onoffseq* is an even length tuple of on and off ink lengths in points. For examples, see /gallery/lines_bars_and_markers/linestyles.

capstyle

[str, default: `rcParams["patch.capstyle"]`] Style to use for capping lines for all paths in the collection. See /gallery/lines_bars_and_markers/joinstyle for a demonstration of each of the allowed values.

joinstyle

[str, default: `rcParams["patch.joinstyle"]`] Style to use for joining lines for all paths in the collection. See /gallery/lines_bars_and_markers/joinstyle for a demonstration of each of the allowed values.

antialiaseds

[bool or list of bool, default: `rcParams["patch.antialiased"]` (default: `True`)] Whether each pach in the collection should be drawn with antialiasing.

offsets

[(float, float) or list thereof, default: (0, 0)] A vector by which to translate each patch after rendering (default is no translation). The translation is performed in screen (pixel) coordinates (i.e. after the Artist's transform is applied).

transOffset

[*Transform*, default: *IdentityTransform*] A single transform which will be applied to each *offsets* vector before it is used.

offset_position

[{'screen' (default), 'data' (deprecated)}] If set to 'data' (deprecated), *offsets* will be treated as if it is in data coordinates instead of in screen coordinates.

norm

[*Normalize*, optional] Forwarded to *ScalarMappable*. The default of `None` means that the first draw call will set `vmin` and `vmax` using the minimum and maximum values of the data.

cmap

[*Colormap*, optional] Forwarded to *ScalarMappable*. The default of `None` will result in `rcParams["image.cmap"]` (default: `'viridis'`) being used.

hatch

[str, optional] Hatching pattern to use in filled paths, if any. Valid strings are ['/', '', '|', '-', '+', 'x', 'o', 'O', '.', '*']. See /gallery/shapes_and_collections/hatch_demo for the meaning of each hatch type.

pickradius

[float, default: 5.0] If `pickradius <= 0`, then *Collection.contains* will return `True` whenever the test point is inside of one of the polygons formed by the control points of a Path in the Collection. On the other hand, if it is greater than 0, then we instead check if the test point is contained in a stroke of width `2*pickradius` following any of the Paths in the Collection.

urls

[list of str, default: None] A URL for each patch to link to once drawn. Currently only works for the SVG backend. See

/gallery/misc/hyperlinks_sgskip for examples.

zorder

[float, default: 1] The drawing order, shared by all Patches in the Collection. See /gallery/misc/zorder_demo for all defaults and examples.

`add_callback`(*self*, *func*)
Add a callback function that will be called whenever one of the *Artist*'s properties changes.

Parameters

func

[callable] The callback function. It must have the signature:

```
def func(artist: Artist) -> Any
```

where *artist* is the calling *Artist*. Return values may exist but are ignored.

Returns

int

The observer id associated with the callback. This id can be used for removing the callback with *remove_callback* later.

See also:

remove_callback

`add_checker`(*self*, *checker*)
[*Deprecated*]

Notes

Deprecated since version 3.3:

`autoscale`(*self*)
Autoscale the scalar limits on the norm instance using the current array

`autoscale_None`(*self*)
Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

`property axes`
The *Axes* instance the artist resides in, or *None*.

`changed`(*self*)
Call this whenever the mappable is changed to notify all the callbackSM listeners to the 'changed' signal.

`check_update`(*self*, *checker*)
[*Deprecated*]

Notes

Deprecated since version 3.3:

`contains`(*self*, *mouseevent*)
Test whether the mouse event occurred in the collection.

Returns `bool, dict(ind=itemlist)`, where every item in itemlist contains the event.

`convert_xunits`(*self*, *x*)
Convert *x* using the unit type of the xaxis.

If the artist is not in contained in an Axes or if the xaxis does not have units, *x* itself is returned.

`convert_yunits`(*self*, *y*)
Convert *y* using the unit type of the yaxis.

If the artist is not in contained in an Axes or if the yaxis does not have units, *y* itself is returned.

`draw`(*self*, *renderer*)
Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

Parameters

renderer

[*RendererBase* subclass.]

Notes

This method is overridden in the Artist subclasses.

`findobj`(*self*, *match=None*, *include_self=True*)
Find artist objects.

Recursively find all *Artist* instances contained in the artist.

Parameters

match

A filter criterion for the matches. This can be

- *None*: Return all objects contained in artist.
- A function with signature `def match(artist: Artist) -> bool`. The result will only contain artists for which the function returns *True*.
- A class instance: e.g., *Line2D*. The result will only contain artists of this class or its subclasses (`isinstance` check).

include_self

[bool] Include *self* in the list to be checked for a match.

Returns

list of *Artist*

`format_cursor_data`(*self*, *data*)
Return a string representation of *data*.

Note: This method is intended to be overridden by artist subclasses. As an end-user of Matplotlib you will most likely not call this method yourself.

The default implementation converts ints and floats and arrays of ints and floats into a comma-separated string enclosed in square brackets.

See also:

get_cursor_data

`get_agg_filter`(*self*)
Return filter function to be used for agg filter.

`get_alpha`(*self*)
Return the alpha value used for blending - not supported on all backends.

`get_animated`(*self*)
Return whether the artist is animated.

`get_array`(*self*)
Return the data array.

`get_capstyle`(*self*)

`get_children`(*self*)
Return a list of the child *Artist*s of this *Artist*.

`get_clim`(*self*)
Return the values (min, max) that are mapped to the colormap limits.

`get_clip_box`(*self*)
Return the clipbox.

`get_clip_on`(*self*)
Return whether the artist uses clipping.

`get_clip_path`(*self*)
Return the clip path.

`get_cmap`(*self*)
Return the *Colormap* instance.

`get_contains`(*self*)
[*Deprecated*] Return the custom contains function of the artist if set, or *None*.

See also:

set_contains

Notes

Deprecated since version 3.3.

`get_cursor_data`(*self*, *event*)
Return the cursor data for a given event.

Note: This method is intended to be overridden by artist subclasses. As an end-user of Matplotlib you will most likely not call this method yourself.

Cursor data can be used by Artists to provide additional context information for a given event. The default implementation just returns *None*.

Subclasses can override the method and return arbitrary data. However, when doing so, they must ensure that *format_cursor_data* can convert the data to a string representation.

The only current use case is displaying the z-value of an *AxesImage* in the status bar of a plot window, while moving the mouse.

Parameters

event

[*matplotlib.backend_bases.MouseEvent*]

See also:

format_cursor_data

get_dashes(*self*)
Alias for *get_linestyle*.

get_datalim(*self*, *transData*)

get_ec(*self*)
Alias for *get_edgecolor*.

get_edgecolor(*self*)

get_edgecolors(*self*)
Alias for *get_edgecolor*.

get_facecolor(*self*)

get_facecolors(*self*)
Alias for *get_facecolor*.

get_fc(*self*)
Alias for *get_facecolor*.

get_figure(*self*)
Return the *Figure* instance the artist belongs to.

get_fill(*self*)
Return whether fill is set.

get_gid(*self*)
Return the group id.

get_hatch(*self*)
Return the current hatching pattern.

get_in_layout(*self*)
Return boolean flag, `True` if artist is included in layout calculations.

E.g. *Constrained Layout Guide*, *Figure.tight_layout()*, and `fig.savefig(fname, bbox_inches='tight')`.

get_joinstyle(*self*)

get_label(*self*)
Return the label used for this artist in the legend.

get_linestyle(*self*)

get_linestyles(*self*)
Alias for *get_linestyle*.

get_linewidth(*self*)

get_linewidths(*self*)
Alias for *get_linewidth*.

get_ls(*self*)
Alias for *get_linestyle*.

get_lw(*self*)
Alias for *get_linewidth*.

get_offset_position(*self*)
[*Deprecated*] Return how offsets are applied for the collection. If *offset_position* is 'screen', the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is 'data', the offset is applied before the master transform, i.e., the offsets are in data coordinates.

Notes

Deprecated since version 3.3.

get_offset_transform(*self*)

get_offsets(*self*)
Return the offsets for the collection.

get_path_effects(*self*)

get_paths(*self*)

get_picker(*self*)
Return the picking behavior of the artist.

The possible values are described in *set_picker*.

See also:

set_picker, *pickable*, *pick*

get_pickradius(*self*)

get_rasterized(*self*)
Return whether the artist is to be rasterized.

get_sketch_params(*self*)
Return the sketch parameters for the artist.

Returns

tuple or None

A 3-tuple with the following elements:

- *scale*: The amplitude of the wiggle perpendicular to the source line.
- *length*: The length of the wiggle along the line.
- *randomness*: The scale factor by which the length is shrunken or expanded.

Returns *None* if no sketch parameters were set.

`get_snap`(*self*)
Return the snap setting.

See *set_snap* for details.

`get_tightbbox`(*self*, *renderer*)
Like *Artist.get_window_extent*, but includes any clipping.

Parameters

renderer

[*RendererBase* subclass] renderer that will be used to draw the figures (i.e. `fig.canvas.get_renderer()`)

Returns

Bbox

The enclosing bounding box (in figure pixel coordinates).

`get_transform`(*self*)
Return the *Transform* instance used by this artist.

`get_transformed_clip_path_and_affine`(*self*)
Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

`get_transforms`(*self*)

`get_url`(*self*)
Return the url.

`get_urls`(*self*)
Return a list of URLs, one for each element of the collection.

The list contains *None* for elements without a URL. See /gallery/misc/hyperlinks_sgskip for an example.

`get_visible`(*self*)
Return the visibility.

`get_window_extent`(*self*, *renderer*)
Get the axes bounding box in display space.

The bounding box' width and height are nonnegative.

Subclasses should override for inclusion in the bounding box "tight" calculation. Default is to return an empty bounding box at 0, 0.

Be careful when using this function, the results will not update if the artist window extent of the artist changes. The extent can change due to any changes in the transform stack, such as changing the axes limits, the figure size, or the canvas used (as is done when saving a figure). This can lead to

unexpected behavior where interactive figures will look fine on the screen, but will save incorrectly.

`get_zorder`(*self*)
Return the artist's zorder.

`have_units`(*self*)
Return *True* if units are set on any axis.

`is_transform_set`(*self*)
Return whether the Artist has an explicitly set transform.

This is *True* after *set_transform* has been called.

`property mouseover`
If this property is set to *True*, the artist will be queried for custom context information when the mouse cursor moves over it.

See also *get_cursor_data()*, *ToolCursorPosition* and *NavigationToolbar2*.

`pchanged`(*self*)
Call all of the registered callbacks.

This function is triggered internally when a property is changed.

See also:

add_callback

remove_callback

`pick`(*self*, *mouseevent*)
Process a pick event.

Each child artist will fire a pick event if *mouseevent* is over the artist and the artist has picker set.

See also:

set_picker, *get_picker*, *pickable*

`pickable`(*self*)
Return whether the artist is pickable.

See also:

set_picker, *get_picker*, *pick*

`properties`(*self*)
Return a dictionary of all the properties of the artist.

`remove`(*self*)
Remove the artist from the figure if possible.

The effect will not be visible until the figure is redrawn, e.g., with *FigureCanvasBase.draw_idle*. Call *relim* to update the axes limits if desired.

Note: *relim* will not see collections even if the collection was added to the axes with *autolim* = True.

Note: there is no support for removing the artist's legend entry.

`remove_callback`(*self*, *oid*)
Remove a callback based on its observer id.

See also:

add_callback

`set`(*self*, ***kwargs*)
A property batch setter. Pass *kwargs* to set properties.

`set_aa`(*self*, *aa*)
Alias for *set_antialiased*.

`set_agg_filter`(*self*, *filter_func*)
Set the agg filter.

Parameters

filter_func

[callable] A filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array.

`set_alpha`(*self*, *alpha*)
Set the alpha value used for blending - not supported on all backends.

Parameters

alpha

[float or None]

`set_animated`(*self*, *b*)
Set the artist's animation state.

Parameters

b

[bool]

`set_antialiased`(*self*, *aa*)
Set the antialiasing state for rendering.

Parameters

aa

[bool or list of bools]

`set_antialiaseds`(*self*, *aa*)
Alias for *set_antialiased*.

`set_array`(*self*, *A*)
Set the image array from numpy array *A*.

Parameters

A

[ndarray]

`set_capstyle`(*self*, *cs*)
Set the capstyle for the collection (for all its elements).

Parameters

cs

[{'butt', 'round', 'projecting'}] The capstyle.

`set_clim`(*self*, *vmin=None*, *vmax=None*)
Set the norm limits for image scaling.

Parameters

vmin, vmax

[float] The limits.

The limits may also be passed as a tuple (*vmin*, *vmax*) as a single positional argument.

`set_clip_box`(*self*, *clipbox*)
Set the artist's clip *Bbox*.

Parameters

clipbox

[*Bbox*]

`set_clip_on`(*self*, *b*)
Set whether the artist uses clipping.

When False artists will be visible outside of the axes which can lead to unexpected results.

Parameters

b

[bool]

set_clip_path(*self*, *path*, *transform=None*)
Set the artist's clip path.

Parameters

path

[*Patch* or *Path* or *TransformedPath* or None] The clip path. If given a *Path*, *transform* must be provided as well. If *None*, a previously set clip path is removed.

transform

[*Transform*, optional] Only used if *path* is a *Path*, in which case the given *Path* is converted to a *TransformedPath* using *transform*.

Notes

For efficiency, if *path* is a *Rectangle* this method will set the clipping box to the corresponding rectangle and set the clipping path to `None`.

For technical reasons (support of *set*), a tuple (*path*, *transform*) is also accepted as a single positional parameter.

set_cmap(*self*, *cmap*)
Set the colormap for luminance data.

Parameters

cmap

[*Colormap* or str or None]

set_color(*self*, *c*)
Set both the edgecolor and the facecolor.

Parameters

c

[color or list of rgba tuples]

See also:

Collection.set_facecolor, *Collection.set_edgecolor*

For setting the edge or face color individually.

set_contains(*self*, *picker*)
[*Deprecated*] Define a custom contains test for the artist.

The provided callable replaces the default *contains* method of the artist.

Parameters

picker

[callable] A custom picker function to evaluate if an event is within the artist. The function must have the signature:

```
def contains(artist: Artist, event: MouseEvent) -> bool, dict
```

that returns:

- a bool indicating if the event is within the artist
- a dict of additional information. The dict should at least return the same information as the default `contains()` implementation of the respective artist, but may provide additional information.

Notes

Deprecated since version 3.3.

set_dashes(*self*, *ls*)
Alias for *set_linestyle*.

set_ec(*self*, *c*)
Alias for *set_edgecolor*.

set_edgecolor(*self*, *c*)
Set the edgecolor(s) of the collection.

Parameters

c

[color or list of colors or 'face'] The collection edgecolor(s). If a sequence, the patches cycle through it. If 'face', match the facecolor.

set_edgecolors(*self*, *c*)
Alias for *set_edgecolor*.

set_facecolor(*self*, *c*)
Set the facecolor(s) of the collection. *c* can be a color (all patches have same color), or a sequence of colors; if it is a sequence the patches will cycle through the sequence.

If *c* is 'none', the patch will not be filled.

Parameters

c

[color or list of colors]

`set_facecolors`(*self*, *c*)
Alias for *set_facecolor*.

`set_fc`(*self*, *c*)
Alias for *set_facecolor*.

`set_figure`(*self*, *fig*)
Set the *Figure* instance the artist belongs to.

Parameters

fig

[*Figure*]

`set_gid`(*self*, *gid*)
Set the (group) id for the artist.

Parameters

gid

[str]

`set_hatch`(*self*, *hatch*)
Set the hatching pattern

hatch can be one of:

```
/   - diagonal hatching
\   - back diagonal
|   - vertical
-   - horizontal
+   - crossed
x   - crossed diagonal
o   - small circle
O   - large circle
.   - dots
*   - stars
```

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Unlike other properties such as linewidth and colors, hatching can only be specified for the collection as a whole, not separately for each member.

Parameters

hatch

[{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}]

`set_in_layout`(*self*, *in_layout*)

Set if artist is to be included in layout calculations, E.g. *Constrained Layout Guide*, *`Figure.tight_layout()`*, and `fig.savefig(fname, bbox_inches='tight')`.

Parameters

in_layout

[bool]

`set_joinstyle`(*self*, *js*)

Set the joinstyle for the collection (for all its elements).

Parameters

js

[{'miter', 'round', 'bevel'}] The joinstyle.

`set_label`(*self*, *s*)

Set a label that will be displayed in the legend.

Parameters

s

[object] *s* will be converted to a string by calling `str`.

`set_linestyle`(*self*, *ls*)

Set the linestyle(s) for the collection.

linestyle	description
`'-'` or `'solid'`	solid line
`'--'` or `'dashed'`	dashed line
`'-.'` or `'dashdot'`	dash-dotted line
`':'` or `'dotted'`	dotted line

Alternatively a dash tuple of the following form can be provided:

```
(offset, onoffseq),
```

where `onoffseq` is an even length tuple of on and off ink in points.

Parameters

ls

[str or tuple or list thereof] Valid values for individual linestyles include {'-', '--', '-.', ':', '', (offset, on-off-seq)}. See *Line2D.set_linestyle* for a complete description.

`set_linestyles`(*self*, *ls*)
Alias for *set_linestyle*.

`set_linewidth`(*self*, *lw*)
Set the linewidth(s) for the collection. *lw* can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence

Parameters

lw

[float or list of floats]

`set_linewidths`(*self*, *lw*)
Alias for *set_linewidth*.

`set_ls`(*self*, *ls*)
Alias for *set_linestyle*.

`set_lw`(*self*, *lw*)
Alias for *set_linewidth*.

`set_norm`(*self*, *norm*)
Set the normalization instance.

Parameters

norm

[*Normalize* or None]

Notes

If there are any colorbars using the mappable for this norm, setting the norm of the mappable will reset the norm, locator, and formatters on the colorbar to default.

`set_offset_position`(*self*, *offset_position*)
[*Deprecated*] Set how offsets are applied. If *offset_position* is 'screen' (default) the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is 'data', the offset is applied before the master transform, i.e., the offsets are in data coordinates.

Parameters

offset_position

[{'screen', 'data'}]

Notes

Deprecated since version 3.3.

`set_offsets`(*self*, *offsets*)
Set the offsets for the collection.

Parameters

offsets

[array-like (N, 2) or (2,)]

`set_path_effects`(*self*, *path_effects*)
Set the path effects.

Parameters

path_effects

[*AbstractPathEffect*]

`set_paths`(*self*)

`set_picker`(*self*, *picker*)
Define the picking behavior of the artist.

Parameters

picker

[None or bool or callable] This can be one of the following:

- *None*: Picking is disabled for this artist (default).
- A boolean: If *True* then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist.
- A function: If picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

to determine the hit test. if the mouse event is over the artist, return *hit=True* and props is a dictionary of properties you want added to the PickEvent attributes.

- *deprecated*: For *Line2D* only, *picker* can also be a float that sets the tolerance for checking whether an event occurred "on" the line; this is deprecated. Use *Line2D.set_pickradius* instead.

`set_pickradius`(*self*, *pr*)
Set the pick radius used for containment tests.

Parameters

d

[float] Pick radius, in points.

`set_rasterized`(*self*, *rasterized*)
Force rasterized (bitmap) drawing in vector backend output.

Defaults to None, which implies the backend's default behavior.

Parameters

rasterized

[bool or None]

`set_sketch_params`(*self*, *scale=None*, *length=None*, *randomness=None*)
Sets the sketch parameters.

Parameters

scale

[float, optional] The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is `None`, or not provided, no sketch filter will be provided.

length

[float, optional] The length of the wiggle along the line, in pixels (default 128.0)

randomness

[float, optional] The scale factor by which the length is shrunken or expanded (default 16.0)

`set_snap`(*self*, *snap*)
Set the snapping behavior.

Snapping aligns positions with the pixel grid, which results in clearer images. For example, if a black line of 1px width was defined at a position in between two pixels, the resulting image would contain the interpolated value of that line in the pixel grid, which would be a grey value on both adjacent pixel positions. In contrast, snapping will move the line to the nearest

integer pixel value, so that the resulting image will really contain a 1px wide black line.

Snapping is currently only supported by the Agg and MacOSX backends.

Parameters

snap

[bool or None] Possible values:

- *True*: Snap vertices to the nearest pixel center.
- *False*: Do not modify vertex positions.
- *None*: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center.

`set_transform`(*self*, *t*)

Set the artist transform.

Parameters

t

[*Transform*]

`set_url`(*self*, *url*)

Set the url for the artist.

Parameters

url

[str]

`set_urls`(*self*, *urls*)

Parameters

urls

[list of str or None]

Notes

URLs are currently only implemented by the SVG backend. They are ignored by all other backends.

`set_visible`(*self*, *b*)

Set the artist's visibility.

Parameters

b

[bool]

`set_zorder`(*self*, *level*)

Set the zorder for the artist. Artists with lower zorder values are drawn first.

Parameters

level

[float]

`property stale`

Whether the artist is 'stale' and needs to be re-drawn for the output to match the internal state of the artist.

`property sticky_edges`

`x` and `y` sticky edge lists for autoscaling.

When performing autoscaling, if a data limit coincides with a value in the corresponding sticky_edges list, then no margin will be added--the view limit "sticks" to the edge. A typical use case is histograms, where one usually expects no margin on the bottom edge (0) of the histogram.

This attribute cannot be assigned to; however, the `x` and `y` lists can be modified in place as needed.

Examples

```
>>> artist.sticky_edges.x[:] = (xmin, xmax)
>>> artist.sticky_edges.y[:] = (ymin, ymax)
```

`to_rgba`(*self*, *x*, *alpha=None*, *bytes=False*, *norm=True*)

Return a normalized rgba array corresponding to *x*.

In the normal case, *x* is a 1-D or 2-D sequence of scalars, and the corresponding ndarray of rgba values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already rgb or rgba, such as might have been read from an image file. If *x* is an ndarray with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an rgb or rgba array, and no mapping will be done. The array can be uint8, or it can be floating point with values in the 0-1 range; otherwise a ValueError will be raised. If it is a masked array, the mask will be ignored. If the last dimension is 3, the *alpha* kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the *alpha* kwarg is ignored; it does not replace the pre-existing alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if *bytes* is *False* (default), the rgba array will be floats in the 0-1 range; if it is *True*, the returned rgba array will be uint8 in the 0 to 255 range.

If norm is False, no normalization of the input data is performed, and it is assumed to be in the range (0-1).

`update`(*self*, *props*)
Update this artist's properties from the dict *props*.

Parameters

props

[dict]

`property update_dict`

`update_from`(*self*, *other*)
Copy properties from other to self.

`update_scalarmappable`(*self*)
Update colors from the scalar mappable array, if it is not None.

`zorder = 0`

`class matplotlib.collections.EllipseCollection`(*widths*, *heights*, *angles*, *units='points'*, ***kwargs*)
Bases: *matplotlib.collections.Collection*

A collection of ellipses, drawn using splines.

Parameters

widths

[array-like] The lengths of the first axes (e.g., major axis lengths).

heights

[array-like] The lengths of second axes.

angles

[array-like] The angles of the first axes, degrees CCW from the x-axis.

units

[{'points', 'inches', 'dots', 'width', 'height', 'x', 'y', 'xy'}] The units in which majors and minors are given; 'width' and 'height' refer to the dimensions of the axes, while 'x' and 'y' refer to the *offsets* data units. 'xy' differs from all others in that the angle as plotted varies with the aspect ratio, and equals the specified angle only when the aspect ratio is unity. Hence it behaves the same as the *Ellipse* with `axes.transData` as its transform.

****kwargs**

Forwarded to *Collection*.

add_callback(*self*, *func*)
Add a callback function that will be called whenever one of the *Artist*'s properties changes.

Parameters

func

[callable] The callback function. It must have the signature:

```
def func(artist: Artist) -> Any
```

where *artist* is the calling *Artist*. Return values may exist but are ignored.

Returns

int

The observer id associated with the callback. This id can be used for removing the callback with *remove_callback* later.

See also:

remove_callback

add_checker(*self*, *checker*)
[*Deprecated*]

Notes

Deprecated since version 3.3:

autoscale(*self*)
Autoscale the scalar limits on the norm instance using the current array

autoscale_None(*self*)
Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

property axes
The *Axes* instance the artist resides in, or *None*.

changed(*self*)
Call this whenever the mappable is changed to notify all the callbackSM listeners to the 'changed' signal.

`check_update`(*self*, *checker*)
[*Deprecated*]

Notes

Deprecated since version 3.3:

`contains`(*self*, *mouseevent*)
Test whether the mouse event occurred in the collection.

Returns `bool, dict(ind=itemlist)`, where every item in itemlist contains the event.

`convert_xunits`(*self*, *x*)
Convert *x* using the unit type of the xaxis.

If the artist is not in contained in an Axes or if the xaxis does not have units, *x* itself is returned.

`convert_yunits`(*self*, *y*)
Convert *y* using the unit type of the yaxis.

If the artist is not in contained in an Axes or if the yaxis does not have units, *y* itself is returned.

`draw`(*self*, *renderer*)
Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

Parameters

renderer

[*RendererBase* subclass.]

Notes

This method is overridden in the Artist subclasses.

`findobj`(*self*, *match=None*, *include_self=True*)
Find artist objects.

Recursively find all *Artist* instances contained in the artist.

Parameters

match

A filter criterion for the matches. This can be

- *None*: Return all objects contained in artist.

- A function with signature `def match(artist: Artist) -> bool`. The result will only contain artists for which the function returns *True*.
- A class instance: e.g., *Line2D*. The result will only contain artists of this class or its subclasses (`isinstance` check).

include_self

[bool] Include *self* in the list to be checked for a match.

Returns

list of *Artist*

`format_cursor_data`(*self*, *data*)
Return a string representation of *data*.

Note: This method is intended to be overridden by artist subclasses. As an end-user of Matplotlib you will most likely not call this method yourself.

The default implementation converts ints and floats and arrays of ints and floats into a comma-separated string enclosed in square brackets.

See also:

get_cursor_data

`get_agg_filter`(*self*)
Return filter function to be used for agg filter.

`get_alpha`(*self*)
Return the alpha value used for blending - not supported on all backends.

`get_animated`(*self*)
Return whether the artist is animated.

`get_array`(*self*)
Return the data array.

`get_capstyle`(*self*)

`get_children`(*self*)
Return a list of the child *Artist*s of this *Artist*.

`get_clim`(*self*)
Return the values (min, max) that are mapped to the colormap limits.

`get_clip_box`(*self*)
Return the clipbox.

`get_clip_on`(*self*)
Return whether the artist uses clipping.

get_clip_path(*self*)
Return the clip path.

get_cmap(*self*)
Return the *Colormap* instance.

get_contains(*self*)
[*Deprecated*] Return the custom contains function of the artist if set, or *None*.

See also:

set_contains

Notes

Deprecated since version 3.3.

get_cursor_data(*self*, *event*)
Return the cursor data for a given event.

Note: This method is intended to be overridden by artist subclasses. As an end-user of Matplotlib you will most likely not call this method yourself.

Cursor data can be used by Artists to provide additional context information for a given event. The default implementation just returns *None*.

Subclasses can override the method and return arbitrary data. However, when doing so, they must ensure that *format_cursor_data* can convert the data to a string representation.

The only current use case is displaying the z-value of an *AxesImage* in the status bar of a plot window, while moving the mouse.

Parameters

event

[*matplotlib.backend_bases.MouseEvent*]

See also:

format_cursor_data

get_dashes(*self*)
Alias for *get_linestyle*.

get_datalim(*self*, *transData*)

get_ec(*self*)
Alias for *get_edgecolor*.

get_edgecolor(*self*)

get_edgecolors(*self*)
Alias for *get_edgecolor*.

get_facecolor(*self*)

get_facecolors(*self*)
Alias for *get_facecolor*.

get_fc(*self*)
Alias for *get_facecolor*.

get_figure(*self*)
Return the *Figure* instance the artist belongs to.

get_fill(*self*)
Return whether fill is set.

get_gid(*self*)
Return the group id.

get_hatch(*self*)
Return the current hatching pattern.

get_in_layout(*self*)
Return boolean flag, `True` if artist is included in layout calculations.

E.g. *Constrained Layout Guide*, *Figure.tight_layout()*, and `fig.savefig(fname, bbox_inches='tight')`.

get_joinstyle(*self*)

get_label(*self*)
Return the label used for this artist in the legend.

get_linestyle(*self*)

get_linestyles(*self*)
Alias for *get_linestyle*.

get_linewidth(*self*)

get_linewidths(*self*)
Alias for *get_linewidth*.

get_ls(*self*)
Alias for *get_linestyle*.

get_lw(*self*)
Alias for *get_linewidth*.

get_offset_position(*self*)
[*Deprecated*] Return how offsets are applied for the collection. If *offset_position* is 'screen', the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position

is 'data', the offset is applied before the master transform, i.e., the offsets are in data coordinates.

Notes

Deprecated since version 3.3.

`get_offset_transform`(*self*)

`get_offsets`(*self*)
Return the offsets for the collection.

`get_path_effects`(*self*)

`get_paths`(*self*)

`get_picker`(*self*)
Return the picking behavior of the artist.

The possible values are described in *set_picker*.

See also:

set_picker, *pickable*, *pick*

`get_pickradius`(*self*)

`get_rasterized`(*self*)
Return whether the artist is to be rasterized.

`get_sketch_params`(*self*)
Return the sketch parameters for the artist.

Returns

tuple or None

A 3-tuple with the following elements:

- *scale*: The amplitude of the wiggle perpendicular to the source line.
- *length*: The length of the wiggle along the line.
- *randomness*: The scale factor by which the length is shrunken or expanded.

Returns *None* if no sketch parameters were set.

`get_snap`(*self*)
Return the snap setting.

See *set_snap* for details.

`get_tightbbox`(*self*, *renderer*)
Like *Artist.get_window_extent*, but includes any clipping.

Parameters

renderer

[*RendererBase* subclass] renderer that will be used to draw the figures (i.e. `fig.canvas.get_renderer()`)

Returns

Bbox

The enclosing bounding box (in figure pixel coordinates).

`get_transform`(*self*)
Return the *Transform* instance used by this artist.

`get_transformed_clip_path_and_affine`(*self*)
Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

`get_transforms`(*self*)

`get_url`(*self*)
Return the url.

`get_urls`(*self*)
Return a list of URLs, one for each element of the collection.

The list contains *None* for elements without a URL. See /gallery/misc/hyperlinks_sgskip for an example.

`get_visible`(*self*)
Return the visibility.

`get_window_extent`(*self*, *renderer*)
Get the axes bounding box in display space.

The bounding box' width and height are nonnegative.

Subclasses should override for inclusion in the bounding box "tight" calculation. Default is to return an empty bounding box at 0, 0.

Be careful when using this function, the results will not update if the artist window extent of the artist changes. The extent can change due to any changes in the transform stack, such as changing the axes limits, the figure size, or the canvas used (as is done when saving a figure). This can lead to unexpected behavior where interactive figures will look fine on the screen, but will save incorrectly.

`get_zorder`(*self*)
Return the artist's zorder.

`have_units`(*self*)
Return *True* if units are set on any axis.

`is_transform_set`(*self*)
Return whether the Artist has an explicitly set transform.

This is *True* after *set_transform* has been called.

`property mouseover`
If this property is set to *True*, the artist will be queried for custom context information when the mouse cursor moves over it.

See also *get_cursor_data()*, *ToolCursorPosition* and *NavigationToolbar2*.

`pchanged`(*self*)
Call all of the registered callbacks.

This function is triggered internally when a property is changed.

See also:

add_callback

remove_callback

`pick`(*self, mouseevent*)
Process a pick event.

Each child artist will fire a pick event if *mouseevent* is over the artist and the artist has picker set.

See also:

set_picker, *get_picker*, *pickable*

`pickable`(*self*)
Return whether the artist is pickable.

See also:

set_picker, *get_picker*, *pick*

`properties`(*self*)
Return a dictionary of all the properties of the artist.

`remove`(*self*)
Remove the artist from the figure if possible.

The effect will not be visible until the figure is redrawn, e.g., with *FigureCanvasBase.draw_idle*. Call *relim* to update the axes limits if desired.

Note: *relim* will not see collections even if the collection was added to the axes with *autolim* = True.

Note: there is no support for removing the artist's legend entry.

`remove_callback`(*self*, *oid*)
Remove a callback based on its observer id.

See also:

add_callback

`set`(*self*, ***kwargs*)
A property batch setter. Pass *kwargs* to set properties.

`set_aa`(*self*, *aa*)
Alias for *set_antialiased*.

`set_agg_filter`(*self*, *filter_func*)
Set the agg filter.

Parameters

filter_func

[callable] A filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array.

`set_alpha`(*self*, *alpha*)
Set the alpha value used for blending - not supported on all backends.

Parameters

alpha

[float or None]

`set_animated`(*self*, *b*)
Set the artist's animation state.

Parameters

b

[bool]

`set_antialiased`(*self*, *aa*)
Set the antialiasing state for rendering.

Parameters

aa

[bool or list of bools]

`set_antialiaseds`(*self*, *aa*)
Alias for *set_antialiased*.

`set_array`(*self*, *A*)
Set the image array from numpy array *A*.

Parameters

A

[ndarray]

`set_capstyle`(*self*, *cs*)
Set the capstyle for the collection (for all its elements).

Parameters

cs

[{'butt', 'round', 'projecting'}] The capstyle.

`set_clim`(*self*, *vmin=None*, *vmax=None*)
Set the norm limits for image scaling.

Parameters

vmin, vmax

[float] The limits.

The limits may also be passed as a tuple (*vmin*, *vmax*) as a single positional argument.

`set_clip_box`(*self*, *clipbox*)
Set the artist's clip *Bbox*.

Parameters

clipbox

[*Bbox*]

`set_clip_on`(*self*, *b*)
Set whether the artist uses clipping.

When False artists will be visible outside of the axes which can lead to unexpected results.

Parameters

b

[bool]

`set_clip_path`(*self*, *path*, *transform=None*)
Set the artist's clip path.

Parameters

path

[*Patch* or *Path* or *TransformedPath* or None] The clip path. If given a *Path*, *transform* must be provided as well. If *None*, a previously set clip path is removed.

transform

[*Transform*, optional] Only used if *path* is a *Path*, in which case the given *Path* is converted to a *TransformedPath* using *transform*.

Notes

For efficiency, if *path* is a *Rectangle* this method will set the clipping box to the corresponding rectangle and set the clipping path to `None`.

For technical reasons (support of *set*), a tuple (*path*, *transform*) is also accepted as a single positional parameter.

`set_cmap`(*self*, *cmap*)
Set the colormap for luminance data.

Parameters

cmap

[*Colormap* or str or None]

`set_color`(*self*, *c*)
Set both the edgecolor and the facecolor.

Parameters

c

[color or list of rgba tuples]

See also:

Collection.set_facecolor, *Collection.set_edgecolor*

For setting the edge or face color individually.

`set_contains`(*self*, *picker*)
[*Deprecated*] Define a custom contains test for the artist.

The provided callable replaces the default *contains* method of the artist.

Parameters

picker

[callable] A custom picker function to evaluate if an event is within the artist. The function must have the signature:

```
def contains(artist: Artist, event: MouseEvent) -> bool, dict
```

that returns:

- a bool indicating if the event is within the artist
- a dict of additional information. The dict should at least return the same information as the default `contains()` implementation of the respective artist, but may provide additional information.

Notes

Deprecated since version 3.3.

`set_dashes`(*self*, *ls*)
Alias for *set_linestyle*.

`set_ec`(*self*, *c*)
Alias for *set_edgecolor*.

`set_edgecolor`(*self*, *c*)
Set the edgecolor(s) of the collection.

Parameters

c

[color or list of colors or 'face'] The collection edgecolor(s). If a sequence, the patches cycle through it. If 'face', match the facecolor.

`set_edgecolors`(*self*, *c*)
Alias for *set_edgecolor*.

`set_facecolor`(*self*, *c*)
Set the facecolor(s) of the collection. *c* can be a color (all patches have same color), or a sequence of colors; if it is a sequence the patches will cycle through the sequence.

If *c* is 'none', the patch will not be filled.

Parameters

c

[color or list of colors]

set_facecolors(*self*, *c*)
Alias for *set_facecolor*.

set_fc(*self*, *c*)
Alias for *set_facecolor*.

set_figure(*self*, *fig*)
Set the *Figure* instance the artist belongs to.

Parameters

fig

[*Figure*]

set_gid(*self*, *gid*)
Set the (group) id for the artist.

Parameters

gid

[str]

set_hatch(*self*, *hatch*)
Set the hatching pattern

hatch can be one of:

```
/   - diagonal hatching
\   - back diagonal
|   - vertical
-   - horizontal
+   - crossed
x   - crossed diagonal
o   - small circle
O   - large circle
.   - dots
*   - stars
```

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Unlike other properties such as linewidth and colors, hatching can only be specified for the collection as a whole, not separately for each member.

Parameters

hatch

[{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}]

`set_in_layout`(*self*, *in_layout*)
Set if artist is to be included in layout calculations, E.g. *Constrained Layout Guide*, *`Figure.tight_layout()`*, and `fig.savefig(fname, bbox_inches='tight')`.

Parameters

in_layout

[bool]

`set_joinstyle`(*self*, *js*)
Set the joinstyle for the collection (for all its elements).

Parameters

js

[{'miter', 'round', 'bevel'}] The joinstyle.

`set_label`(*self*, *s*)
Set a label that will be displayed in the legend.

Parameters

s

[object] *s* will be converted to a string by calling `str`.

`set_linestyle`(*self*, *ls*)
Set the linestyle(s) for the collection.

linestyle	description
`'-'` or `'solid'`	solid line
`'--'` or `'dashed'`	dashed line
`'-.'` or `'dashdot'`	dash-dotted line
`':'` or `'dotted'`	dotted line

Alternatively a dash tuple of the following form can be provided:

```
(offset, onoffseq),
```

where `onoffseq` is an even length tuple of on and off ink in points.

Parameters

ls

[str or tuple or list thereof] Valid values for individual linestyles include {'-', '--', '-.', ':', '', (offset, on-off-seq)}. See *`Line2D.set_linestyle`* for a complete description.

`set_linestyles`(*self*, *ls*)
Alias for *set_linestyle*.

`set_linewidth`(*self*, *lw*)
Set the linewidth(s) for the collection. *lw* can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence

Parameters

lw

[float or list of floats]

`set_linewidths`(*self*, *lw*)
Alias for *set_linewidth*.

`set_ls`(*self*, *ls*)
Alias for *set_linestyle*.

`set_lw`(*self*, *lw*)
Alias for *set_linewidth*.

`set_norm`(*self*, *norm*)
Set the normalization instance.

Parameters

norm

[*Normalize* or None]

Notes

If there are any colorbars using the mappable for this norm, setting the norm of the mappable will reset the norm, locator, and formatters on the colorbar to default.

`set_offset_position`(*self*, *offset_position*)
[*Deprecated*] Set how offsets are applied. If *offset_position* is 'screen' (default) the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is 'data', the offset is applied before the master transform, i.e., the offsets are in data coordinates.

Parameters

offset_position

[{'screen', 'data'}]

Notes

Deprecated since version 3.3.

`set_offsets`(*self*, *offsets*)
Set the offsets for the collection.

Parameters

offsets

[array-like (N, 2) or (2,)]

`set_path_effects`(*self*, *path_effects*)
Set the path effects.

Parameters

path_effects

[*AbstractPathEffect*]

`set_paths`(*self*)

`set_picker`(*self*, *picker*)
Define the picking behavior of the artist.

Parameters

picker

[None or bool or callable] This can be one of the following:

- *None*: Picking is disabled for this artist (default).
- A boolean: If *True* then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist.
- A function: If picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

to determine the hit test. if the mouse event is over the artist, return *hit=True* and props is a dictionary of properties you want added to the PickEvent attributes.

- *deprecated*: For *Line2D* only, *picker* can also be a float that sets the tolerance for checking whether an event occurred "on" the line; this is deprecated. Use *Line2D.set_pickradius* instead.

set_pickradius(*self*, *pr*)
Set the pick radius used for containment tests.

Parameters

d

[float] Pick radius, in points.

set_rasterized(*self*, *rasterized*)
Force rasterized (bitmap) drawing in vector backend output.

Defaults to None, which implies the backend's default behavior.

Parameters

rasterized

[bool or None]

set_sketch_params(*self*, *scale=None*, *length=None*, *randomness=None*)
Sets the sketch parameters.

Parameters

scale

[float, optional] The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is `None`, or not provided, no sketch filter will be provided.

length

[float, optional] The length of the wiggle along the line, in pixels (default 128.0)

randomness

[float, optional] The scale factor by which the length is shrunken or expanded (default 16.0)

set_snap(*self*, *snap*)
Set the snapping behavior.

Snapping aligns positions with the pixel grid, which results in clearer images. For example, if a black line of 1px width was defined at a position in between two pixels, the resulting image would contain the interpolated value of that line in the pixel grid, which would be a grey value on both adjacent pixel positions. In contrast, snapping will move the line to the nearest integer pixel value, so that the resulting image will really contain a 1px wide black line.

Snapping is currently only supported by the Agg and MacOSX backends.

Parameters

snap

[bool or None] Possible values:

- *True*: Snap vertices to the nearest pixel center.
- *False*: Do not modify vertex positions.
- *None*: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center.

`set_transform`(*self*, *t*)
Set the artist transform.

Parameters

t

[*Transform*]

`set_url`(*self*, *url*)
Set the url for the artist.

Parameters

url

[str]

`set_urls`(*self*, *urls*)

Parameters

urls

[list of str or None]

Notes

URLs are currently only implemented by the SVG backend. They are ignored by all other backends.

`set_visible`(*self*, *b*)
Set the artist's visibility.

Parameters

b

[bool]

`set_zorder`(*self*, *level*)

Set the zorder for the artist. Artists with lower zorder values are drawn first.

Parameters

level

[float]

`property stale`

Whether the artist is 'stale' and needs to be re-drawn for the output to match the internal state of the artist.

`property sticky_edges`

`x` and `y` sticky edge lists for autoscaling.

When performing autoscaling, if a data limit coincides with a value in the corresponding sticky_edges list, then no margin will be added--the view limit "sticks" to the edge. A typical use case is histograms, where one usually expects no margin on the bottom edge (0) of the histogram.

This attribute cannot be assigned to; however, the `x` and `y` lists can be modified in place as needed.

Examples

```
>>> artist.sticky_edges.x[:] = (xmin, xmax)
>>> artist.sticky_edges.y[:] = (ymin, ymax)
```

`to_rgba`(*self*, *x*, *alpha=None*, *bytes=False*, *norm=True*)

Return a normalized rgba array corresponding to *x*.

In the normal case, *x* is a 1-D or 2-D sequence of scalars, and the corresponding ndarray of rgba values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already rgb or rgba, such as might have been read from an image file. If *x* is an ndarray with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an rgb or rgba array, and no mapping will be done. The array can be uint8, or it can be floating point with values in the 0-1 range; otherwise a ValueError will be raised. If it is a masked array, the mask will be ignored. If the last dimension is 3, the *alpha* kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the *alpha* kwarg is ignored; it does not replace the pre-existing alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if *bytes* is *False* (default), the rgba array will be floats in the 0-1 range; if it is *True*, the returned rgba array will be uint8 in the 0 to 255 range.

If norm is False, no normalization of the input data is performed, and it is assumed to be in the range (0-1).

`update`(*self*, *props*)
Update this artist's properties from the dict *props*.

Parameters

props

[dict]

`property update_dict`

`update_from`(*self*, *other*)
Copy properties from other to self.

`update_scalarmappable`(*self*)
Update colors from the scalar mappable array, if it is not None.

`zorder = 0`

`class matplotlib.collections.EventCollection`(*positions*, *orientation='horizontal'*, *lineoffset=0*, *linelength=1*, *linewidth=None*, *color=None*, *linestyle='solid'*, *antialiased=None*, ***kwargs*)
Bases: *matplotlib.collections.LineCollection*

A collection of locations along a single axis at which an "event" occured.

The events are given by a 1-dimensional array. They do not have an amplitude and are displayed as parallel lines.

Parameters

positions

[1D array-like] Each value is an event.

orientation

[{'horizontal', 'vertical'}, default: 'horizontal'] The sequence of events is plotted along this direction. The marker lines of the single events are along the orthogonal direction.

lineoffset

[float, default: 0] The offset of the center of the markers from the origin, in the direction orthogonal to *orientation*.

linelength

[float, default: 1] The total height of the marker (i.e. the marker stretches from `lineoffset - linelength/2` to `lineoffset + linelength/2`).

linewidth

[float or list thereof, default: `rcParams["lines.linewidth"]` (default: `1.5`)] The line width of the event lines, in points.

color

[color or list of colors, default: `rcParams["lines.color"]` (default: `'C0'`)] The color of the event lines.

linestyle

[str or tuple or list thereof, default: 'solid'] Valid strings are ['solid', 'dashed', 'dashdot', 'dotted', '-', '--', '-.', ':']. Dash tuples should be of the form:

```
(offset, onoffseq),
```

where *onoffseq* is an even length tuple of on and off ink in points.

antialiased

[bool or list thereof, default: `rcParams["lines.antialiased"]` (default: `True`)] Whether to use antialiasing for drawing the lines.

****kwargs**

Forwarded to *LineCollection*.

Examples

`add_callback`(*self*, *func*)

Add a callback function that will be called whenever one of the *Artist*'s properties changes.

Parameters

func

[callable] The callback function. It must have the signature:

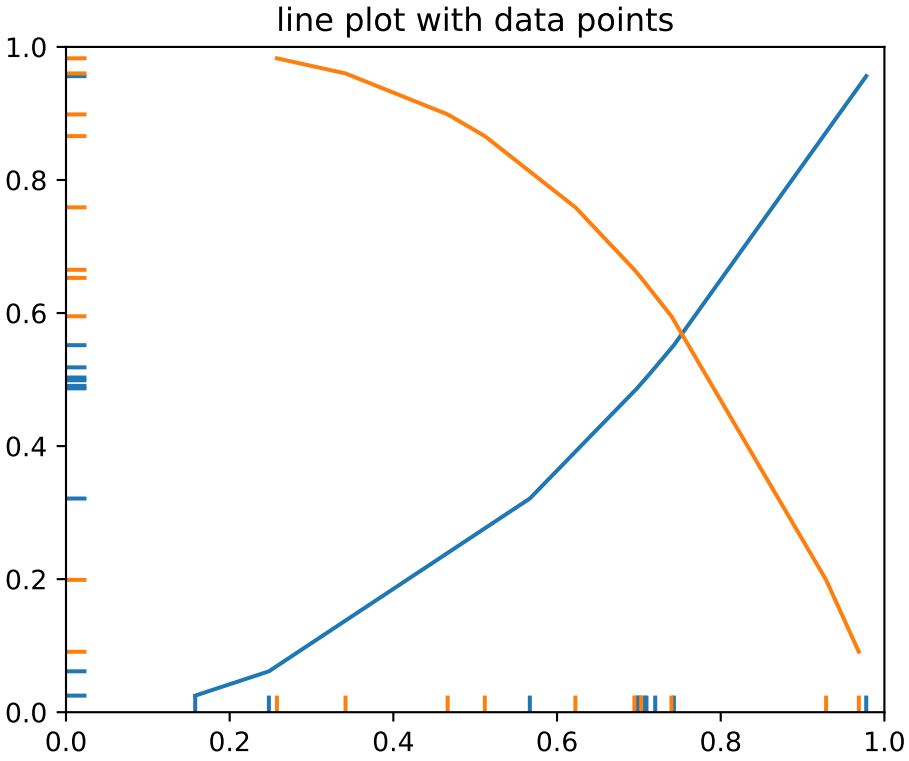
```
def func(artist: Artist) -> Any
```

where *artist* is the calling *Artist*. Return values may exist but are ignored.

Returns

int

The observer id associated with the callback. This id can be used for removing the callback with *remove_callback* later.



See also:

remove_callback

`add_checker`(*self*, *checker*)
[*Deprecated*]

Notes

Deprecated since version 3.3:

`add_positions`(*self*, *position*)
Add one or more events at the specified positions.

`append_positions`(*self*, *position*)
Add one or more events at the specified positions.

`autoscale`(*self*)
Autoscale the scalar limits on the norm instance using the current array

`autoscale_None`(*self*)
Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

`property axes`
The *Axes* instance the artist resides in, or *None*.

`changed`(*self*)
Call this whenever the mappable is changed to notify all the callbackSM listeners to the 'changed' signal.

`check_update`(*self*, *checker*)
[*Deprecated*]

Notes

Deprecated since version 3.3:

`contains`(*self*, *mouseevent*)
Test whether the mouse event occurred in the collection.

Returns `bool, dict(ind=itemlist)`, where every item in itemlist contains the event.

`convert_xunits`(*self*, *x*)
Convert *x* using the unit type of the xaxis.

If the artist is not in contained in an Axes or if the xaxis does not have units, *x* itself is returned.

`convert_yunits`(*self*, *y*)
Convert *y* using the unit type of the yaxis.

If the artist is not in contained in an Axes or if the yaxis does not have units, *y* itself is returned.

`draw`(*self*, *renderer*)
Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

Parameters

renderer

[*RendererBase* subclass.]

Notes

This method is overridden in the Artist subclasses.

`extend_positions`(*self*, *position*)
Add one or more events at the specified positions.

`findobj`(*self*, *match=None*, *include_self=True*)
Find artist objects.

Recursively find all *Artist* instances contained in the artist.

Parameters

match

A filter criterion for the matches. This can be

- *None*: Return all objects contained in artist.
- A function with signature `def match(artist: Artist) -> bool`. The result will only contain artists for which the function returns *True*.
- A class instance: e.g., *Line2D*. The result will only contain artists of this class or its subclasses (`isinstance` check).

include_self

[bool] Include *self* in the list to be checked for a match.

Returns

list of *Artist*

`format_cursor_data`(*self*, *data*)
Return a string representation of *data*.

Note: This method is intended to be overridden by artist subclasses. As an end-user of Matplotlib you will most likely not call this method yourself.

The default implementation converts ints and floats and arrays of ints and floats into a comma-separated string enclosed in square brackets.

See also:

get_cursor_data

`get_agg_filter`(*self*)
Return filter function to be used for agg filter.

`get_alpha`(*self*)
Return the alpha value used for blending - not supported on all backends.

`get_animated`(*self*)
Return whether the artist is animated.

`get_array`(*self*)
Return the data array.

`get_capstyle`(*self*)

`get_children`(*self*)
Return a list of the child *Artist*s of this *Artist*.

`get_clim`(*self*)
Return the values (min, max) that are mapped to the colormap limits.

`get_clip_box`(*self*)
Return the clipbox.

`get_clip_on`(*self*)
Return whether the artist uses clipping.

`get_clip_path`(*self*)
Return the clip path.

`get_cmap`(*self*)
Return the *Colormap* instance.

`get_color`(*self*)
Return the color of the lines used to mark each event.

`get_colors`(*self*)

`get_contains`(*self*)
[*Deprecated*] Return the custom contains function of the artist if set, or *None*.

See also:

set_contains

Notes

Deprecated since version 3.3.

`get_cursor_data`(*self*, *event*)
Return the cursor data for a given event.

Note: This method is intended to be overridden by artist subclasses. As an end-user of Matplotlib you will most likely not call this method yourself.

Cursor data can be used by Artists to provide additional context information for a given event. The default implementation just returns *None*.

Subclasses can override the method and return arbitrary data. However, when doing so, they must ensure that *format_cursor_data* can convert the data to a string representation.

The only current use case is displaying the z-value of an *AxesImage* in the status bar of a plot window, while moving the mouse.

Parameters

event

[*matplotlib.backend_bases.MouseEvent*]

See also:

format_cursor_data

`get_dashes`(*self*)
Alias for *get_linestyle*.

`get_datalim`(*self*, *transData*)

`get_ec`(*self*)
Alias for *get_edgecolor*.

`get_edgecolor`(*self*)

`get_edgecolors`(*self*)
Alias for *get_edgecolor*.

`get_facecolor`(*self*)

`get_facecolors`(*self*)
Alias for *get_facecolor*.

`get_fc`(*self*)
Alias for *get_facecolor*.

`get_figure`(*self*)
Return the *Figure* instance the artist belongs to.

`get_fill`(*self*)
Return whether fill is set.

`get_gid`(*self*)
Return the group id.

`get_hatch`(*self*)
Return the current hatching pattern.

`get_in_layout`(*self*)
Return boolean flag, `True` if artist is included in layout calculations.

E.g. *Constrained Layout Guide*, *Figure.tight_layout()*, and `fig.savefig(fname, bbox_inches='tight')`.

`get_joinstyle`(*self*)

`get_label`(*self*)
Return the label used for this artist in the legend.

`get_linelength`(*self*)
Return the length of the lines used to mark each event.

`get_lineoffset`(*self*)
Return the offset of the lines used to mark each event.

`get_linestyle`(*self*)

`get_linestyles`(*self*)
Alias for *get_linestyle*.

`get_linewidth`(*self*)
Get the width of the lines used to mark each event.

`get_linewidths`(*self*)
Alias for *get_linewidth*.

`get_ls`(*self*)
Alias for *get_linestyle*.

`get_lw`(*self*)
Alias for *get_linewidth*.

`get_offset_position`(*self*)
[*Deprecated*] Return how offsets are applied for the collection. If *offset_position* is 'screen', the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is 'data', the offset is applied before the master transform, i.e., the offsets are in data coordinates.

Notes

Deprecated since version 3.3.

`get_offset_transform`(*self*)

`get_offsets`(*self*)
Return the offsets for the collection.

`get_orientation`(*self*)
Return the orientation of the event line ('horizontal' or 'vertical').

`get_path_effects`(*self*)

`get_paths`(*self*)

get_picker(*self*)
Return the picking behavior of the artist.

The possible values are described in *set_picker*.

See also:

set_picker, *pickable*, *pick*

get_pickradius(*self*)

get_positions(*self*)
Return an array containing the floating-point values of the positions.

get_rasterized(*self*)
Return whether the artist is to be rasterized.

get_segments(*self*)

Returns

list

List of segments in the LineCollection. Each list item contains an array of vertices.

get_sketch_params(*self*)
Return the sketch parameters for the artist.

Returns

tuple or None

A 3-tuple with the following elements:

- *scale*: The amplitude of the wiggle perpendicular to the source line.
- *length*: The length of the wiggle along the line.
- *randomness*: The scale factor by which the length is shrunken or expanded.

Returns *None* if no sketch parameters were set.

get_snap(*self*)
Return the snap setting.

See *set_snap* for details.

get_tightbbox(*self*, *renderer*)
Like *Artist.get_window_extent*, but includes any clipping.

Parameters

renderer

[*RendererBase* subclass] renderer that will be used to draw the figures (i.e. `fig.canvas.get_renderer()`)

Returns

Bbox

The enclosing bounding box (in figure pixel coordinates).

`get_transform`(*self*)
Return the *Transform* instance used by this artist.

`get_transformed_clip_path_and_affine`(*self*)
Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

`get_transforms`(*self*)

`get_url`(*self*)
Return the url.

`get_urls`(*self*)
Return a list of URLs, one for each element of the collection.

The list contains *None* for elements without a URL. See /gallery/misc/hyperlinks_sgskip for an example.

`get_visible`(*self*)
Return the visibility.

`get_window_extent`(*self*, *renderer*)
Get the axes bounding box in display space.

The bounding box' width and height are nonnegative.

Subclasses should override for inclusion in the bounding box "tight" calculation. Default is to return an empty bounding box at 0, 0.

Be careful when using this function, the results will not update if the artist window extent of the artist changes. The extent can change due to any changes in the transform stack, such as changing the axes limits, the figure size, or the canvas used (as is done when saving a figure). This can lead to unexpected behavior where interactive figures will look fine on the screen, but will save incorrectly.

`get_zorder`(*self*)
Return the artist's zorder.

`have_units`(*self*)
Return *True* if units are set on any axis.

`is_horizontal`(*self*)
True if the eventcollection is horizontal, False if vertical.

`is_transform_set`(*self*)
Return whether the Artist has an explicitly set transform.

This is *True* after *set_transform* has been called.

`property mouseover`
If this property is set to *True*, the artist will be queried for custom context information when the mouse cursor moves over it.

See also *get_cursor_data()*, *ToolCursorPosition* and *NavigationToolbar2*.

`pchanged`(*self*)
Call all of the registered callbacks.

This function is triggered internally when a property is changed.

See also:

add_callback

remove_callback

`pick`(*self*, *mouseevent*)
Process a pick event.

Each child artist will fire a pick event if *mouseevent* is over the artist and the artist has picker set.

See also:

set_picker, *get_picker*, *pickable*

`pickable`(*self*)
Return whether the artist is pickable.

See also:

set_picker, *get_picker*, *pick*

`properties`(*self*)
Return a dictionary of all the properties of the artist.

`remove`(*self*)
Remove the artist from the figure if possible.

The effect will not be visible until the figure is redrawn, e.g., with *FigureCanvasBase.draw_idle*. Call *relim* to update the axes limits if desired.

Note: *relim* will not see collections even if the collection was added to the axes with *autolim* = True.

Note: there is no support for removing the artist's legend entry.

`remove_callback`(*self*, *oid*)
Remove a callback based on its observer id.

See also:

add_callback

`set`(*self*, ***kwargs*)
A property batch setter. Pass *kwargs* to set properties.

`set_aa`(*self*, *aa*)
Alias for *set_antialiased*.

`set_agg_filter`(*self*, *filter_func*)
Set the agg filter.

Parameters

filter_func

[callable] A filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array.

`set_alpha`(*self*, *alpha*)
Set the alpha value used for blending - not supported on all backends.

Parameters

alpha

[float or None]

`set_animated`(*self*, *b*)
Set the artist's animation state.

Parameters

b

[bool]

`set_antialiased`(*self*, *aa*)
Set the antialiasing state for rendering.

Parameters

aa

[bool or list of bools]

`set_antialiaseds`(*self*, *aa*)
Alias for *set_antialiased*.

`set_array`(*self*, *A*)
Set the image array from numpy array *A*.

Parameters

A

[ndarray]

`set_capstyle`(*self*, *cs*)
Set the capstyle for the collection (for all its elements).

Parameters

cs

[{'butt', 'round', 'projecting'}] The capstyle.

`set_clim`(*self*, *vmin=None*, *vmax=None*)
Set the norm limits for image scaling.

Parameters

vmin, vmax

[float] The limits.

The limits may also be passed as a tuple (*vmin*, *vmax*) as a single positional argument.

`set_clip_box`(*self*, *clipbox*)
Set the artist's clip *Bbox*.

Parameters

clipbox

[*Bbox*]

`set_clip_on`(*self*, *b*)
Set whether the artist uses clipping.

When False artists will be visible outside of the axes which can lead to unexpected results.

Parameters

b

[bool]

`set_clip_path`(*self*, *path*, *transform=None*)
Set the artist's clip path.

Parameters

path

[*Patch* or *Path* or *TransformedPath* or None] The clip path. If given a *Path*, *transform* must be provided as well. If *None*, a previously set clip path is removed.

transform

[*Transform*, optional] Only used if *path* is a *Path*, in which case the given *Path* is converted to a *TransformedPath* using *transform*.

Notes

For efficiency, if *path* is a *Rectangle* this method will set the clipping box to the corresponding rectangle and set the clipping path to `None`.

For technical reasons (support of *set*), a tuple (*path*, *transform*) is also accepted as a single positional parameter.

`set_cmap`(*self*, *cmap*)
Set the colormap for luminance data.

Parameters

cmap

[*Colormap* or str or None]

`set_color`(*self*, *c*)
Set the color(s) of the LineCollection.

Parameters

c

[color or list of colors] Single color (all patches have same color), or a sequence of rgba tuples; if it is a sequence the patches will cycle through the sequence.

`set_contains`(*self*, *picker*)
[*Deprecated*] Define a custom contains test for the artist.

The provided callable replaces the default *contains* method of the artist.

Parameters

picker

[callable] A custom picker function to evaluate if an event is within the artist. The function must have the signature:

```
def contains(artist: Artist, event: MouseEvent) -> bool, dict
```

that returns:

- a bool indicating if the event is within the artist
- a dict of additional information. The dict should at least return the same information as the default `contains()` implementation of the respective artist, but may provide additional information.

Notes

Deprecated since version 3.3.

`set_dashes`(*self*, *ls*)
Alias for *set_linestyle*.

`set_ec`(*self*, *c*)
Alias for *set_edgecolor*.

`set_edgecolor`(*self*, *c*)
Set the edgecolor(s) of the collection.

Parameters

c

[color or list of colors or 'face'] The collection edgecolor(s). If a sequence, the patches cycle through it. If 'face', match the facecolor.

`set_edgecolors`(*self*, *c*)
Alias for *set_edgecolor*.

`set_facecolor`(*self*, *c*)
Set the facecolor(s) of the collection. *c* can be a color (all patches have same color), or a sequence of colors; if it is a sequence the patches will cycle through the sequence.

If *c* is 'none', the patch will not be filled.

Parameters

c

[color or list of colors]

`set_facecolors`(*self*, *c*)
Alias for *set_facecolor*.

set_fc(*self*, *c*)
Alias for *set_facecolor*.

set_figure(*self*, *fig*)
Set the *Figure* instance the artist belongs to.

Parameters

fig

[*Figure*]

set_gid(*self*, *gid*)
Set the (group) id for the artist.

Parameters

gid

[str]

set_hatch(*self*, *hatch*)
Set the hatching pattern

hatch can be one of:

```
/   - diagonal hatching
\   - back diagonal
|   - vertical
-   - horizontal
+   - crossed
x   - crossed diagonal
o   - small circle
O   - large circle
.   - dots
*   - stars
```

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Unlike other properties such as linewidth and colors, hatching can only be specified for the collection as a whole, not separately for each member.

Parameters

hatch

[{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}]

set_in_layout(*self*, *in_layout*)
Set if artist is to be included in layout calculations, E.g. *Constrained Layout Guide*, *Figure.tight_layout()*, and `fig.savefig(fname, bbox_inches='tight')`.

Parameters

in_layout

[bool]

`set_joinstyle`(*self*, *js*)
Set the joinstyle for the collection (for all its elements).

Parameters

js

[{'miter', 'round', 'bevel'}] The joinstyle.

`set_label`(*self*, *s*)
Set a label that will be displayed in the legend.

Parameters

s

[object] *s* will be converted to a string by calling `str`.

`set_linelength`(*self*, *linelength*)
Set the length of the lines used to mark each event.

`set_lineoffset`(*self*, *lineoffset*)
Set the offset of the lines used to mark each event.

`set_linestyle`(*self*, *ls*)
Set the linestyle(s) for the collection.

linestyle	description
`'-'` or `'solid'`	solid line
`'--'` or `'dashed'`	dashed line
`'-.'` or `'dashdot'`	dash-dotted line
`':'` or `'dotted'`	dotted line

Alternatively a dash tuple of the following form can be provided:

```
(offset, onoffseq),
```

where `onoffseq` is an even length tuple of on and off ink in points.

Parameters

ls

[str or tuple or list thereof] Valid values for individual linestyles include {'-', '--', '-.', ':', '', (offset, on-off-seq)}. See *Line2D.set_linestyle* for a complete description.

`set_linestyles`(*self*, *ls*)
Alias for *set_linestyle*.

`set_linewidth`(*self*, *lw*)
Set the linewidth(s) for the collection. *lw* can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence

Parameters

lw

[float or list of floats]

`set_linewidths`(*self*, *lw*)
Alias for *set_linewidth*.

`set_ls`(*self*, *ls*)
Alias for *set_linestyle*.

`set_lw`(*self*, *lw*)
Alias for *set_linewidth*.

`set_norm`(*self*, *norm*)
Set the normalization instance.

Parameters

norm

[*Normalize* or None]

Notes

If there are any colorbars using the mappable for this norm, setting the norm of the mappable will reset the norm, locator, and formatters on the colorbar to default.

`set_offset_position`(*self*, *offset_position*)
[*Deprecated*] Set how offsets are applied. If *offset_position* is 'screen' (default) the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is 'data', the offset is applied before the master transform, i.e., the offsets are in data coordinates.

Parameters

offset_position

[{'screen', 'data'}]

Notes

Deprecated since version 3.3.

`set_offsets`(*self*, *offsets*)
Set the offsets for the collection.

Parameters

offsets

[array-like (N, 2) or (2,)]

`set_orientation`(*self*, *orientation=None*)
Set the orientation of the event line.

Parameters

orientation

[{'horizontal', 'vertical'}]

`set_path_effects`(*self*, *path_effects*)
Set the path effects.

Parameters

path_effects

[*AbstractPathEffect*]

`set_paths`(*self*, *segments*)

`set_picker`(*self*, *picker*)
Define the picking behavior of the artist.

Parameters

picker

[None or bool or callable] This can be one of the following:

- *None*: Picking is disabled for this artist (default).
- A boolean: If *True* then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist.
- A function: If picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

to determine the hit test. if the mouse event is over the artist, return *hit=True* and props is a dictionary of properties you want added to the PickEvent attributes.

- *deprecated*: For *Line2D* only, *picker* can also be a float that sets the tolerance for checking whether an event occurred "on" the line; this is deprecated. Use *Line2D.set_pickradius* instead.

`set_pickradius`(*self*, *pr*)
Set the pick radius used for containment tests.

Parameters

d

[float] Pick radius, in points.

`set_positions`(*self*, *positions*)
Set the positions of the events.

`set_rasterized`(*self*, *rasterized*)
Force rasterized (bitmap) drawing in vector backend output.

Defaults to None, which implies the backend's default behavior.

Parameters

rasterized

[bool or None]

`set_segments`(*self*, *segments*)

`set_sketch_params`(*self*, *scale=None*, *length=None*, *randomness=None*)
Sets the sketch parameters.

Parameters

scale

[float, optional] The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is `None`, or not provided, no sketch filter will be provided.

length

[float, optional] The length of the wiggle along the line, in pixels (default 128.0)

randomness

[float, optional] The scale factor by which the length is shrunken or expanded (default 16.0)

set_snap(*self*, *snap*)
Set the snapping behavior.

Snapping aligns positions with the pixel grid, which results in clearer images. For example, if a black line of 1px width was defined at a position in between two pixels, the resulting image would contain the interpolated value of that line in the pixel grid, which would be a grey value on both adjacent pixel positions. In contrast, snapping will move the line to the nearest integer pixel value, so that the resulting image will really contain a 1px wide black line.

Snapping is currently only supported by the Agg and MacOSX backends.

Parameters

snap

[bool or None] Possible values:

- *True*: Snap vertices to the nearest pixel center.
- *False*: Do not modify vertex positions.
- *None*: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center.

set_transform(*self*, *t*)
Set the artist transform.

Parameters

t

[*Transform*]

set_url(*self*, *url*)
Set the url for the artist.

Parameters

url

[str]

set_urls(*self*, *urls*)

Parameters

urls

[list of str or None]

Notes

URLs are currently only implemented by the SVG backend. They are ignored by all other backends.

`set_verts`(*self*, *segments*)

`set_visible`(*self*, *b*)
Set the artist's visibility.

Parameters

b

[bool]

`set_zorder`(*self*, *level*)
Set the zorder for the artist. Artists with lower zorder values are drawn first.

Parameters

level

[float]

`property stale`
Whether the artist is 'stale' and needs to be re-drawn for the output to match the internal state of the artist.

`property sticky_edges`
`x` and `y` sticky edge lists for autoscaling.

When performing autoscaling, if a data limit coincides with a value in the corresponding sticky_edges list, then no margin will be added--the view limit "sticks" to the edge. A typical use case is histograms, where one usually expects no margin on the bottom edge (0) of the histogram.

This attribute cannot be assigned to; however, the `x` and `y` lists can be modified in place as needed.

Examples

```
>>> artist.sticky_edges.x[:] = (xmin, xmax)
>>> artist.sticky_edges.y[:] = (ymin, ymax)
```

`switch_orientation`(*self*)
Switch the orientation of the event line, either from vertical to horizontal or vice versus.

to_rgba(*self*, *x*, *alpha=None*, *bytes=False*, *norm=True*)

Return a normalized rgba array corresponding to *x*.

In the normal case, *x* is a 1-D or 2-D sequence of scalars, and the corresponding ndarray of rgba values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already rgb or rgba, such as might have been read from an image file. If *x* is an ndarray with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an rgb or rgba array, and no mapping will be done. The array can be uint8, or it can be floating point with values in the 0-1 range; otherwise a ValueError will be raised. If it is a masked array, the mask will be ignored. If the last dimension is 3, the *alpha* kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the *alpha* kwarg is ignored; it does not replace the pre-existing alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if *bytes* is *False* (default), the rgba array will be floats in the 0-1 range; if it is *True*, the returned rgba array will be uint8 in the 0 to 255 range.

If norm is False, no normalization of the input data is performed, and it is assumed to be in the range (0-1).

update(*self*, *props*)

Update this artist's properties from the dict *props*.

Parameters

props

[dict]

property update_dict

update_from(*self*, *other*)

Copy properties from other to self.

update_scalarmappable(*self*)

Update colors from the scalar mappable array, if it is not None.

zorder = 0

class matplotlib.collections.LineCollection(*segments*, *linewidths=None*, *colors=None*, *antialiaseds=None*, *linestyles='solid'*, *offsets=None*, *transOffset=None*, *norm=None*, *cmap=None*, *pickradius=5*, *zorder=2*, *facecolors='none'*, ***kwargs*)

Bases: *matplotlib.collections.Collection*

Represents a sequence of *Line2D*s that should be drawn together.

This class extends *Collection* to represent a sequence of *Line2D*s instead of just a sequence of *Patch*s. Just as in *Collection*, each property of a *LineCollection* may be either a single value or a list of values. This list is then used cyclically for each element of the LineCollection, so the property of the `i`th element of the collection is:

```
prop[i % len(prop)]
```

The properties of each member of a *LineCollection* default to their values in `rcParams["lines.*"]` instead of `rcParams["patch.*"]`, and the property *colors* is added in place of *edgecolors*.

Parameters

segments: list of array-like

A sequence of (*line0*, *line1*, *line2*), where:

```
linen = (x0, y0), (x1, y1), ... (xm, ym)
```

or the equivalent numpy array with two columns. Each line can have a different number of segments.

linewidths

[float or list of float, default: `rcParams["lines.linewidth"]` (default: `1.5`)] The width of each line in points.

colors

[color or list of color, default: `rcParams["lines.color"]` (default: `'C0'`)] A sequence of RGBA tuples (e.g., arbitrary color strings, etc, not allowed).

antialiaseds

[bool or list of bool, default: `rcParams["lines.antialiased"]` (default: `True`)] Whether to use antialiasing for each line.

zorder

[int, default: 2] zorder of the lines once drawn.

facecolors

[color or list of color, default: 'none'] The facecolors of the LineCollection. Setting to a value other than 'none' will lead to each line being "filled in" as if there was an implicit line segment joining the last and first points of that line back around to each other. In order to manually specify what should count as the "interior" of each line, please use *PathCollection* instead, where the "interior" can be specified by appropriate usage of *CLOSEPOLY*.

****kwargs**

Forwareded to *Collection*.

`add_callback`(*self*, *func*)
Add a callback function that will be called whenever one of the *Artist*'s properties changes.

Parameters

func

[callable] The callback function. It must have the signature:

```
def func(artist: Artist) -> Any
```

where *artist* is the calling *Artist*. Return values may exist but are ignored.

Returns

int

The observer id associated with the callback. This id can be used for removing the callback with *remove_callback* later.

See also:

remove_callback

`add_checker`(*self*, *checker*)
[*Deprecated*]

Notes

Deprecated since version 3.3:

`autoscale`(*self*)
Autoscale the scalar limits on the norm instance using the current array

`autoscale_None`(*self*)
Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

`property axes`
The *Axes* instance the artist resides in, or *None*.

`changed`(*self*)
Call this whenever the mappable is changed to notify all the callbackSM listeners to the 'changed' signal.

`check_update`(*self*, *checker*)
[*Deprecated*]

Notes

Deprecated since version 3.3:

`contains`(*self*, *mouseevent*)
Test whether the mouse event occurred in the collection.

Returns `bool, dict(ind=itemlist)`, where every item in itemlist contains the event.

`convert_xunits`(*self*, *x*)
Convert *x* using the unit type of the xaxis.

If the artist is not in contained in an Axes or if the xaxis does not have units, *x* itself is returned.

`convert_yunits`(*self*, *y*)
Convert *y* using the unit type of the yaxis.

If the artist is not in contained in an Axes or if the yaxis does not have units, *y* itself is returned.

`draw`(*self*, *renderer*)
Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

Parameters

renderer

[*RendererBase* subclass.]

Notes

This method is overridden in the Artist subclasses.

`findobj`(*self*, *match=None*, *include_self=True*)
Find artist objects.

Recursively find all *Artist* instances contained in the artist.

Parameters

match

A filter criterion for the matches. This can be

- *None*: Return all objects contained in artist.
- A function with signature `def match(artist: Artist) -> bool`. The result will only contain artists for which the function returns *True*.

- A class instance: e.g., *Line2D*. The result will only contain artists of this class or its subclasses (`isinstance` check).

include_self

[bool] Include *self* in the list to be checked for a match.

Returns

list of *Artist*

`format_cursor_data`(*self*, *data*)
Return a string representation of *data*.

Note: This method is intended to be overridden by artist subclasses. As an end-user of Matplotlib you will most likely not call this method yourself.

The default implementation converts ints and floats and arrays of ints and floats into a comma-separated string enclosed in square brackets.

See also:

get_cursor_data

`get_agg_filter`(*self*)
Return filter function to be used for agg filter.

`get_alpha`(*self*)
Return the alpha value used for blending - not supported on all backends.

`get_animated`(*self*)
Return whether the artist is animated.

`get_array`(*self*)
Return the data array.

`get_capstyle`(*self*)

`get_children`(*self*)
Return a list of the child *Artist*s of this *Artist*.

`get_clim`(*self*)
Return the values (min, max) that are mapped to the colormap limits.

`get_clip_box`(*self*)
Return the clipbox.

`get_clip_on`(*self*)
Return whether the artist uses clipping.

`get_clip_path`(*self*)
Return the clip path.

get_cmap(*self*)
Return the *Colormap* instance.

get_color(*self*)

get_colors(*self*)

get_contains(*self*)
[*Deprecated*] Return the custom contains function of the artist if set, or *None*.

See also:

set_contains

Notes

Deprecated since version 3.3.

get_cursor_data(*self*, *event*)
Return the cursor data for a given event.

Note: This method is intended to be overridden by artist subclasses. As an end-user of Matplotlib you will most likely not call this method yourself.

Cursor data can be used by Artists to provide additional context information for a given event. The default implementation just returns *None*.

Subclasses can override the method and return arbitrary data. However, when doing so, they must ensure that *format_cursor_data* can convert the data to a string representation.

The only current use case is displaying the z-value of an *AxesImage* in the status bar of a plot window, while moving the mouse.

Parameters

event

[*matplotlib.backend_bases.MouseEvent*]

See also:

format_cursor_data

get_dashes(*self*)
Alias for *get_linestyle*.

get_datalim(*self*, *transData*)

get_ec(*self*)
Alias for *get_edgecolor*.

get_edgecolor(*self*)

get_edgecolors(*self*)
Alias for *get_edgecolor*.

get_facecolor(*self*)

get_facecolors(*self*)
Alias for *get_facecolor*.

get_fc(*self*)
Alias for *get_facecolor*.

get_figure(*self*)
Return the *Figure* instance the artist belongs to.

get_fill(*self*)
Return whether fill is set.

get_gid(*self*)
Return the group id.

get_hatch(*self*)
Return the current hatching pattern.

get_in_layout(*self*)
Return boolean flag, `True` if artist is included in layout calculations.

E.g. *Constrained Layout Guide*, *Figure.tight_layout()*, and `fig.savefig(fname, bbox_inches='tight')`.

get_joinstyle(*self*)

get_label(*self*)
Return the label used for this artist in the legend.

get_linestyle(*self*)

get_linestyles(*self*)
Alias for *get_linestyle*.

get_linewidth(*self*)

get_linewidths(*self*)
Alias for *get_linewidth*.

get_ls(*self*)
Alias for *get_linestyle*.

get_lw(*self*)
Alias for *get_linewidth*.

get_offset_position(*self*)
[*Deprecated*] Return how offsets are applied for the collection. If *offset_position* is 'screen', the offset is applied after the master transform has

been applied, that is, the offsets are in screen coordinates. If offset_position is 'data', the offset is applied before the master transform, i.e., the offsets are in data coordinates.

Notes

Deprecated since version 3.3.

`get_offset_transform`(*self*)

`get_offsets`(*self*)
Return the offsets for the collection.

`get_path_effects`(*self*)

`get_paths`(*self*)

`get_picker`(*self*)
Return the picking behavior of the artist.

The possible values are described in *set_picker*.

See also:

set_picker, *pickable*, *pick*

`get_pickradius`(*self*)

`get_rasterized`(*self*)
Return whether the artist is to be rasterized.

`get_segments`(*self*)

Returns

list

List of segments in the LineCollection. Each list item contains an array of vertices.

`get_sketch_params`(*self*)
Return the sketch parameters for the artist.

Returns

tuple or None

A 3-tuple with the following elements:

- *scale*: The amplitude of the wiggle perpendicular to the source line.
- *length*: The length of the wiggle along the line.

- *randomness*: The scale factor by which the length is shrunken or expanded.

Returns *None* if no sketch parameters were set.

`get_snap`(*self*)
Return the snap setting.

See *set_snap* for details.

`get_tightbbox`(*self*, *renderer*)
Like *Artist.get_window_extent*, but includes any clipping.

Parameters

renderer

[*RendererBase* subclass] renderer that will be used to draw the figures (i.e. `fig.canvas.get_renderer()`)

Returns

Bbox

The enclosing bounding box (in figure pixel coordinates).

`get_transform`(*self*)
Return the *Transform* instance used by this artist.

`get_transformed_clip_path_and_affine`(*self*)
Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

`get_transforms`(*self*)

`get_url`(*self*)
Return the url.

`get_urls`(*self*)
Return a list of URLs, one for each element of the collection.

The list contains *None* for elements without a URL. See /gallery/misc/hyperlinks_sgskip for an example.

`get_visible`(*self*)
Return the visibility.

`get_window_extent`(*self*, *renderer*)
Get the axes bounding box in display space.

The bounding box' width and height are nonnegative.

Subclasses should override for inclusion in the bounding box "tight" calculation. Default is to return an empty bounding box at 0, 0.

Be careful when using this function, the results will not update if the artist window extent of the artist changes. The extent can change due to any changes in the transform stack, such as changing the axes limits, the figure size, or the canvas used (as is done when saving a figure). This can lead to unexpected behavior where interactive figures will look fine on the screen, but will save incorrectly.

`get_zorder`(*self*)
Return the artist's zorder.

`have_units`(*self*)
Return *True* if units are set on any axis.

`is_transform_set`(*self*)
Return whether the Artist has an explicitly set transform.

This is *True* after *set_transform* has been called.

`property mouseover`
If this property is set to *True*, the artist will be queried for custom context information when the mouse cursor moves over it.

See also *get_cursor_data()*, *ToolCursorPosition* and *NavigationToolbar2*.

`pchanged`(*self*)
Call all of the registered callbacks.

This function is triggered internally when a property is changed.

See also:

add_callback

remove_callback

`pick`(*self*, *mouseevent*)
Process a pick event.

Each child artist will fire a pick event if *mouseevent* is over the artist and the artist has picker set.

See also:

set_picker, *get_picker*, *pickable*

`pickable`(*self*)
Return whether the artist is pickable.

See also:

set_picker, *get_picker*, *pick*

`properties`(*self*)
Return a dictionary of all the properties of the artist.

`remove`(*self*)
Remove the artist from the figure if possible.

The effect will not be visible until the figure is redrawn, e.g., with *FigureCanvasBase.draw_idle*. Call *relim* to update the axes limits if desired.

Note: *relim* will not see collections even if the collection was added to the axes with *autolim* = True.

Note: there is no support for removing the artist's legend entry.

`remove_callback`(*self*, *oid*)
Remove a callback based on its observer id.

See also:

add_callback

`set`(*self*, ***kwargs*)
A property batch setter. Pass *kwargs* to set properties.

`set_aa`(*self*, *aa*)
Alias for *set_antialiased*.

`set_agg_filter`(*self*, *filter_func*)
Set the agg filter.

Parameters

filter_func

[callable] A filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array.

`set_alpha`(*self*, *alpha*)
Set the alpha value used for blending - not supported on all backends.

Parameters

alpha

[float or None]

`set_animated`(*self*, *b*)
Set the artist's animation state.

Parameters

b

[bool]

`set_antialiased`(*self*, *aa*)
Set the antialiasing state for rendering.

Parameters

aa

[bool or list of bools]

set_antialiaseds(*self*, *aa*)
Alias for *set_antialiased*.

set_array(*self*, *A*)
Set the image array from numpy array *A*.

Parameters

A

[ndarray]

set_capstyle(*self*, *cs*)
Set the capstyle for the collection (for all its elements).

Parameters

cs

[{'butt', 'round', 'projecting'}] The capstyle.

set_clim(*self*, *vmin=None*, *vmax=None*)
Set the norm limits for image scaling.

Parameters

vmin, vmax

[float] The limits.

The limits may also be passed as a tuple (*vmin*, *vmax*) as a single positional argument.

set_clip_box(*self*, *clipbox*)
Set the artist's clip *Bbox*.

Parameters

clipbox

[*Bbox*]

set_clip_on(*self*, *b*)
Set whether the artist uses clipping.

When False artists will be visible outside of the axes which can lead to unexpected results.

Parameters

b

[bool]

`set_clip_path`(*self*, *path*, *transform=None*)
Set the artist's clip path.

Parameters

path

[*Patch* or *Path* or *TransformedPath* or None] The clip path. If given a *Path*, *transform* must be provided as well. If *None*, a previously set clip path is removed.

transform

[*Transform*, optional] Only used if *path* is a *Path*, in which case the given *Path* is converted to a *TransformedPath* using *transform*.

Notes

For efficiency, if *path* is a *Rectangle* this method will set the clipping box to the corresponding rectangle and set the clipping path to `None`.

For technical reasons (support of *set*), a tuple (*path*, *transform*) is also accepted as a single positional parameter.

`set_cmap`(*self*, *cmap*)
Set the colormap for luminance data.

Parameters

cmap

[*Colormap* or str or None]

`set_color`(*self*, *c*)
Set the color(s) of the LineCollection.

Parameters

c

[color or list of colors] Single color (all patches have same color), or a sequence of rgba tuples; if it is a sequence the patches will cycle through the sequence.

set_contains(*self*, *picker*)
[*Deprecated*] Define a custom contains test for the artist.

The provided callable replaces the default *contains* method of the artist.

Parameters

picker

[callable] A custom picker function to evaluate if an event is within the artist. The function must have the signature:

```
def contains(artist: Artist, event: MouseEvent) -> bool, dict
```

that returns:

- a bool indicating if the event is within the artist
- a dict of additional information. The dict should at least return the same information as the default `contains()` implementation of the respective artist, but may provide additional information.

Notes

Deprecated since version 3.3.

set_dashes(*self*, *ls*)
Alias for *set_linestyle*.

set_ec(*self*, *c*)
Alias for *set_edgecolor*.

set_edgecolor(*self*, *c*)
Set the edgecolor(s) of the collection.

Parameters

c

[color or list of colors or 'face'] The collection edgecolor(s). If a sequence, the patches cycle through it. If 'face', match the facecolor.

set_edgecolors(*self*, *c*)
Alias for *set_edgecolor*.

set_facecolor(*self*, *c*)
Set the facecolor(s) of the collection. *c* can be a color (all patches have same color), or a sequence of colors; if it is a sequence the patches will cycle through the sequence.

If *c* is 'none', the patch will not be filled.

Parameters

c

[color or list of colors]

`set_facecolors`(*self*, *c*)
Alias for *set_facecolor*.

`set_fc`(*self*, *c*)
Alias for *set_facecolor*.

`set_figure`(*self*, *fig*)
Set the *Figure* instance the artist belongs to.

Parameters

fig

[*Figure*]

`set_gid`(*self*, *gid*)
Set the (group) id for the artist.

Parameters

gid

[str]

`set_hatch`(*self*, *hatch*)
Set the hatching pattern

hatch can be one of:

```
/   - diagonal hatching
\   - back diagonal
|   - vertical
-   - horizontal
+   - crossed
x   - crossed diagonal
o   - small circle
O   - large circle
.   - dots
*   - stars
```

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Unlike other properties such as linewidth and colors, hatching can only be specified for the collection as a whole, not separately for each member.

Parameters

hatch

[{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}]

`set_in_layout`(*self*, *in_layout*)

Set if artist is to be included in layout calculations, E.g. *Constrained Layout Guide*, *`Figure.tight_layout()`*, and `fig.savefig(fname, bbox_inches='tight')`.

Parameters

in_layout

[bool]

`set_joinstyle`(*self*, *js*)

Set the joinstyle for the collection (for all its elements).

Parameters

js

[{'miter', 'round', 'bevel'}] The joinstyle.

`set_label`(*self*, *s*)

Set a label that will be displayed in the legend.

Parameters

s

[object] *s* will be converted to a string by calling `str`.

`set_linestyle`(*self*, *ls*)

Set the linestyle(s) for the collection.

linestyle	description
`'-'` or `'solid'`	solid line
`'--'` or `'dashed'`	dashed line
`'-.'` or `'dashdot'`	dash-dotted line
`':'` or `'dotted'`	dotted line

Alternatively a dash tuple of the following form can be provided:

```
(offset, onoffseq),
```

where `onoffseq` is an even length tuple of on and off ink in points.

Parameters

ls

[str or tuple or list thereof] Valid values for individual linestyles include {'-', '--', '-.', ':', '', (offset, on-off-seq)}. See *Line2D.set_linestyle* for a complete description.

`set_linestyles`(*self*, *ls*)
Alias for *set_linestyle*.

`set_linewidth`(*self*, *lw*)
Set the linewidth(s) for the collection. *lw* can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence

Parameters

lw

[float or list of floats]

`set_linewidths`(*self*, *lw*)
Alias for *set_linewidth*.

`set_ls`(*self*, *ls*)
Alias for *set_linestyle*.

`set_lw`(*self*, *lw*)
Alias for *set_linewidth*.

`set_norm`(*self*, *norm*)
Set the normalization instance.

Parameters

norm

[*Normalize* or None]

Notes

If there are any colorbars using the mappable for this norm, setting the norm of the mappable will reset the norm, locator, and formatters on the colorbar to default.

`set_offset_position`(*self*, *offset_position*)
[*Deprecated*] Set how offsets are applied. If *offset_position* is 'screen' (default) the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is 'data', the offset is applied before the master transform, i.e., the offsets are in data coordinates.

Parameters

offset_position

[{'screen', 'data'}]

Notes

Deprecated since version 3.3.

`set_offsets`(*self*, *offsets*)
Set the offsets for the collection.

Parameters

offsets

[array-like (N, 2) or (2,)]

`set_path_effects`(*self*, *path_effects*)
Set the path effects.

Parameters

path_effects

[*AbstractPathEffect*]

`set_paths`(*self*, *segments*)

`set_picker`(*self*, *picker*)
Define the picking behavior of the artist.

Parameters

picker

[None or bool or callable] This can be one of the following:

- *None*: Picking is disabled for this artist (default).
- A boolean: If *True* then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist.
- A function: If picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

to determine the hit test. if the mouse event is over the artist, return *hit=True* and props is a dictionary of properties you want added to the PickEvent attributes.

- *deprecated*: For *Line2D* only, *picker* can also be a float that sets the tolerance for checking whether an event occurred "on" the line; this is deprecated. Use *Line2D.set_pickradius* instead.

`set_pickradius`(*self*, *pr*)
Set the pick radius used for containment tests.

Parameters

d

[float] Pick radius, in points.

`set_rasterized`(*self*, *rasterized*)
Force rasterized (bitmap) drawing in vector backend output.

Defaults to None, which implies the backend's default behavior.

Parameters

rasterized

[bool or None]

`set_segments`(*self*, *segments*)

`set_sketch_params`(*self*, *scale=None*, *length=None*, *randomness=None*)
Sets the sketch parameters.

Parameters

scale

[float, optional] The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is `None`, or not provided, no sketch filter will be provided.

length

[float, optional] The length of the wiggle along the line, in pixels (default 128.0)

randomness

[float, optional] The scale factor by which the length is shrunken or expanded (default 16.0)

`set_snap`(*self*, *snap*)
Set the snapping behavior.

Snapping aligns positions with the pixel grid, which results in clearer images. For example, if a black line of 1px width was defined at a position in between two pixels, the resulting image would contain the interpolated

value of that line in the pixel grid, which would be a grey value on both adjacent pixel positions. In contrast, snapping will move the line to the nearest integer pixel value, so that the resulting image will really contain a 1px wide black line.

Snapping is currently only supported by the Agg and MacOSX backends.

Parameters

snap

[bool or None] Possible values:

- *True*: Snap vertices to the nearest pixel center.
- *False*: Do not modify vertex positions.
- *None*: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center.

`set_transform`(*self*, *t*)

Set the artist transform.

Parameters

t

[*Transform*]

`set_url`(*self*, *url*)

Set the url for the artist.

Parameters

url

[str]

`set_urls`(*self*, *urls*)

Parameters

urls

[list of str or None]

Notes

URLs are currently only implemented by the SVG backend. They are ignored by all other backends.

`set_verts`(*self*, *segments*)

`set_visible`(*self*, *b*)
Set the artist's visibility.

Parameters

b

[bool]

`set_zorder`(*self*, *level*)
Set the zorder for the artist. Artists with lower zorder values are drawn first.

Parameters

level

[float]

`property stale`
Whether the artist is 'stale' and needs to be re-drawn for the output to match the internal state of the artist.

`property sticky_edges`
`x` and `y` sticky edge lists for autoscaling.

When performing autoscaling, if a data limit coincides with a value in the corresponding sticky_edges list, then no margin will be added--the view limit "sticks" to the edge. A typical use case is histograms, where one usually expects no margin on the bottom edge (0) of the histogram.

This attribute cannot be assigned to; however, the `x` and `y` lists can be modified in place as needed.

Examples

```
>>> artist.sticky_edges.x[:] = (xmin, xmax)
>>> artist.sticky_edges.y[:] = (ymin, ymax)
```

`to_rgba`(*self*, *x*, *alpha=None*, *bytes=False*, *norm=True*)
Return a normalized rgba array corresponding to *x*.

In the normal case, *x* is a 1-D or 2-D sequence of scalars, and the corresponding ndarray of rgba values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already rgb or rgba, such as might have been read from an image file. If *x* is an ndarray with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an rgb or rgba array, and no mapping will be done. The array can be uint8, or it can be floating point with values in the 0-1 range; otherwise a ValueError will be raised. If it is a masked array, the mask will be ignored. If the last dimension is 3, the *alpha* kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the *alpha* kwarg is ignored; it does not replace the pre-existing alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if *bytes* is *False* (default), the rgba array will be floats in the 0-1 range; if it is *True*, the returned rgba array will be uint8 in the 0 to 255 range.

If norm is False, no normalization of the input data is performed, and it is assumed to be in the range (0-1).

`update`(*self*, *props*)
Update this artist's properties from the dict *props*.

Parameters

props

[dict]

`property update_dict`

`update_from`(*self*, *other*)
Copy properties from other to self.

`update_scalarmappable`(*self*)
Update colors from the scalar mappable array, if it is not None.

`zorder = 0`

`class matplotlib.collections.PatchCollection`(*patches*, *match_original=False*, ***kwargs*)
Bases: *matplotlib.collections.Collection*

A generic collection of patches.

This makes it easier to assign a color map to a heterogeneous collection of patches.

This also may improve plotting speed, since PatchCollection will draw faster than a large number of patches.

patches

a sequence of Patch objects. This list may include a heterogeneous assortment of different patch types.

match_original

If True, use the colors and linewidths of the original patches. If False, new colors may be assigned by providing the standard collection arguments, facecolor, edgecolor, linewidths, norm or cmap.

If any of *edgecolors*, *facecolors*, *linewidths*, *antialiaseds* are None, they default to their *rcParams* patch setting, in sequence form.

The use of *ScalarMappable* functionality is optional. If the *ScalarMappable* matrix `_A` has been set (via a call to *set_array*), at draw time a call to scalar mappable will be made to set the face colors.

`add_callback`(*self*, *func*)

Add a callback function that will be called whenever one of the *Artist*'s properties changes.

Parameters

func

[callable] The callback function. It must have the signature:

```
def func(artist: Artist) -> Any
```

where *artist* is the calling *Artist*. Return values may exist but are ignored.

Returns

int

The observer id associated with the callback. This id can be used for removing the callback with *remove_callback* later.

See also:

remove_callback

`add_checker`(*self*, *checker*)

[*Deprecated*]

Notes

Deprecated since version 3.3:

`autoscale`(*self*)

Autoscale the scalar limits on the norm instance using the current array

`autoscale_None`(*self*)

Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

property axes
The *Axes* instance the artist resides in, or *None*.

changed(*self*)
Call this whenever the mappable is changed to notify all the callbackSM listeners to the 'changed' signal.

check_update(*self*, *checker*)
[*Deprecated*]

Notes

Deprecated since version 3.3:

contains(*self*, *mouseevent*)
Test whether the mouse event occurred in the collection.

Returns `bool, dict(ind=itemlist)`, where every item in itemlist contains the event.

convert_xunits(*self*, *x*)
Convert *x* using the unit type of the xaxis.

If the artist is not in contained in an Axes or if the xaxis does not have units, *x* itself is returned.

convert_yunits(*self*, *y*)
Convert *y* using the unit type of the yaxis.

If the artist is not in contained in an Axes or if the yaxis does not have units, *y* itself is returned.

draw(*self*, *renderer*)
Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

Parameters

renderer

[*RendererBase* subclass.]

Notes

This method is overridden in the Artist subclasses.

`findobj`(*self*, *match=None*, *include_self=True*)
Find artist objects.

Recursively find all *Artist* instances contained in the artist.

Parameters

match

A filter criterion for the matches. This can be

- *None*: Return all objects contained in artist.
- A function with signature `def match(artist: Artist) -> bool`. The result will only contain artists for which the function returns *True*.
- A class instance: e.g., *Line2D*. The result will only contain artists of this class or its subclasses (`isinstance` check).

include_self

[bool] Include *self* in the list to be checked for a match.

Returns

list of *Artist*

`format_cursor_data`(*self*, *data*)
Return a string representation of *data*.

Note: This method is intended to be overridden by artist subclasses. As an end-user of Matplotlib you will most likely not call this method yourself.

The default implementation converts ints and floats and arrays of ints and floats into a comma-separated string enclosed in square brackets.

See also:

get_cursor_data

`get_agg_filter`(*self*)
Return filter function to be used for agg filter.

`get_alpha`(*self*)
Return the alpha value used for blending - not supported on all backends.

`get_animated`(*self*)
Return whether the artist is animated.

`get_array`(*self*)
Return the data array.

`get_capstyle`(*self*)

`get_children`(*self*)
Return a list of the child *Artist*s of this *Artist*.

`get_clim`(*self*)
Return the values (min, max) that are mapped to the colormap limits.

`get_clip_box`(*self*)
Return the clipbox.

`get_clip_on`(*self*)
Return whether the artist uses clipping.

`get_clip_path`(*self*)
Return the clip path.

`get_cmap`(*self*)
Return the *Colormap* instance.

`get_contains`(*self*)
[*Deprecated*] Return the custom contains function of the artist if set, or *None*.

See also:

set_contains

Notes

Deprecated since version 3.3.

`get_cursor_data`(*self*, *event*)
Return the cursor data for a given event.

Note: This method is intended to be overridden by artist subclasses. As an end-user of Matplotlib you will most likely not call this method yourself.

Cursor data can be used by Artists to provide additional context information for a given event. The default implementation just returns *None*.

Subclasses can override the method and return arbitrary data. However, when doing so, they must ensure that *format_cursor_data* can convert the data to a string representation.

The only current use case is displaying the z-value of an *AxesImage* in the status bar of a plot window, while moving the mouse.

Parameters

event

[*matplotlib.backend_bases.MouseEvent*]

See also:

format_cursor_data

`get_dashes`(*self*)
Alias for *get_linestyle*.

`get_datalim`(*self*, *transData*)

`get_ec`(*self*)
Alias for *get_edgecolor*.

`get_edgecolor`(*self*)

`get_edgecolors`(*self*)
Alias for *get_edgecolor*.

`get_facecolor`(*self*)

`get_facecolors`(*self*)
Alias for *get_facecolor*.

`get_fc`(*self*)
Alias for *get_facecolor*.

`get_figure`(*self*)
Return the *Figure* instance the artist belongs to.

`get_fill`(*self*)
Return whether fill is set.

`get_gid`(*self*)
Return the group id.

`get_hatch`(*self*)
Return the current hatching pattern.

`get_in_layout`(*self*)
Return boolean flag, `True` if artist is included in layout calculations.

E.g. *Constrained Layout Guide*, *Figure.tight_layout()*, and `fig.savefig(fname, bbox_inches='tight')`.

`get_joinstyle`(*self*)

`get_label`(*self*)
Return the label used for this artist in the legend.

get_linestyle(*self*)

get_linestyles(*self*)
Alias for *get_linestyle*.

get_linewidth(*self*)

get_linewidths(*self*)
Alias for *get_linewidth*.

get_ls(*self*)
Alias for *get_linestyle*.

get_lw(*self*)
Alias for *get_linewidth*.

get_offset_position(*self*)
[*Deprecated*] Return how offsets are applied for the collection. If *offset_position* is 'screen', the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is 'data', the offset is applied before the master transform, i.e., the offsets are in data coordinates.

Notes

Deprecated since version 3.3.

get_offset_transform(*self*)

get_offsets(*self*)
Return the offsets for the collection.

get_path_effects(*self*)

get_paths(*self*)

get_picker(*self*)
Return the picking behavior of the artist.

The possible values are described in *set_picker*.

See also:

set_picker, *pickable*, *pick*

get_pickradius(*self*)

get_rasterized(*self*)
Return whether the artist is to be rasterized.

get_sketch_params(*self*)
Return the sketch parameters for the artist.

Returns

tuple or None

A 3-tuple with the following elements:

- *scale*: The amplitude of the wiggle perpendicular to the source line.
- *length*: The length of the wiggle along the line.
- *randomness*: The scale factor by which the length is shrunken or expanded.

Returns *None* if no sketch parameters were set.

`get_snap`(*self*)
Return the snap setting.

See *set_snap* for details.

`get_tightbbox`(*self*, *renderer*)
Like *Artist.get_window_extent*, but includes any clipping.

Parameters

renderer

[*RendererBase* subclass] renderer that will be used to draw the figures (i.e. `fig.canvas.get_renderer()`)

Returns

Bbox

The enclosing bounding box (in figure pixel coordinates).

`get_transform`(*self*)
Return the *Transform* instance used by this artist.

`get_transformed_clip_path_and_affine`(*self*)
Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

`get_transforms`(*self*)

`get_url`(*self*)
Return the url.

`get_urls`(*self*)
Return a list of URLs, one for each element of the collection.

The list contains *None* for elements without a URL. See /gallery/misc/hyperlinks_sgskip for an example.

`get_visible`(*self*)
Return the visibility.

`get_window_extent`(*self*, *renderer*)
Get the axes bounding box in display space.

The bounding box' width and height are nonnegative.

Subclasses should override for inclusion in the bounding box "tight" calculation. Default is to return an empty bounding box at 0, 0.

Be careful when using this function, the results will not update if the artist window extent of the artist changes. The extent can change due to any changes in the transform stack, such as changing the axes limits, the figure size, or the canvas used (as is done when saving a figure). This can lead to unexpected behavior where interactive figures will look fine on the screen, but will save incorrectly.

`get_zorder`(*self*)
Return the artist's zorder.

`have_units`(*self*)
Return *True* if units are set on any axis.

`is_transform_set`(*self*)
Return whether the Artist has an explicitly set transform.

This is *True* after *set_transform* has been called.

`property mouseover`
If this property is set to *True*, the artist will be queried for custom context information when the mouse cursor moves over it.

See also *get_cursor_data()*, *ToolCursorPosition* and *NavigationToolbar2*.

`pchanged`(*self*)
Call all of the registered callbacks.

This function is triggered internally when a property is changed.

See also:

add_callback

remove_callback

`pick`(*self*, *mouseevent*)
Process a pick event.

Each child artist will fire a pick event if *mouseevent* is over the artist and the artist has picker set.

See also:

set_picker, *get_picker*, *pickable*

`pickable`(*self*)
Return whether the artist is pickable.

See also:

set_picker, *get_picker*, *pick*

`properties`(*self*)
Return a dictionary of all the properties of the artist.

`remove`(*self*)
Remove the artist from the figure if possible.

The effect will not be visible until the figure is redrawn, e.g., with *FigureCanvasBase.draw_idle*. Call *relim* to update the axes limits if desired.

Note: *relim* will not see collections even if the collection was added to the axes with *autolim* = True.

Note: there is no support for removing the artist's legend entry.

`remove_callback`(*self*, *oid*)
Remove a callback based on its observer id.

See also:

add_callback

`set`(*self*, ***kwargs*)
A property batch setter. Pass *kwargs* to set properties.

`set_aa`(*self*, *aa*)
Alias for *set_antialiased*.

`set_agg_filter`(*self*, *filter_func*)
Set the agg filter.

Parameters

filter_func

[callable] A filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array.

`set_alpha`(*self*, *alpha*)
Set the alpha value used for blending - not supported on all backends.

Parameters

alpha

[float or None]

`set_animated`(*self*, *b*)
Set the artist's animation state.

Parameters

b

[bool]

`set_antialiased`(*self*, *aa*)
Set the antialiasing state for rendering.

Parameters

aa

[bool or list of bools]

`set_antialiaseds`(*self*, *aa*)
Alias for *set_antialiased*.

`set_array`(*self*, *A*)
Set the image array from numpy array *A*.

Parameters

A

[ndarray]

`set_capstyle`(*self*, *cs*)
Set the capstyle for the collection (for all its elements).

Parameters

cs

[{'butt', 'round', 'projecting'}] The capstyle.

`set_clim`(*self*, *vmin=None*, *vmax=None*)
Set the norm limits for image scaling.

Parameters

vmin, vmax

[float] The limits.

The limits may also be passed as a tuple (*vmin*, *vmax*) as a single positional argument.

`set_clip_box`(*self*, *clipbox*)
Set the artist's clip *Bbox*.

Parameters

clipbox

[*Bbox*]

set_clip_on(*self*, *b*)
Set whether the artist uses clipping.

When False artists will be visible outside of the axes which can lead to unexpected results.

Parameters

b

[bool]

set_clip_path(*self*, *path*, *transform=None*)
Set the artist's clip path.

Parameters

path

[*Patch* or *Path* or *TransformedPath* or None] The clip path. If given a *Path*, *transform* must be provided as well. If *None*, a previously set clip path is removed.

transform

[*Transform*, optional] Only used if *path* is a *Path*, in which case the given *Path* is converted to a *TransformedPath* using *transform*.

Notes

For efficiency, if *path* is a *Rectangle* this method will set the clipping box to the corresponding rectangle and set the clipping path to `None`.

For technical reasons (support of *set*), a tuple (*path*, *transform*) is also accepted as a single positional parameter.

set_cmap(*self*, *cmap*)
Set the colormap for luminance data.

Parameters

cmap

[*Colormap* or str or None]

set_color(*self*, *c*)
Set both the edgecolor and the facecolor.

Parameters

c

[color or list of rgba tuples]

See also:

Collection.set_facecolor, *Collection.set_edgecolor*

For setting the edge or face color individually.

`set_contains`(*self*, *picker*)
[*Deprecated*] Define a custom contains test for the artist.

The provided callable replaces the default *contains* method of the artist.

Parameters

picker

[callable] A custom picker function to evaluate if an event is within the artist. The function must have the signature:

```
def contains(artist: Artist, event: MouseEvent) -> bool, dict
```

that returns:

- a bool indicating if the event is within the artist
- a dict of additional information. The dict should at least return the same information as the default `contains()` implementation of the respective artist, but may provide additional information.

Notes

Deprecated since version 3.3.

`set_dashes`(*self*, *ls*)
Alias for *set_linestyle*.

`set_ec`(*self*, *c*)
Alias for *set_edgecolor*.

`set_edgecolor`(*self*, *c*)
Set the edgecolor(s) of the collection.

Parameters

c

[color or list of colors or 'face'] The collection edgecolor(s). If a sequence, the patches cycle through it. If 'face', match the facecolor.

set_edgecolors(*self*, *c*)
Alias for *set_edgecolor*.

set_facecolor(*self*, *c*)
Set the facecolor(s) of the collection. *c* can be a color (all patches have same color), or a sequence of colors; if it is a sequence the patches will cycle through the sequence.

If *c* is 'none', the patch will not be filled.

Parameters

c

[color or list of colors]

set_facecolors(*self*, *c*)
Alias for *set_facecolor*.

set_fc(*self*, *c*)
Alias for *set_facecolor*.

set_figure(*self*, *fig*)
Set the *Figure* instance the artist belongs to.

Parameters

fig

[*Figure*]

set_gid(*self*, *gid*)
Set the (group) id for the artist.

Parameters

gid

[str]

set_hatch(*self*, *hatch*)
Set the hatching pattern

hatch can be one of:

```
/   - diagonal hatching
\   - back diagonal
|   - vertical
-   - horizontal
+   - crossed
x   - crossed diagonal
o   - small circle
O   - large circle
```

(continues on next page)

(continued from previous page)

```
.   - dots
*   - stars
```

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Unlike other properties such as linewidth and colors, hatching can only be specified for the collection as a whole, not separately for each member.

Parameters

hatch

[{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}]

`set_in_layout`(*self*, *in_layout*)

Set if artist is to be included in layout calculations, E.g. *Constrained Layout Guide*, *`Figure.tight_layout()`*, and `fig.savefig(fname, bbox_inches='tight')`.

Parameters

in_layout

[bool]

`set_joinstyle`(*self*, *js*)

Set the joinstyle for the collection (for all its elements).

Parameters

js

[{'miter', 'round', 'bevel'}] The joinstyle.

`set_label`(*self*, *s*)

Set a label that will be displayed in the legend.

Parameters

s

[object] *s* will be converted to a string by calling `str`.

`set_linestyle`(*self*, *ls*)

Set the linestyle(s) for the collection.

linestyle	description
`'-'` or `'solid'`	solid line
`'--'` or `'dashed'`	dashed line
`'-.'` or `'dashdot'`	dash-dotted line
`':'` or `'dotted'`	dotted line

Alternatively a dash tuple of the following form can be provided:

```
(offset, onoffseq),
```

where `onoffseq` is an even length tuple of on and off ink in points.

Parameters

ls

[str or tuple or list thereof] Valid values for individual linestyles include {'-', '--', '-.', ':', '', (offset, on-off-seq)}. See *Line2D.set_linestyle* for a complete description.

`set_linestyles`(*self*, *ls*)
Alias for *set_linestyle*.

`set_linewidth`(*self*, *lw*)
Set the linewidth(s) for the collection. *lw* can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence

Parameters

lw

[float or list of floats]

`set_linewidths`(*self*, *lw*)
Alias for *set_linewidth*.

`set_ls`(*self*, *ls*)
Alias for *set_linestyle*.

`set_lw`(*self*, *lw*)
Alias for *set_linewidth*.

`set_norm`(*self*, *norm*)
Set the normalization instance.

Parameters

norm

[*Normalize* or None]

Notes

If there are any colorbars using the mappable for this norm, setting the norm of the mappable will reset the norm, locator, and formatters on the colorbar to default.

`set_offset_position`(*self*, *offset_position*)
[*Deprecated*] Set how offsets are applied. If *offset_position* is 'screen' (default) the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is 'data', the offset is applied before the master transform, i.e., the offsets are in data coordinates.

Parameters

offset_position

[{'screen', 'data'}]

Notes

Deprecated since version 3.3.

`set_offsets`(*self*, *offsets*)
Set the offsets for the collection.

Parameters

offsets

[array-like (N, 2) or (2,)]

`set_path_effects`(*self*, *path_effects*)
Set the path effects.

Parameters

path_effects

[*AbstractPathEffect*]

`set_paths`(*self*, *patches*)

`set_picker`(*self*, *picker*)
Define the picking behavior of the artist.

Parameters

picker

[None or bool or callable] This can be one of the following:

- *None*: Picking is disabled for this artist (default).
- A boolean: If *True* then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist.
- A function: If picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

to determine the hit test. if the mouse event is over the artist, return *hit=True* and props is a dictionary of properties you want added to the PickEvent attributes.

- *deprecated*: For *Line2D* only, *picker* can also be a float that sets the tolerance for checking whether an event occurred "on" the line; this is deprecated. Use *Line2D.set_pickradius* instead.

`set_pickradius`(*self*, *pr*)

Set the pick radius used for containment tests.

Parameters

d

[float] Pick radius, in points.

`set_rasterized`(*self*, *rasterized*)

Force rasterized (bitmap) drawing in vector backend output.

Defaults to None, which implies the backend's default behavior.

Parameters

rasterized

[bool or None]

`set_sketch_params`(*self*, *scale=None*, *length=None*, *randomness=None*)

Sets the sketch parameters.

Parameters

scale

[float, optional] The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is `None`, or not provided, no sketch filter will be provided.

length

[float, optional] The length of the wiggle along the line, in pixels (default 128.0)

randomness

[float, optional] The scale factor by which the length is shrunken or expanded (default 16.0)

`set_snap`(*self*, *snap*)

Set the snapping behavior.

Snapping aligns positions with the pixel grid, which results in clearer images. For example, if a black line of 1px width was defined at a position in between two pixels, the resulting image would contain the interpolated value of that line in the pixel grid, which would be a grey value on both adjacent pixel positions. In contrast, snapping will move the line to the nearest integer pixel value, so that the resulting image will really contain a 1px wide black line.

Snapping is currently only supported by the Agg and MacOSX backends.

Parameters

snap

[bool or None] Possible values:

- *True*: Snap vertices to the nearest pixel center.
- *False*: Do not modify vertex positions.
- *None*: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center.

`set_transform`(*self*, *t*)

Set the artist transform.

Parameters

t

[*Transform*]

`set_url`(*self*, *url*)

Set the url for the artist.

Parameters

url

[str]

`set_urls`(*self*, *urls*)

Parameters

urls

[list of str or None]

Notes

URLs are currently only implemented by the SVG backend. They are ignored by all other backends.

`set_visible`(*self*, *b*)
Set the artist's visibility.

Parameters

b

[bool]

`set_zorder`(*self*, *level*)
Set the zorder for the artist. Artists with lower zorder values are drawn first.

Parameters

level

[float]

`property stale`
Whether the artist is 'stale' and needs to be re-drawn for the output to match the internal state of the artist.

`property sticky_edges`
`x` and `y` sticky edge lists for autoscaling.

When performing autoscaling, if a data limit coincides with a value in the corresponding sticky_edges list, then no margin will be added--the view limit "sticks" to the edge. A typical use case is histograms, where one usually expects no margin on the bottom edge (0) of the histogram.

This attribute cannot be assigned to; however, the `x` and `y` lists can be modified in place as needed.

Examples

```
>>> artist.sticky_edges.x[:] = (xmin, xmax)
>>> artist.sticky_edges.y[:] = (ymin, ymax)
```

`to_rgba`(*self*, *x*, *alpha=None*, *bytes=False*, *norm=True*)
Return a normalized rgba array corresponding to *x*.

In the normal case, *x* is a 1-D or 2-D sequence of scalars, and the corresponding ndarray of rgba values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already rgb or rgba, such as might have been read from an image file. If *x* is an ndarray with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an rgb or rgba array, and no mapping will be done. The array can be uint8, or it can be floating point with values in the 0-1 range; otherwise a ValueError will be raised. If it is a masked array, the mask will be ignored. If the last dimension is 3, the *alpha* kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the *alpha* kwarg is ignored; it does not replace the pre-existing alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if *bytes* is *False* (default), the rgba array will be floats in the 0-1 range; if it is *True*, the returned rgba array will be uint8 in the 0 to 255 range.

If norm is False, no normalization of the input data is performed, and it is assumed to be in the range (0-1).

`update`(*self*, *props*)
Update this artist's properties from the dict *props*.

Parameters

props

[dict]

`property update_dict`

`update_from`(*self*, *other*)
Copy properties from other to self.

`update_scalarmappable`(*self*)
Update colors from the scalar mappable array, if it is not None.

`zorder = 0`

`class matplotlib.collections.PathCollection`(*paths*, *sizes=None*, ***kwargs*)
Bases: `matplotlib.collections._CollectionWithSizes`

A collection of *Path*s, as created by e.g. *scatter*.

Parameters

paths

[list of *path.Path*] The paths that will make up the *Collection*.

sizes

[array-like] The factor by which to scale each drawn *Path*. One unit squared in the Path's data space is scaled to be `sizes**2` points when rendered.

****kwargs**

Forwarded to *Collection*.

`add_callback`(*self*, *func*)

Add a callback function that will be called whenever one of the *Artist*'s properties changes.

Parameters

func

[callable] The callback function. It must have the signature:

```
def func(artist: Artist) -> Any
```

where *artist* is the calling *Artist*. Return values may exist but are ignored.

Returns

int

The observer id associated with the callback. This id can be used for removing the callback with *remove_callback* later.

See also:

remove_callback

`add_checker`(*self*, *checker*)

[*Deprecated*]

Notes

Deprecated since version 3.3:

`autoscale`(*self*)
Autoscale the scalar limits on the norm instance using the current array

`autoscale_None`(*self*)
Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

`property axes`
The *Axes* instance the artist resides in, or *None*.

`changed`(*self*)
Call this whenever the mappable is changed to notify all the callbackSM listeners to the 'changed' signal.

`check_update`(*self*, *checker*)
[*Deprecated*]

Notes

Deprecated since version 3.3:

`contains`(*self*, *mouseevent*)
Test whether the mouse event occurred in the collection.

Returns `bool, dict(ind=itemlist)`, where every item in itemlist contains the event.

`convert_xunits`(*self*, *x*)
Convert *x* using the unit type of the xaxis.

If the artist is not in contained in an Axes or if the xaxis does not have units, *x* itself is returned.

`convert_yunits`(*self*, *y*)
Convert *y* using the unit type of the yaxis.

If the artist is not in contained in an Axes or if the yaxis does not have units, *y* itself is returned.

`draw`(*self*, *renderer*)
Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

Parameters

renderer

[*RendererBase* subclass.]

Notes

This method is overridden in the Artist subclasses.

`findobj`(*self*, *match=None*, *include_self=True*)
Find artist objects.

Recursively find all *Artist* instances contained in the artist.

Parameters

match

A filter criterion for the matches. This can be

- *None*: Return all objects contained in artist.
- A function with signature `def match(artist: Artist) -> bool`. The result will only contain artists for which the function returns *True*.
- A class instance: e.g., *Line2D*. The result will only contain artists of this class or its subclasses (`isinstance` check).

include_self

[bool] Include *self* in the list to be checked for a match.

Returns

list of *Artist*

`format_cursor_data`(*self*, *data*)
Return a string representation of *data*.

Note: This method is intended to be overridden by artist subclasses. As an end-user of Matplotlib you will most likely not call this method yourself.

The default implementation converts ints and floats and arrays of ints and floats into a comma-separated string enclosed in square brackets.

See also:

get_cursor_data

`get_agg_filter`(*self*)
Return filter function to be used for agg filter.

`get_alpha`(*self*)
Return the alpha value used for blending - not supported on all backends.

`get_animated`(*self*)
Return whether the artist is animated.

`get_array`(*self*)
Return the data array.

`get_capstyle`(*self*)

`get_children`(*self*)
Return a list of the child *Artist*s of this *Artist*.

`get_clim`(*self*)
Return the values (min, max) that are mapped to the colormap limits.

`get_clip_box`(*self*)
Return the clipbox.

`get_clip_on`(*self*)
Return whether the artist uses clipping.

`get_clip_path`(*self*)
Return the clip path.

`get_cmap`(*self*)
Return the *Colormap* instance.

`get_contains`(*self*)
[*Deprecated*] Return the custom contains function of the artist if set, or *None*.

See also:

set_contains

Notes

Deprecated since version 3.3.

`get_cursor_data`(*self*, *event*)
Return the cursor data for a given event.

Note: This method is intended to be overridden by artist subclasses. As an end-user of Matplotlib you will most likely not call this method yourself.

Cursor data can be used by Artists to provide additional context information for a given event. The default implementation just returns *None*.

Subclasses can override the method and return arbitrary data. However, when doing so, they must ensure that *format_cursor_data* can convert the data to a string representation.

The only current use case is displaying the z-value of an *AxesImage* in the status bar of a plot window, while moving the mouse.

Parameters

event

[*matplotlib.backend_bases.MouseEvent*]

See also:

format_cursor_data

`get_dashes`(*self*)
Alias for *get_linestyle*.

`get_datalim`(*self*, *transData*)

`get_ec`(*self*)
Alias for *get_edgecolor*.

`get_edgecolor`(*self*)

`get_edgecolors`(*self*)
Alias for *get_edgecolor*.

`get_facecolor`(*self*)

`get_facecolors`(*self*)
Alias for *get_facecolor*.

`get_fc`(*self*)
Alias for *get_facecolor*.

`get_figure`(*self*)
Return the *Figure* instance the artist belongs to.

`get_fill`(*self*)
Return whether fill is set.

`get_gid`(*self*)
Return the group id.

`get_hatch`(*self*)
Return the current hatching pattern.

`get_in_layout`(*self*)
Return boolean flag, `True` if artist is included in layout calculations.

E.g. *Constrained Layout Guide*, *Figure.tight_layout()*, and `fig.savefig(fname, bbox_inches='tight')`.

`get_joinstyle`(*self*)

`get_label`(*self*)
Return the label used for this artist in the legend.

`get_linestyle`(*self*)

`get_linestyles`(*self*)
Alias for *get_linestyle*.

`get_linewidth`(*self*)

`get_linewidths`(*self*)
Alias for *get_linewidth*.

`get_ls`(*self*)
Alias for *get_linestyle*.

`get_lw`(*self*)
Alias for *get_linewidth*.

`get_offset_position`(*self*)
[*Deprecated*] Return how offsets are applied for the collection. If *offset_position* is 'screen', the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is 'data', the offset is applied before the master transform, i.e., the offsets are in data coordinates.

Notes

Deprecated since version 3.3.

`get_offset_transform`(*self*)

`get_offsets`(*self*)
Return the offsets for the collection.

`get_path_effects`(*self*)

`get_paths`(*self*)

`get_picker`(*self*)
Return the picking behavior of the artist.

The possible values are described in *set_picker*.

See also:

set_picker, *pickable*, *pick*

`get_pickradius`(*self*)

`get_rasterized`(*self*)
Return whether the artist is to be rasterized.

`get_sizes`(*self*)
Return the sizes ('areas') of the elements in the collection.

Returns

array

The 'area' of each element.

`get_sketch_params`(*self*)
Return the sketch parameters for the artist.

Returns

tuple or None

A 3-tuple with the following elements:

- *scale*: The amplitude of the wiggle perpendicular to the source line.
- *length*: The length of the wiggle along the line.
- *randomness*: The scale factor by which the length is shrunken or expanded.

Returns *None* if no sketch parameters were set.

`get_snap`(*self*)
Return the snap setting.

See *set_snap* for details.

`get_tightbbox`(*self*, *renderer*)
Like *Artist.get_window_extent*, but includes any clipping.

Parameters

renderer

[*RendererBase* subclass] renderer that will be used to draw the figures (i.e. `fig.canvas.get_renderer()`)

Returns

Bbox

The enclosing bounding box (in figure pixel coordinates).

`get_transform`(*self*)
Return the *Transform* instance used by this artist.

`get_transformed_clip_path_and_affine`(*self*)
Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

`get_transforms`(*self*)

`get_url`(*self*)
Return the url.

`get_urls`(*self*)
Return a list of URLs, one for each element of the collection.

The list contains *None* for elements without a URL. See /gallery/misc/hyperlinks_sgskip for an example.

`get_visible`(*self*)
Return the visibility.

`get_window_extent`(*self*, *renderer*)
Get the axes bounding box in display space.

The bounding box' width and height are nonnegative.

Subclasses should override for inclusion in the bounding box "tight" calculation. Default is to return an empty bounding box at 0, 0.

Be careful when using this function, the results will not update if the artist window extent of the artist changes. The extent can change due to any changes in the transform stack, such as changing the axes limits, the figure size, or the canvas used (as is done when saving a figure). This can lead to unexpected behavior where interactive figures will look fine on the screen, but will save incorrectly.

`get_zorder`(*self*)
Return the artist's zorder.

`have_units`(*self*)
Return *True* if units are set on any axis.

`is_transform_set`(*self*)
Return whether the Artist has an explicitly set transform.

This is *True* after *set_transform* has been called.

`legend_elements`(*self*, *prop='colors'*, *num='auto'*, *fmt=None*, *func=<function PathCollection.<lambda> at 0x7f5434b37af0>*, ***kwargs*)
Create legend handles and labels for a PathCollection.

Each legend handle is a *Line2D* representing the Path that was drawn, and each label is a string what each Path represents.

This is useful for obtaining a legend for a *scatter* plot; e.g.:

```
scatter = plt.scatter([1, 2, 3],  [4, 5, 6],  c=[7, 2, 3])
plt.legend(*scatter.legend_elements())
```

creates three legend elements, one for each color with the numerical values passed to *c* as the labels.

Also see the automatedlegendcreation example.

Parameters

prop

[{"colors", "sizes"}, default: "colors"] If "colors", the legend handles will show the different colors of the collection. If "sizes", the legend will show the different sizes. To set both, use *kwargs* to directly edit the *Line2D* properties.

num

[int, None, "auto" (default), array-like, or *Locator*,] Target number of elements to create. If None, use all unique elements of the mappable array. If an integer, target to use *num* elements in the normed range. If "*auto*", try to determine which option better suits the nature of the data. The number of created elements may slightly deviate from *num* due to a *Locator* being used to find useful locations. If a list or array, use exactly those elements for the legend. Finally, a *Locator* can be provided.

fmt

[str, *Formatter*, or None (default)] The format or formatter to use for the labels. If a string must be a valid input for a *StrMethodFormatter*. If None (the default), use a *ScalarFormatter*.

func

[function, default *lambda x: x*] Function to calculate the labels. Often the size (or color) argument to *scatter* will have been pre-processed by the user using a function `s = f(x)` to make the markers visible; e.g. `size = np.log10(x)`. Providing the inverse of this function here allows that pre-processing to be inverted, so that the legend labels have the correct values; e.g. `func = lambda x: 10**x`.

****kwargs**

Allowed keyword arguments are *color* and *size*. E.g. it may be useful to set the color of the markers if *prop*="*sizes*" is used; similarly to set the size of the markers if *prop*="*colors*" is used. Any further parameters are passed onto the *Line2D* instance. This may be useful to e.g. specify a different *markeredgecolor* or *alpha* for the legend handles.

Returns

handles

[list of *Line2D*] Visual representation of each element of the legend.

labels

[list of str] The string labels for elements of the legend.

`property mouseover`
: If this property is set to *True*, the artist will be queried for custom context information when the mouse cursor moves over it.

  See also *get_cursor_data()*, *ToolCursorPosition* and *NavigationToolbar2*.

`pchanged`(*self*)
: Call all of the registered callbacks.

  This function is triggered internally when a property is changed.

  See also:

  add_callback

  remove_callback

`pick`(*self*, *mouseevent*)
: Process a pick event.

  Each child artist will fire a pick event if *mouseevent* is over the artist and the artist has picker set.

  See also:

  set_picker, *get_picker*, *pickable*

`pickable`(*self*)
: Return whether the artist is pickable.

  See also:

  set_picker, *get_picker*, *pick*

`properties`(*self*)
: Return a dictionary of all the properties of the artist.

`remove`(*self*)
: Remove the artist from the figure if possible.

  The effect will not be visible until the figure is redrawn, e.g., with *FigureCanvasBase.draw_idle*. Call *relim* to update the axes limits if desired.

  Note: *relim* will not see collections even if the collection was added to the axes with *autolim* = True.

  Note: there is no support for removing the artist's legend entry.

`remove_callback`(*self*, *oid*)
: Remove a callback based on its observer id.

  See also:

  add_callback

`set`(*self*, ***kwargs*)
A property batch setter. Pass *kwargs* to set properties.

`set_aa`(*self*, *aa*)
Alias for *set_antialiased*.

`set_agg_filter`(*self*, *filter_func*)
Set the agg filter.

Parameters

filter_func

[callable] A filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array.

`set_alpha`(*self*, *alpha*)
Set the alpha value used for blending - not supported on all backends.

Parameters

alpha

[float or None]

`set_animated`(*self*, *b*)
Set the artist's animation state.

Parameters

b

[bool]

`set_antialiased`(*self*, *aa*)
Set the antialiasing state for rendering.

Parameters

aa

[bool or list of bools]

`set_antialiaseds`(*self*, *aa*)
Alias for *set_antialiased*.

`set_array`(*self*, *A*)
Set the image array from numpy array *A*.

Parameters

A

[ndarray]

`set_capstyle`(*self*, *cs*)
Set the capstyle for the collection (for all its elements).

Parameters

cs

[{'butt', 'round', 'projecting'}] The capstyle.

`set_clim`(*self*, *vmin=None*, *vmax=None*)
Set the norm limits for image scaling.

Parameters

vmin, vmax

[float] The limits.

The limits may also be passed as a tuple (*vmin*, *vmax*) as a single positional argument.

`set_clip_box`(*self*, *clipbox*)
Set the artist's clip *Bbox*.

Parameters

clipbox

[*Bbox*]

`set_clip_on`(*self*, *b*)
Set whether the artist uses clipping.

When False artists will be visible outside of the axes which can lead to unexpected results.

Parameters

b

[bool]

`set_clip_path`(*self*, *path*, *transform=None*)
Set the artist's clip path.

Parameters

path

[*Patch* or *Path* or *TransformedPath* or None] The clip path. If given a *Path*, *transform* must be provided as well. If *None*, a previously set clip path is removed.

transform

[*Transform*, optional] Only used if *path* is a *Path*, in which case the given *Path* is converted to a *TransformedPath* using *transform*.

Notes

For efficiency, if *path* is a *Rectangle* this method will set the clipping box to the corresponding rectangle and set the clipping path to `None`.

For technical reasons (support of *set*), a tuple (*path*, *transform*) is also accepted as a single positional parameter.

`set_cmap`(*self*, *cmap*)
Set the colormap for luminance data.

Parameters

cmap

[*Colormap* or str or None]

`set_color`(*self*, *c*)
Set both the edgecolor and the facecolor.

Parameters

c

[color or list of rgba tuples]

See also:

Collection.set_facecolor, *Collection.set_edgecolor*

For setting the edge or face color individually.

`set_contains`(*self*, *picker*)
[*Deprecated*] Define a custom contains test for the artist.

The provided callable replaces the default *contains* method of the artist.

Parameters

picker

[callable] A custom picker function to evaluate if an event is within the artist. The function must have the signature:

```
def contains(artist: Artist, event: MouseEvent) -> bool, dict
```

that returns:

- a bool indicating if the event is within the artist
- a dict of additional information. The dict should at least return the same information as the default `contains()` implementation of the respective artist, but may provide additional information.

Notes

Deprecated since version 3.3.

`set_dashes`(*self*, *ls*)
Alias for *set_linestyle*.

`set_ec`(*self*, *c*)
Alias for *set_edgecolor*.

`set_edgecolor`(*self*, *c*)
Set the edgecolor(s) of the collection.

Parameters

c

[color or list of colors or 'face'] The collection edgecolor(s). If a sequence, the patches cycle through it. If 'face', match the facecolor.

`set_edgecolors`(*self*, *c*)
Alias for *set_edgecolor*.

`set_facecolor`(*self*, *c*)
Set the facecolor(s) of the collection. *c* can be a color (all patches have same color), or a sequence of colors; if it is a sequence the patches will cycle through the sequence.

If *c* is 'none', the patch will not be filled.

Parameters

c

[color or list of colors]

`set_facecolors`(*self*, *c*)
Alias for *set_facecolor*.

`set_fc`(*self*, *c*)
Alias for *set_facecolor*.

`set_figure`(*self*, *fig*)
Set the *Figure* instance the artist belongs to.

Parameters

fig

[*Figure*]

`set_gid`(*self*, *gid*)
Set the (group) id for the artist.

Parameters

gid

[str]

`set_hatch`(*self*, *hatch*)
Set the hatching pattern

hatch can be one of:

```
/   - diagonal hatching
\   - back diagonal
|   - vertical
-   - horizontal
+   - crossed
x   - crossed diagonal
o   - small circle
O   - large circle
.   - dots
*   - stars
```

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Unlike other properties such as linewidth and colors, hatching can only be specified for the collection as a whole, not separately for each member.

Parameters

hatch

[{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}]

`set_in_layout`(*self*, *in_layout*)
Set if artist is to be included in layout calculations, E.g. *Constrained Layout Guide*, *Figure.tight_layout()*, and `fig.savefig(fname, bbox_inches='tight')`.

Parameters

in_layout

[bool]

set_joinstyle(*self*, *js*)
Set the joinstyle for the collection (for all its elements).

Parameters

js

[{'miter', 'round', 'bevel'}] The joinstyle.

set_label(*self*, *s*)
Set a label that will be displayed in the legend.

Parameters

s

[object] *s* will be converted to a string by calling `str`.

set_linestyle(*self*, *ls*)
Set the linestyle(s) for the collection.

linestyle	description
`'-'` or `'solid'`	solid line
`'--'` or `'dashed'`	dashed line
`'-.'` or `'dashdot'`	dash-dotted line
`':'` or `'dotted'`	dotted line

Alternatively a dash tuple of the following form can be provided:

```
(offset, onoffseq),
```

where `onoffseq` is an even length tuple of on and off ink in points.

Parameters

ls

[str or tuple or list thereof] Valid values for individual linestyles include {'-', '--', '-.', ':', '', (offset, on-off-seq)}. See *Line2D.set_linestyle* for a complete description.

set_linestyles(*self*, *ls*)
Alias for *set_linestyle*.

set_linewidth(*self*, *lw*)
Set the linewidth(s) for the collection. *lw* can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence

Parameters

lw

[float or list of floats]

`set_linewidths`(*self*, *lw*)
Alias for *set_linewidth*.

`set_ls`(*self*, *ls*)
Alias for *set_linestyle*.

`set_lw`(*self*, *lw*)
Alias for *set_linewidth*.

`set_norm`(*self*, *norm*)
Set the normalization instance.

Parameters

norm

[*Normalize* or None]

Notes

If there are any colorbars using the mappable for this norm, setting the norm of the mappable will reset the norm, locator, and formatters on the colorbar to default.

`set_offset_position`(*self*, *offset_position*)
[*Deprecated*] Set how offsets are applied. If *offset_position* is 'screen' (default) the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is 'data', the offset is applied before the master transform, i.e., the offsets are in data coordinates.

Parameters

offset_position

[{'screen', 'data'}]

Notes

Deprecated since version 3.3.

`set_offsets`(*self*, *offsets*)
Set the offsets for the collection.

Parameters

offsets

[array-like (N, 2) or (2,)]

`set_path_effects`(*self*, *path_effects*)
Set the path effects.

Parameters

path_effects

[*AbstractPathEffect*]

`set_paths`(*self*, *paths*)

`set_picker`(*self*, *picker*)
Define the picking behavior of the artist.

Parameters

picker

[None or bool or callable] This can be one of the following:

- *None*: Picking is disabled for this artist (default).
- A boolean: If *True* then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist.
- A function: If picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

to determine the hit test. if the mouse event is over the artist, return *hit=True* and props is a dictionary of properties you want added to the PickEvent attributes.

- *deprecated*: For *Line2D* only, *picker* can also be a float that sets the tolerance for checking whether an event occurred "on" the line; this is deprecated. Use *Line2D.set_pickradius* instead.

`set_pickradius`(*self*, *pr*)
Set the pick radius used for containment tests.

Parameters

d

[float] Pick radius, in points.

`set_rasterized`(*self*, *rasterized*)
Force rasterized (bitmap) drawing in vector backend output.

Defaults to None, which implies the backend's default behavior.

Parameters

rasterized

[bool or None]

`set_sizes`(*self*, *sizes*, *dpi=72.0*)
Set the sizes of each member of the collection.

Parameters

sizes

[ndarray or None] The size to set for each element of the collection. The value is the 'area' of the element.

dpi

[float, default: 72] The dpi of the canvas.

`set_sketch_params`(*self*, *scale=None*, *length=None*, *randomness=None*)
Sets the sketch parameters.

Parameters

scale

[float, optional] The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is `None`, or not provided, no sketch filter will be provided.

length

[float, optional] The length of the wiggle along the line, in pixels (default 128.0)

randomness

[float, optional] The scale factor by which the length is shrunken or expanded (default 16.0)

`set_snap`(*self*, *snap*)
Set the snapping behavior.

Snapping aligns positions with the pixel grid, which results in clearer images. For example, if a black line of 1px width was defined at a position in between two pixels, the resulting image would contain the interpolated value of that line in the pixel grid, which would be a grey value on both adjacent pixel positions. In contrast, snapping will move the line to the nearest integer pixel value, so that the resulting image will really contain a 1px wide black line.

Snapping is currently only supported by the Agg and MacOSX backends.

Parameters

snap

[bool or None] Possible values:

- *True*: Snap vertices to the nearest pixel center.
- *False*: Do not modify vertex positions.
- *None*: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center.

`set_transform`(*self*, *t*)
Set the artist transform.

Parameters

t

[*Transform*]

`set_url`(*self*, *url*)
Set the url for the artist.

Parameters

url

[str]

`set_urls`(*self*, *urls*)

Parameters

urls

[list of str or None]

Notes

URLs are currently only implemented by the SVG backend. They are ignored by all other backends.

`set_visible`(*self*, *b*)
Set the artist's visibility.

Parameters

b

[bool]

`set_zorder`(*self*, *level*)
Set the zorder for the artist. Artists with lower zorder values are drawn first.

Parameters

level

[float]

`property stale`
Whether the artist is 'stale' and needs to be re-drawn for the output to match the internal state of the artist.

`property sticky_edges`
`x` and `y` sticky edge lists for autoscaling.

When performing autoscaling, if a data limit coincides with a value in the corresponding sticky_edges list, then no margin will be added--the view limit "sticks" to the edge. A typical use case is histograms, where one usually expects no margin on the bottom edge (0) of the histogram.

This attribute cannot be assigned to; however, the `x` and `y` lists can be modified in place as needed.

Examples

```
>>> artist.sticky_edges.x[:] = (xmin, xmax)
>>> artist.sticky_edges.y[:] = (ymin, ymax)
```

`to_rgba`(*self*, *x*, *alpha=None*, *bytes=False*, *norm=True*)
Return a normalized rgba array corresponding to *x*.

In the normal case, *x* is a 1-D or 2-D sequence of scalars, and the corresponding ndarray of rgba values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already rgb or rgba, such as might have been read from an image file. If *x* is an ndarray with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an rgb or rgba array, and no mapping will be done. The array can be uint8, or it can be floating point with values in the 0-1 range; otherwise a ValueError will be raised. If it is a masked array, the mask will be ignored. If the last dimension is 3, the *alpha* kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the *alpha* kwarg is ignored; it does not replace the pre-existing alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if *bytes* is *False* (default), the rgba array will be floats in the 0-1 range; if it is *True*, the returned rgba array will be uint8 in the 0 to 255 range.

If norm is False, no normalization of the input data is performed, and it is assumed to be in the range (0-1).

`update`(*self*, *props*)
Update this artist's properties from the dict *props*.

Parameters

props

[dict]

`property update_dict`

`update_from`(*self*, *other*)
Copy properties from other to self.

`update_scalarmappable`(*self*)
Update colors from the scalar mappable array, if it is not None.

`zorder = 0`

`class matplotlib.collections.PolyCollection`(*verts*, *sizes=None*, *closed=True*, ***kwargs*)
Bases: `matplotlib.collections._CollectionWithSizes`

Parameters

verts

[list of array-like] The sequence of polygons [*verts0*, *verts1*, ...] where each element *verts_i* defines the vertices of polygon *i* as a 2D array-like of shape (M, 2).

sizes

[array-like, default: None] Squared scaling factors for the polygons. The coordinates of each polygon *verts_i* are multiplied by the square-root of the corresponding entry in *sizes* (i.e., *sizes*

specify the scaling of areas). The scaling is applied before the Artist master transform.

closed

[bool, default: True] Whether the polygon should be closed by adding a CLOSEPOLY connection at the end.

****kwargs**

Forwarded to *Collection*.

`add_callback`(*self*, *func*)

Add a callback function that will be called whenever one of the *Artist*'s properties changes.

Parameters

func

[callable] The callback function. It must have the signature:

```
def func(artist: Artist) -> Any
```

where *artist* is the calling *Artist*. Return values may exist but are ignored.

Returns

int

The observer id associated with the callback. This id can be used for removing the callback with *remove_callback* later.

See also:

remove_callback

`add_checker`(*self*, *checker*)

[*Deprecated*]

Notes

Deprecated since version 3.3:

`autoscale`(*self*)

Autoscale the scalar limits on the norm instance using the current array

`autoscale_None`(*self*)

Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

property axes
: The *Axes* instance the artist resides in, or *None*.

changed(*self*)
: Call this whenever the mappable is changed to notify all the callbackSM listeners to the 'changed' signal.

check_update(*self*, *checker*)
: [*Deprecated*]

  Notes

  Deprecated since version 3.3:

contains(*self*, *mouseevent*)
: Test whether the mouse event occurred in the collection.

  Returns `bool, dict(ind=itemlist)`, where every item in itemlist contains the event.

convert_xunits(*self*, *x*)
: Convert *x* using the unit type of the xaxis.

  If the artist is not in contained in an Axes or if the xaxis does not have units, *x* itself is returned.

convert_yunits(*self*, *y*)
: Convert *y* using the unit type of the yaxis.

  If the artist is not in contained in an Axes or if the yaxis does not have units, *y* itself is returned.

draw(*self*, *renderer*)
: Draw the Artist (and its children) using the given renderer.

  This has no effect if the artist is not visible (*Artist.get_visible* returns False).

  Parameters

  renderer

  [*RendererBase* subclass.]

Notes

This method is overridden in the Artist subclasses.

`findobj`(*self*, *match=None*, *include_self=True*)
Find artist objects.

Recursively find all *Artist* instances contained in the artist.

Parameters

match

A filter criterion for the matches. This can be

- *None*: Return all objects contained in artist.
- A function with signature `def match(artist: Artist) -> bool`. The result will only contain artists for which the function returns *True*.
- A class instance: e.g., *Line2D*. The result will only contain artists of this class or its subclasses (`isinstance` check).

include_self

[bool] Include *self* in the list to be checked for a match.

Returns

list of *Artist*

`format_cursor_data`(*self*, *data*)
Return a string representation of *data*.

Note: This method is intended to be overridden by artist subclasses. As an end-user of Matplotlib you will most likely not call this method yourself.

The default implementation converts ints and floats and arrays of ints and floats into a comma-separated string enclosed in square brackets.

See also:

get_cursor_data

`get_agg_filter`(*self*)
Return filter function to be used for agg filter.

`get_alpha`(*self*)
Return the alpha value used for blending - not supported on all backends.

`get_animated`(*self*)
Return whether the artist is animated.

`get_array`(*self*)
Return the data array.

`get_capstyle`(*self*)

`get_children`(*self*)
Return a list of the child *Artist*s of this *Artist*.

`get_clim`(*self*)
Return the values (min, max) that are mapped to the colormap limits.

`get_clip_box`(*self*)
Return the clipbox.

`get_clip_on`(*self*)
Return whether the artist uses clipping.

`get_clip_path`(*self*)
Return the clip path.

`get_cmap`(*self*)
Return the *Colormap* instance.

`get_contains`(*self*)
[*Deprecated*] Return the custom contains function of the artist if set, or *None*.

See also:

set_contains

Notes

Deprecated since version 3.3.

`get_cursor_data`(*self*, *event*)
Return the cursor data for a given event.

Note: This method is intended to be overridden by artist subclasses. As an end-user of Matplotlib you will most likely not call this method yourself.

Cursor data can be used by Artists to provide additional context information for a given event. The default implementation just returns *None*.

Subclasses can override the method and return arbitrary data. However, when doing so, they must ensure that *format_cursor_data* can convert the data to a string representation.

The only current use case is displaying the z-value of an *AxesImage* in the status bar of a plot window, while moving the mouse.

Parameters

event

[*matplotlib.backend_bases.MouseEvent*]

See also:

format_cursor_data

`get_dashes`(*self*)
Alias for *get_linestyle*.

`get_datalim`(*self*, *transData*)

`get_ec`(*self*)
Alias for *get_edgecolor*.

`get_edgecolor`(*self*)

`get_edgecolors`(*self*)
Alias for *get_edgecolor*.

`get_facecolor`(*self*)

`get_facecolors`(*self*)
Alias for *get_facecolor*.

`get_fc`(*self*)
Alias for *get_facecolor*.

`get_figure`(*self*)
Return the *Figure* instance the artist belongs to.

`get_fill`(*self*)
Return whether fill is set.

`get_gid`(*self*)
Return the group id.

`get_hatch`(*self*)
Return the current hatching pattern.

`get_in_layout`(*self*)
Return boolean flag, `True` if artist is included in layout calculations.

E.g. *Constrained Layout Guide*, *Figure.tight_layout()*, and `fig.savefig(fname, bbox_inches='tight')`.

`get_joinstyle`(*self*)

`get_label`(*self*)
Return the label used for this artist in the legend.

`get_linestyle`(*self*)

`get_linestyles`(*self*)
Alias for *get_linestyle*.

`get_linewidth`(*self*)

`get_linewidths`(*self*)
Alias for *get_linewidth*.

`get_ls`(*self*)
Alias for *get_linestyle*.

`get_lw`(*self*)
Alias for *get_linewidth*.

`get_offset_position`(*self*)
[*Deprecated*] Return how offsets are applied for the collection. If *offset_position* is 'screen', the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is 'data', the offset is applied before the master transform, i.e., the offsets are in data coordinates.

Notes

Deprecated since version 3.3.

`get_offset_transform`(*self*)

`get_offsets`(*self*)
Return the offsets for the collection.

`get_path_effects`(*self*)

`get_paths`(*self*)

`get_picker`(*self*)
Return the picking behavior of the artist.

The possible values are described in *set_picker*.

See also:

set_picker, *pickable*, *pick*

`get_pickradius`(*self*)

`get_rasterized`(*self*)
Return whether the artist is to be rasterized.

`get_sizes`(*self*)
Return the sizes ('areas') of the elements in the collection.

Returns

array

The 'area' of each element.

`get_sketch_params`(*self*)
Return the sketch parameters for the artist.

Returns

tuple or None

A 3-tuple with the following elements:

- *scale*: The amplitude of the wiggle perpendicular to the source line.
- *length*: The length of the wiggle along the line.
- *randomness*: The scale factor by which the length is shrunken or expanded.

Returns *None* if no sketch parameters were set.

`get_snap`(*self*)
Return the snap setting.

See *set_snap* for details.

`get_tightbbox`(*self*, *renderer*)
Like *Artist.get_window_extent*, but includes any clipping.

Parameters

renderer

[*RendererBase* subclass] renderer that will be used to draw the figures (i.e. `fig.canvas.get_renderer()`)

Returns

Bbox

The enclosing bounding box (in figure pixel coordinates).

`get_transform`(*self*)
Return the *Transform* instance used by this artist.

`get_transformed_clip_path_and_affine`(*self*)
Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

`get_transforms`(*self*)

`get_url`(*self*)
Return the url.

`get_urls`(*self*)
Return a list of URLs, one for each element of the collection.

The list contains *None* for elements without a URL. See /gallery/misc/hyperlinks_sgskip for an example.

`get_visible`(*self*)
Return the visibility.

`get_window_extent`(*self*, *renderer*)
Get the axes bounding box in display space.

The bounding box' width and height are nonnegative.

Subclasses should override for inclusion in the bounding box "tight" calculation. Default is to return an empty bounding box at 0, 0.

Be careful when using this function, the results will not update if the artist window extent of the artist changes. The extent can change due to any changes in the transform stack, such as changing the axes limits, the figure size, or the canvas used (as is done when saving a figure). This can lead to unexpected behavior where interactive figures will look fine on the screen, but will save incorrectly.

`get_zorder`(*self*)
Return the artist's zorder.

`have_units`(*self*)
Return *True* if units are set on any axis.

`is_transform_set`(*self*)
Return whether the Artist has an explicitly set transform.

This is *True* after *set_transform* has been called.

`property mouseover`
If this property is set to *True*, the artist will be queried for custom context information when the mouse cursor moves over it.

See also *get_cursor_data()*, *ToolCursorPosition* and *NavigationToolbar2*.

`pchanged`(*self*)
Call all of the registered callbacks.

This function is triggered internally when a property is changed.

See also:

add_callback

remove_callback

`pick`(*self*, *mouseevent*)
Process a pick event.

Each child artist will fire a pick event if *mouseevent* is over the artist and the artist has picker set.

See also:

set_picker, *get_picker*, *pickable*

`pickable`(*self*)
Return whether the artist is pickable.

See also:

set_picker, *get_picker*, *pick*

`properties`(*self*)
Return a dictionary of all the properties of the artist.

`remove`(*self*)
Remove the artist from the figure if possible.

The effect will not be visible until the figure is redrawn, e.g., with *FigureCanvasBase.draw_idle*. Call *relim* to update the axes limits if desired.

Note: *relim* will not see collections even if the collection was added to the axes with *autolim* = True.

Note: there is no support for removing the artist's legend entry.

`remove_callback`(*self*, *oid*)
Remove a callback based on its observer id.

See also:

add_callback

`set`(*self*, ***kwargs*)
A property batch setter. Pass *kwargs* to set properties.

`set_aa`(*self*, *aa*)
Alias for *set_antialiased*.

`set_agg_filter`(*self*, *filter_func*)
Set the agg filter.

Parameters

filter_func

[callable] A filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array.

`set_alpha`(*self*, *alpha*)
Set the alpha value used for blending - not supported on all backends.

Parameters

alpha

[float or None]

`set_animated`(*self*, *b*)
Set the artist's animation state.

Parameters

b

[bool]

`set_antialiased`(*self*, *aa*)
Set the antialiasing state for rendering.

Parameters

aa

[bool or list of bools]

`set_antialiaseds`(*self*, *aa*)
Alias for *set_antialiased*.

`set_array`(*self*, *A*)
Set the image array from numpy array *A*.

Parameters

A

[ndarray]

`set_capstyle`(*self*, *cs*)
Set the capstyle for the collection (for all its elements).

Parameters

cs

[{'butt', 'round', 'projecting'}] The capstyle.

`set_clim`(*self*, *vmin=None*, *vmax=None*)
Set the norm limits for image scaling.

Parameters

vmin, vmax

[float] The limits.

The limits may also be passed as a tuple (*vmin*, *vmax*) as a single positional argument.

`set_clip_box`(*self*, *clipbox*)
Set the artist's clip *Bbox*.

Parameters

clipbox

[*Bbox*]

`set_clip_on`(*self*, *b*)
Set whether the artist uses clipping.

When False artists will be visible outside of the axes which can lead to unexpected results.

Parameters

b

[bool]

`set_clip_path`(*self*, *path*, *transform=None*)
Set the artist's clip path.

Parameters

path

[*Patch* or *Path* or *TransformedPath* or None] The clip path. If given a *Path*, *transform* must be provided as well. If *None*, a previously set clip path is removed.

transform

[*Transform*, optional] Only used if *path* is a *Path*, in which case the given *Path* is converted to a *TransformedPath* using *transform*.

Notes

For efficiency, if *path* is a *Rectangle* this method will set the clipping box to the corresponding rectangle and set the clipping path to `None`.

For technical reasons (support of *set*), a tuple (*path*, *transform*) is also accepted as a single positional parameter.

`set_cmap`(*self*, *cmap*)
Set the colormap for luminance data.

Parameters

cmap

[*Colormap* or str or None]

set_color(*self*, *c*)
Set both the edgecolor and the facecolor.

Parameters

c

[color or list of rgba tuples]

See also:

Collection.set_facecolor, *Collection.set_edgecolor*

For setting the edge or face color individually.

set_contains(*self*, *picker*)
[*Deprecated*] Define a custom contains test for the artist.

The provided callable replaces the default *contains* method of the artist.

Parameters

picker

[callable] A custom picker function to evaluate if an event is within the artist. The function must have the signature:

```
def contains(artist: Artist, event: MouseEvent) -> bool, dict
```

that returns:

- a bool indicating if the event is within the artist
- a dict of additional information. The dict should at least return the same information as the default `contains()` implementation of the respective artist, but may provide additional information.

Notes

Deprecated since version 3.3.

set_dashes(*self*, *ls*)
Alias for *set_linestyle*.

set_ec(*self*, *c*)
Alias for *set_edgecolor*.

set_edgecolor(*self*, *c*)
Set the edgecolor(s) of the collection.

Parameters

c

[color or list of colors or 'face'] The collection edgecolor(s). If a sequence, the patches cycle through it. If 'face', match the facecolor.

set_edgecolors(*self*, *c*)
Alias for *set_edgecolor*.

set_facecolor(*self*, *c*)
Set the facecolor(s) of the collection. *c* can be a color (all patches have same color), or a sequence of colors; if it is a sequence the patches will cycle through the sequence.

If *c* is 'none', the patch will not be filled.

Parameters

c

[color or list of colors]

set_facecolors(*self*, *c*)
Alias for *set_facecolor*.

set_fc(*self*, *c*)
Alias for *set_facecolor*.

set_figure(*self*, *fig*)
Set the *Figure* instance the artist belongs to.

Parameters

fig

[*Figure*]

set_gid(*self*, *gid*)
Set the (group) id for the artist.

Parameters

gid

[str]

set_hatch(*self*, *hatch*)
Set the hatching pattern

hatch can be one of:

```
/   - diagonal hatching
\   - back diagonal
|   - vertical
-   - horizontal
+   - crossed
x   - crossed diagonal
o   - small circle
O   - large circle
.   - dots
*   - stars
```

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Unlike other properties such as linewidth and colors, hatching can only be specified for the collection as a whole, not separately for each member.

Parameters

hatch

[{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}]

`set_in_layout`(*self*, *in_layout*)

Set if artist is to be included in layout calculations, E.g. *Constrained Layout Guide*, *`Figure.tight_layout()`*, and `fig.savefig(fname, bbox_inches='tight')`.

Parameters

in_layout

[bool]

`set_joinstyle`(*self*, *js*)

Set the joinstyle for the collection (for all its elements).

Parameters

js

[{'miter', 'round', 'bevel'}] The joinstyle.

`set_label`(*self*, *s*)

Set a label that will be displayed in the legend.

Parameters

s

[object] *s* will be converted to a string by calling `str`.

set_linestyle(*self*, *ls*)
Set the linestyle(s) for the collection.

linestyle	description
'-' or 'solid'	solid line
'--' or 'dashed'	dashed line
'-.' or 'dashdot'	dash-dotted line
':' or 'dotted'	dotted line

Alternatively a dash tuple of the following form can be provided:

```
(offset, onoffseq),
```

where `onoffseq` is an even length tuple of on and off ink in points.

Parameters

ls

[str or tuple or list thereof] Valid values for individual linestyles include {'-', '--', '-.', ':', '', (offset, on-off-seq)}. See *Line2D.set_linestyle* for a complete description.

set_linestyles(*self*, *ls*)
Alias for *set_linestyle*.

set_linewidth(*self*, *lw*)
Set the linewidth(s) for the collection. *lw* can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence

Parameters

lw

[float or list of floats]

set_linewidths(*self*, *lw*)
Alias for *set_linewidth*.

set_ls(*self*, *ls*)
Alias for *set_linestyle*.

set_lw(*self*, *lw*)
Alias for *set_linewidth*.

set_norm(*self*, *norm*)
Set the normalization instance.

Parameters

norm

[*Normalize* or None]

Notes

If there are any colorbars using the mappable for this norm, setting the norm of the mappable will reset the norm, locator, and formatters on the colorbar to default.

`set_offset_position`(*self*, *offset_position*)
[*Deprecated*] Set how offsets are applied. If *offset_position* is 'screen' (default) the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is 'data', the offset is applied before the master transform, i.e., the offsets are in data coordinates.

Parameters

offset_position

[{'screen', 'data'}]

Notes

Deprecated since version 3.3.

`set_offsets`(*self*, *offsets*)
Set the offsets for the collection.

Parameters

offsets

[array-like (N, 2) or (2,)]

`set_path_effects`(*self*, *path_effects*)
Set the path effects.

Parameters

path_effects

[*AbstractPathEffect*]

`set_paths`(*self*, *verts*, *closed=True*)
Set the vertices of the polygons.

Parameters

verts

[list of array-like] The sequence of polygons [*verts0*, *verts1*, ...] where each element *verts_i* defines the vertices of polygon *i* as a 2D array-like of shape (M, 2).

closed

[bool, default: True] Whether the polygon should be closed by adding a CLOSEPOLY connection at the end.

`set_picker`(*self*, *picker*)

Define the picking behavior of the artist.

Parameters

picker

[None or bool or callable] This can be one of the following:

- *None*: Picking is disabled for this artist (default).
- A boolean: If *True* then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist.
- A function: If picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

to determine the hit test. if the mouse event is over the artist, return *hit=True* and props is a dictionary of properties you want added to the PickEvent attributes.

- *deprecated*: For *Line2D* only, *picker* can also be a float that sets the tolerance for checking whether an event occurred "on" the line; this is deprecated. Use *Line2D.set_pickradius* instead.

`set_pickradius`(*self*, *pr*)

Set the pick radius used for containment tests.

Parameters

d

[float] Pick radius, in points.

`set_rasterized`(*self*, *rasterized*)

Force rasterized (bitmap) drawing in vector backend output.

Defaults to None, which implies the backend's default behavior.

Parameters

rasterized

[bool or None]

`set_sizes`(*self*, *sizes*, *dpi=72.0*)

Set the sizes of each member of the collection.

Parameters

sizes

[ndarray or None] The size to set for each element of the collection. The value is the 'area' of the element.

dpi

[float, default: 72] The dpi of the canvas.

`set_sketch_params`(*self*, *scale=None*, *length=None*, *randomness=None*)

Sets the sketch parameters.

Parameters

scale

[float, optional] The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is `None`, or not provided, no sketch filter will be provided.

length

[float, optional] The length of the wiggle along the line, in pixels (default 128.0)

randomness

[float, optional] The scale factor by which the length is shrunken or expanded (default 16.0)

`set_snap`(*self*, *snap*)

Set the snapping behavior.

Snapping aligns positions with the pixel grid, which results in clearer images. For example, if a black line of 1px width was defined at a position in between two pixels, the resulting image would contain the interpolated value of that line in the pixel grid, which would be a grey value on both adjacent pixel positions. In contrast, snapping will move the line to the nearest integer pixel value, so that the resulting image will really contain a 1px wide black line.

Snapping is currently only supported by the Agg and MacOSX backends.

Parameters

snap

[bool or None] Possible values:

- *True*: Snap vertices to the nearest pixel center.
- *False*: Do not modify vertex positions.
- *None*: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center.

`set_transform`(*self*, *t*)
Set the artist transform.

Parameters

t

[*Transform*]

`set_url`(*self*, *url*)
Set the url for the artist.

Parameters

url

[str]

`set_urls`(*self*, *urls*)

Parameters

urls

[list of str or None]

Notes

URLs are currently only implemented by the SVG backend. They are ignored by all other backends.

`set_verts`(*self*, *verts*, *closed=True*)
Set the vertices of the polygons.

Parameters

verts

[list of array-like] The sequence of polygons [*verts0*, *verts1*, ...] where each element *verts_i* defines the vertices of polygon *i* as a 2D array-like of shape (M, 2).

closed

[bool, default: True] Whether the polygon should be closed by adding a CLOSEPOLY connection at the end.

`set_verts_and_codes`(*self*, *verts*, *codes*)
Initialize vertices with path codes.

`set_visible`(*self*, *b*)
Set the artist's visibility.

Parameters

b

[bool]

`set_zorder`(*self*, *level*)
Set the zorder for the artist. Artists with lower zorder values are drawn first.

Parameters

level

[float]

`property stale`
Whether the artist is 'stale' and needs to be re-drawn for the output to match the internal state of the artist.

`property sticky_edges`
`x` and `y` sticky edge lists for autoscaling.

When performing autoscaling, if a data limit coincides with a value in the corresponding sticky_edges list, then no margin will be added--the view limit "sticks" to the edge. A typical use case is histograms, where one usually expects no margin on the bottom edge (0) of the histogram.

This attribute cannot be assigned to; however, the `x` and `y` lists can be modified in place as needed.

Examples

```
>>> artist.sticky_edges.x[:] = (xmin, xmax)
>>> artist.sticky_edges.y[:] = (ymin, ymax)
```

`to_rgba`(*self*, *x*, *alpha=None*, *bytes=False*, *norm=True*)
Return a normalized rgba array corresponding to *x*.

In the normal case, *x* is a 1-D or 2-D sequence of scalars, and the corresponding ndarray of rgba values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already rgb or rgba, such as might have been read from an image file. If *x* is an ndarray with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an rgb or rgba array, and no mapping will be done. The array can be uint8, or it can be floating point with values in the 0-1 range; otherwise a ValueError will be raised. If it is a masked array, the mask will be ignored. If the last dimension is 3, the *alpha* kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the *alpha* kwarg is ignored; it does not replace the pre-existing alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if *bytes* is *False* (default), the rgba array will be floats in the 0-1 range; if it is *True*, the returned rgba array will be uint8 in the 0 to 255 range.

If norm is False, no normalization of the input data is performed, and it is assumed to be in the range (0-1).

`update`(*self*, *props*)
Update this artist's properties from the dict *props*.

Parameters

props

[dict]

`property update_dict`

`update_from`(*self*, *other*)
Copy properties from other to self.

`update_scalarmappable`(*self*)
Update colors from the scalar mappable array, if it is not None.

`zorder = 0`

`class matplotlib.collections.QuadMesh`(*meshWidth*, *meshHeight*, *coordinates*, *antialiased=True*, *shading='flat'*, ***kwargs*)
Bases: *matplotlib.collections.Collection*

Class for the efficient drawing of a quadrilateral mesh.

A quadrilateral mesh consists of a grid of vertices. The dimensions of this array are (*meshWidth* + 1, *meshHeight* + 1). Each vertex in the mesh has a different set of "mesh coordinates" representing its position in the topology of the mesh. For any values (*m*, *n*) such that 0 <= *m* <= *meshWidth* and 0 <= *n* <= *meshHeight*, the vertices at mesh coordinates (*m*, *n*), (*m*, *n* + 1), (*m* + 1, *n* + 1), and (*m* + 1, *n*) form one of the quadrilaterals in the mesh. There are thus (*meshWidth* *

meshHeight) quadrilaterals in the mesh. The mesh need not be regular and the polygons need not be convex.

A quadrilateral mesh is represented by a (2 x ((*meshWidth* + 1) * (*meshHeight* + 1))) numpy array *coordinates*, where each row is the *x* and *y* coordinates of one of the vertices. To define the function that maps from a data point to its corresponding color, use the *set_cmap()* method. Each of these arrays is indexed in row-major order by the mesh coordinates of the vertex (or the mesh coordinates of the lower left vertex, in the case of the colors).

For example, the first entry in *coordinates* is the coordinates of the vertex at mesh coordinates (0, 0), then the one at (0, 1), then at (0, 2) .. (0, meshWidth), (1, 0), (1, 1), and so on.

shading may be 'flat', or 'gouraud'

Parameters

edgecolors

[color or list of colors, default: `rcParams["patch.edgecolor"]` (default: `'black'`)] Edge color for each patch making up the collection. The special value 'face' can be passed to make the edgecolor match the facecolor.

facecolors

[color or list of colors, default: `rcParams["patch.facecolor"]` (default: `'C0'`)] Face color for each patch making up the collection.

linewidths

[float or list of floats, default: `rcParams["patch.linewidth"]` (default: `1.0`)] Line width for each patch making up the collection.

linestyles

[str or tuple or list thereof, default: 'solid'] Valid strings are ['solid', 'dashed', 'dashdot', 'dotted', '-', '--', '-.', ':']. Dash tuples should be of the form:

```
(offset, onoffseq),
```

where *onoffseq* is an even length tuple of on and off ink lengths in points. For examples, see /gallery/lines_bars_and_markers/linestyles.

capstyle

[str, default: `rcParams["patch.capstyle"]`] Style to use for capping lines for all paths in the collection. See /gallery/lines_bars_and_markers/joinstyle for a demonstration of each of the allowed values.

joinstyle

[str, default: `rcParams["patch.joinstyle"]`] Style to use for joining lines for all paths in the collection. See /gallery/lines_bars_and_markers/joinstyle for a demonstration of each of the allowed values.

antialiaseds

[bool or list of bool, default: `rcParams["patch.antialiased"]` (default: `True`)] Whether each pach in the collection should be drawn with antialiasing.

offsets

[(float, float) or list thereof, default: (0, 0)] A vector by which to translate each patch after rendering (default is no translation). The translation is performed in screen (pixel) coordinates (i.e. after the Artist's transform is applied).

transOffset

[*Transform*, default: *IdentityTransform*] A single transform which will be applied to each *offsets* vector before it is used.

offset_position

[{'screen' (default), 'data' (deprecated)}] If set to 'data' (deprecated), *offsets* will be treated as if it is in data coordinates instead of in screen coordinates.

norm

[*Normalize*, optional] Forwarded to *ScalarMappable*. The default of `None` means that the first draw call will set `vmin` and `vmax` using the minimum and maximum values of the data.

cmap

[*Colormap*, optional] Forwarded to *ScalarMappable*. The default of `None` will result in `rcParams["image.cmap"]` (default: `'viridis'`) being used.

hatch

[str, optional] Hatching pattern to use in filled paths, if any. Valid strings are ['/', '', '|', '-', '+', 'x', 'o', 'O', '.', '*']. See /gallery/shapes_and_collections/hatch_demo for the meaning of each hatch type.

pickradius

[float, default: 5.0] If `pickradius <= 0`, then *Collection.contains* will return `True` whenever the test point is inside of one of the polygons formed by the control points of a Path in the Collection. On the other hand, if it is greater than 0, then we instead check

if the test point is contained in a stroke of width `2*pickradius` following any of the Paths in the Collection.

urls

[list of str, default: None] A URL for each patch to link to once drawn. Currently only works for the SVG backend. See /gallery/misc/hyperlinks_sgskip for examples.

zorder

[float, default: 1] The drawing order, shared by all Patches in the Collection. See /gallery/misc/zorder_demo for all defaults and examples.

`add_callback`(*self*, *func*)

Add a callback function that will be called whenever one of the *Artist*'s properties changes.

Parameters

func

[callable] The callback function. It must have the signature:

```
def func(artist: Artist) -> Any
```

where *artist* is the calling *Artist*. Return values may exist but are ignored.

Returns

int

The observer id associated with the callback. This id can be used for removing the callback with *remove_callback* later.

See also:

remove_callback

`add_checker`(*self*, *checker*)

[*Deprecated*]

Notes

Deprecated since version 3.3:

`autoscale`(*self*)
Autoscale the scalar limits on the norm instance using the current array

`autoscale_None`(*self*)
Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

`property axes`
The *Axes* instance the artist resides in, or *None*.

`changed`(*self*)
Call this whenever the mappable is changed to notify all the callbackSM listeners to the 'changed' signal.

`check_update`(*self*, *checker*)
[*Deprecated*]

Notes

Deprecated since version 3.3:

`contains`(*self*, *mouseevent*)
Test whether the mouse event occurred in the collection.

Returns `bool, dict(ind=itemlist)`, where every item in itemlist contains the event.

`static convert_mesh_to_paths`(*meshWidth*, *meshHeight*, *coordinates*)
Convert a given mesh into a sequence of *Path* objects.

This function is primarily of use to implementers of backends that do not directly support quadmeshes.

`convert_mesh_to_triangles`(*self*, *meshWidth*, *meshHeight*, *coordinates*)
Convert a given mesh into a sequence of triangles, each point with its own color. This is useful for experiments using *draw_gouraud_triangle*.

`convert_xunits`(*self*, *x*)
Convert *x* using the unit type of the xaxis.

If the artist is not in contained in an Axes or if the xaxis does not have units, *x* itself is returned.

`convert_yunits`(*self*, *y*)
Convert *y* using the unit type of the yaxis.

If the artist is not in contained in an Axes or if the yaxis does not have units, *y* itself is returned.

`draw`(*self*, *renderer*)
Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

Parameters

renderer

[*RendererBase* subclass.]

Notes

This method is overridden in the Artist subclasses.

`findobj`(*self*, *match=None*, *include_self=True*)
Find artist objects.

Recursively find all *Artist* instances contained in the artist.

Parameters

match

A filter criterion for the matches. This can be

- *None*: Return all objects contained in artist.
- A function with signature `def match(artist: Artist) -> bool`. The result will only contain artists for which the function returns *True*.
- A class instance: e.g., *Line2D*. The result will only contain artists of this class or its subclasses (`isinstance` check).

include_self

[bool] Include *self* in the list to be checked for a match.

Returns

list of *Artist*

`format_cursor_data`(*self*, *data*)
Return a string representation of *data*.

Note: This method is intended to be overridden by artist subclasses. As an end-user of Matplotlib you will most likely not call this method yourself.

The default implementation converts ints and floats and arrays of ints and floats into a comma-separated string enclosed in square brackets.

See also:

get_cursor_data

get_agg_filter(*self*)
Return filter function to be used for agg filter.

get_alpha(*self*)
Return the alpha value used for blending - not supported on all backends.

get_animated(*self*)
Return whether the artist is animated.

get_array(*self*)
Return the data array.

get_capstyle(*self*)

get_children(*self*)
Return a list of the child *Artist*s of this *Artist*.

get_clim(*self*)
Return the values (min, max) that are mapped to the colormap limits.

get_clip_box(*self*)
Return the clipbox.

get_clip_on(*self*)
Return whether the artist uses clipping.

get_clip_path(*self*)
Return the clip path.

get_cmap(*self*)
Return the *Colormap* instance.

get_contains(*self*)
[*Deprecated*] Return the custom contains function of the artist if set, or *None*.

See also:

set_contains

Notes

Deprecated since version 3.3.

get_cursor_data(*self*, *event*)
Return the cursor data for a given event.

Note: This method is intended to be overridden by artist subclasses. As an end-user of Matplotlib you will most likely not call this method yourself.

Cursor data can be used by Artists to provide additional context information for a given event. The default implementation just returns *None*.

Subclasses can override the method and return arbitrary data. However, when doing so, they must ensure that *format_cursor_data* can convert the data to a string representation.

The only current use case is displaying the z-value of an *AxesImage* in the status bar of a plot window, while moving the mouse.

Parameters

event

[*matplotlib.backend_bases.MouseEvent*]

See also:

format_cursor_data

`get_dashes`(*self*)
Alias for *get_linestyle*.

`get_datalim`(*self*, *transData*)

`get_ec`(*self*)
Alias for *get_edgecolor*.

`get_edgecolor`(*self*)

`get_edgecolors`(*self*)
Alias for *get_edgecolor*.

`get_facecolor`(*self*)

`get_facecolors`(*self*)
Alias for *get_facecolor*.

`get_fc`(*self*)
Alias for *get_facecolor*.

`get_figure`(*self*)
Return the *Figure* instance the artist belongs to.

`get_fill`(*self*)
Return whether fill is set.

`get_gid`(*self*)
Return the group id.

get_hatch(*self*)
Return the current hatching pattern.

get_in_layout(*self*)
Return boolean flag, `True` if artist is included in layout calculations.

E.g. *Constrained Layout Guide*, *Figure.tight_layout()*, and `fig.savefig(fname, bbox_inches='tight')`.

get_joinstyle(*self*)

get_label(*self*)
Return the label used for this artist in the legend.

get_linestyle(*self*)

get_linestyles(*self*)
Alias for *get_linestyle*.

get_linewidth(*self*)

get_linewidths(*self*)
Alias for *get_linewidth*.

get_ls(*self*)
Alias for *get_linestyle*.

get_lw(*self*)
Alias for *get_linewidth*.

get_offset_position(*self*)
[*Deprecated*] Return how offsets are applied for the collection. If *offset_position* is 'screen', the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is 'data', the offset is applied before the master transform, i.e., the offsets are in data coordinates.

Notes

Deprecated since version 3.3.

get_offset_transform(*self*)

get_offsets(*self*)
Return the offsets for the collection.

get_path_effects(*self*)

get_paths(*self*)

get_picker(*self*)
Return the picking behavior of the artist.

The possible values are described in *set_picker*.

See also:

set_picker, *pickable*, *pick*

get_pickradius(*self*)

get_rasterized(*self*)
Return whether the artist is to be rasterized.

get_sketch_params(*self*)
Return the sketch parameters for the artist.

Returns

tuple or None

A 3-tuple with the following elements:

- *scale*: The amplitude of the wiggle perpendicular to the source line.
- *length*: The length of the wiggle along the line.
- *randomness*: The scale factor by which the length is shrunken or expanded.

Returns *None* if no sketch parameters were set.

get_snap(*self*)
Return the snap setting.

See *set_snap* for details.

get_tightbbox(*self*, *renderer*)
Like *Artist.get_window_extent*, but includes any clipping.

Parameters

renderer

[*RendererBase* subclass] renderer that will be used to draw the figures (i.e. `fig.canvas.get_renderer()`)

Returns

Bbox

The enclosing bounding box (in figure pixel coordinates).

get_transform(*self*)
Return the *Transform* instance used by this artist.

get_transformed_clip_path_and_affine(*self*)
Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

get_transforms(*self*)

`get_url`(*self*)
Return the url.

`get_urls`(*self*)
Return a list of URLs, one for each element of the collection.

The list contains *None* for elements without a URL. See /gallery/misc/hyperlinks_sgskip for an example.

`get_visible`(*self*)
Return the visibility.

`get_window_extent`(*self*, *renderer*)
Get the axes bounding box in display space.

The bounding box' width and height are nonnegative.

Subclasses should override for inclusion in the bounding box "tight" calculation. Default is to return an empty bounding box at 0, 0.

Be careful when using this function, the results will not update if the artist window extent of the artist changes. The extent can change due to any changes in the transform stack, such as changing the axes limits, the figure size, or the canvas used (as is done when saving a figure). This can lead to unexpected behavior where interactive figures will look fine on the screen, but will save incorrectly.

`get_zorder`(*self*)
Return the artist's zorder.

`have_units`(*self*)
Return *True* if units are set on any axis.

`is_transform_set`(*self*)
Return whether the Artist has an explicitly set transform.

This is *True* after *set_transform* has been called.

`property mouseover`
If this property is set to *True*, the artist will be queried for custom context information when the mouse cursor moves over it.

See also *get_cursor_data()*, *ToolCursorPosition* and *NavigationToolbar2*.

`pchanged`(*self*)
Call all of the registered callbacks.

This function is triggered internally when a property is changed.

See also:

add_callback

remove_callback

pick(*self*, *mouseevent*)
Process a pick event.

Each child artist will fire a pick event if *mouseevent* is over the artist and the artist has picker set.

See also:

set_picker, *get_picker*, *pickable*

pickable(*self*)
Return whether the artist is pickable.

See also:

set_picker, *get_picker*, *pick*

properties(*self*)
Return a dictionary of all the properties of the artist.

remove(*self*)
Remove the artist from the figure if possible.

The effect will not be visible until the figure is redrawn, e.g., with *FigureCanvasBase.draw_idle*. Call *relim* to update the axes limits if desired.

Note: *relim* will not see collections even if the collection was added to the axes with *autolim* = True.

Note: there is no support for removing the artist's legend entry.

remove_callback(*self*, *oid*)
Remove a callback based on its observer id.

See also:

add_callback

set(*self*, ***kwargs*)
A property batch setter. Pass *kwargs* to set properties.

set_aa(*self*, *aa*)
Alias for *set_antialiased*.

set_agg_filter(*self*, *filter_func*)
Set the agg filter.

Parameters

filter_func

[callable] A filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array.

`set_alpha`(*self*, *alpha*)
Set the alpha value used for blending - not supported on all backends.

Parameters

alpha

[float or None]

`set_animated`(*self*, *b*)
Set the artist's animation state.

Parameters

b

[bool]

`set_antialiased`(*self*, *aa*)
Set the antialiasing state for rendering.

Parameters

aa

[bool or list of bools]

`set_antialiaseds`(*self*, *aa*)
Alias for *set_antialiased*.

`set_array`(*self*, *A*)
Set the image array from numpy array *A*.

Parameters

A

[ndarray]

`set_capstyle`(*self*, *cs*)
Set the capstyle for the collection (for all its elements).

Parameters

cs

[{'butt', 'round', 'projecting'}] The capstyle.

`set_clim`(*self*, *vmin=None*, *vmax=None*)
Set the norm limits for image scaling.

Parameters

vmin, vmax

[float] The limits.

The limits may also be passed as a tuple (*vmin*, *vmax*) as a single positional argument.

`set_clip_box`(*self*, *clipbox*)
Set the artist's clip *Bbox*.

Parameters

clipbox

[*Bbox*]

`set_clip_on`(*self*, *b*)
Set whether the artist uses clipping.

When False artists will be visible outside of the axes which can lead to unexpected results.

Parameters

b

[bool]

`set_clip_path`(*self*, *path*, *transform=None*)
Set the artist's clip path.

Parameters

path

[*Patch* or *Path* or *TransformedPath* or None] The clip path. If given a *Path*, *transform* must be provided as well. If *None*, a previously set clip path is removed.

transform

[*Transform*, optional] Only used if *path* is a *Path*, in which case the given *Path* is converted to a *TransformedPath* using *transform*.

Notes

For efficiency, if *path* is a *Rectangle* this method will set the clipping box to the corresponding rectangle and set the clipping path to `None`.

For technical reasons (support of *set*), a tuple (*path*, *transform*) is also accepted as a single positional parameter.

`set_cmap`(*self*, *cmap*)
Set the colormap for luminance data.

Parameters

cmap

[*Colormap* or str or None]

`set_color`(*self*, *c*)
Set both the edgecolor and the facecolor.

Parameters

c

[color or list of rgba tuples]

See also:

Collection.set_facecolor, *Collection.set_edgecolor*

For setting the edge or face color individually.

`set_contains`(*self*, *picker*)
[*Deprecated*] Define a custom contains test for the artist.

The provided callable replaces the default *contains* method of the artist.

Parameters

picker

[callable] A custom picker function to evaluate if an event is within the artist. The function must have the signature:

```
def contains(artist: Artist, event: MouseEvent) -> bool, dict
```

that returns:

- a bool indicating if the event is within the artist
- a dict of additional information. The dict should at least return the same information as the default `contains()` implementation of the respective artist, but may provide additional information.

Notes

Deprecated since version 3.3.

`set_dashes`(*self*, *ls*)
Alias for *set_linestyle*.

`set_ec`(*self*, *c*)
Alias for *set_edgecolor*.

`set_edgecolor`(*self*, *c*)
Set the edgecolor(s) of the collection.

Parameters

c

[color or list of colors or 'face'] The collection edgecolor(s). If a sequence, the patches cycle through it. If 'face', match the facecolor.

`set_edgecolors`(*self*, *c*)
Alias for *set_edgecolor*.

`set_facecolor`(*self*, *c*)
Set the facecolor(s) of the collection. *c* can be a color (all patches have same color), or a sequence of colors; if it is a sequence the patches will cycle through the sequence.

If *c* is 'none', the patch will not be filled.

Parameters

c

[color or list of colors]

`set_facecolors`(*self*, *c*)
Alias for *set_facecolor*.

`set_fc`(*self*, *c*)
Alias for *set_facecolor*.

`set_figure`(*self*, *fig*)
Set the *Figure* instance the artist belongs to.

Parameters

fig

[*Figure*]

`set_gid`(*self*, *gid*)
Set the (group) id for the artist.

Parameters

gid

[str]

set_hatch(*self*, *hatch*)

Set the hatching pattern

hatch can be one of:

```
/   - diagonal hatching
\   - back diagonal
|   - vertical
-   - horizontal
+   - crossed
x   - crossed diagonal
o   - small circle
O   - large circle
.   - dots
*   - stars
```

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Unlike other properties such as linewidth and colors, hatching can only be specified for the collection as a whole, not separately for each member.

Parameters

hatch

[{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}]

set_in_layout(*self*, *in_layout*)

Set if artist is to be included in layout calculations, E.g. *Constrained Layout Guide*, *Figure.tight_layout()*, and `fig.savefig(fname, bbox_inches='tight')`.

Parameters

in_layout

[bool]

set_joinstyle(*self*, *js*)

Set the joinstyle for the collection (for all its elements).

Parameters

js

[{'miter', 'round', 'bevel'}] The joinstyle.

`set_label`(*self*, *s*)
Set a label that will be displayed in the legend.

Parameters

s

[object] *s* will be converted to a string by calling `str`.

`set_linestyle`(*self*, *ls*)
Set the linestyle(s) for the collection.

linestyle	description
`'-'` or `'solid'`	solid line
`'--'` or `'dashed'`	dashed line
`'-.'` or `'dashdot'`	dash-dotted line
`':'` or `'dotted'`	dotted line

Alternatively a dash tuple of the following form can be provided:

```
(offset, onoffseq),
```

where `onoffseq` is an even length tuple of on and off ink in points.

Parameters

ls

[str or tuple or list thereof] Valid values for individual linestyles include {'-', '--', '-.', ':', '', (offset, on-off-seq)}. See *Line2D.set_linestyle* for a complete description.

`set_linestyles`(*self*, *ls*)
Alias for *set_linestyle*.

`set_linewidth`(*self*, *lw*)
Set the linewidth(s) for the collection. *lw* can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence

Parameters

lw

[float or list of floats]

`set_linewidths`(*self*, *lw*)
Alias for *set_linewidth*.

`set_ls`(*self*, *ls*)
Alias for *set_linestyle*.

`set_lw`(*self*, *lw*)
Alias for *set_linewidth*.

`set_norm`(*self*, *norm*)
Set the normalization instance.

Parameters

norm

[*Normalize* or None]

Notes

If there are any colorbars using the mappable for this norm, setting the norm of the mappable will reset the norm, locator, and formatters on the colorbar to default.

`set_offset_position`(*self*, *offset_position*)
[*Deprecated*] Set how offsets are applied. If *offset_position* is 'screen' (default) the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is 'data', the offset is applied before the master transform, i.e., the offsets are in data coordinates.

Parameters

offset_position

[{'screen', 'data'}]

Notes

Deprecated since version 3.3.

`set_offsets`(*self*, *offsets*)
Set the offsets for the collection.

Parameters

offsets

[array-like (N, 2) or (2,)]

`set_path_effects`(*self*, *path_effects*)
Set the path effects.

Parameters

path_effects

[*AbstractPathEffect*]

`set_paths`(*self*)

`set_picker`(*self*, *picker*)

Define the picking behavior of the artist.

Parameters

picker

[None or bool or callable] This can be one of the following:

- *None*: Picking is disabled for this artist (default).
- A boolean: If *True* then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist.
- A function: If picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

to determine the hit test. if the mouse event is over the artist, return *hit=True* and props is a dictionary of properties you want added to the PickEvent attributes.

- *deprecated*: For *Line2D* only, *picker* can also be a float that sets the tolerance for checking whether an event occurred "on" the line; this is deprecated. Use *Line2D.set_pickradius* instead.

`set_pickradius`(*self*, *pr*)

Set the pick radius used for containment tests.

Parameters

d

[float] Pick radius, in points.

`set_rasterized`(*self*, *rasterized*)

Force rasterized (bitmap) drawing in vector backend output.

Defaults to None, which implies the backend's default behavior.

Parameters

rasterized

[bool or None]

`set_sketch_params`(*self*, *scale=None*, *length=None*, *randomness=None*)
Sets the sketch parameters.

Parameters

scale

[float, optional] The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is `None`, or not provided, no sketch filter will be provided.

length

[float, optional] The length of the wiggle along the line, in pixels (default 128.0)

randomness

[float, optional] The scale factor by which the length is shrunken or expanded (default 16.0)

`set_snap`(*self*, *snap*)
Set the snapping behavior.

Snapping aligns positions with the pixel grid, which results in clearer images. For example, if a black line of 1px width was defined at a position in between two pixels, the resulting image would contain the interpolated value of that line in the pixel grid, which would be a grey value on both adjacent pixel positions. In contrast, snapping will move the line to the nearest integer pixel value, so that the resulting image will really contain a 1px wide black line.

Snapping is currently only supported by the Agg and MacOSX backends.

Parameters

snap

[bool or None] Possible values:

- *True*: Snap vertices to the nearest pixel center.
- *False*: Do not modify vertex positions.
- *None*: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center.

`set_transform`(*self*, *t*)
Set the artist transform.

Parameters

t

[*Transform*]

set_url(*self*, *url*)
Set the url for the artist.

Parameters

url

[str]

set_urls(*self*, *urls*)

Parameters

urls

[list of str or None]

Notes

URLs are currently only implemented by the SVG backend. They are ignored by all other backends.

set_visible(*self*, *b*)
Set the artist's visibility.

Parameters

b

[bool]

set_zorder(*self*, *level*)
Set the zorder for the artist. Artists with lower zorder values are drawn first.

Parameters

level

[float]

property stale
Whether the artist is 'stale' and needs to be re-drawn for the output to match the internal state of the artist.

property sticky_edges
`x` and `y` sticky edge lists for autoscaling.

When performing autoscaling, if a data limit coincides with a value in the corresponding sticky_edges list, then no margin will be added--the view limit "sticks" to the edge. A typical use case is histograms, where one usually expects no margin on the bottom edge (0) of the histogram.

This attribute cannot be assigned to; however, the `x` and `y` lists can be modified in place as needed.

Examples

```
>>> artist.sticky_edges.x[:] = (xmin, xmax)
>>> artist.sticky_edges.y[:] = (ymin, ymax)
```

`to_rgba`(*self*, *x*, *alpha=None*, *bytes=False*, *norm=True*)
Return a normalized rgba array corresponding to *x*.

In the normal case, *x* is a 1-D or 2-D sequence of scalars, and the corresponding ndarray of rgba values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already rgb or rgba, such as might have been read from an image file. If *x* is an ndarray with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an rgb or rgba array, and no mapping will be done. The array can be uint8, or it can be floating point with values in the 0-1 range; otherwise a ValueError will be raised. If it is a masked array, the mask will be ignored. If the last dimension is 3, the *alpha* kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the *alpha* kwarg is ignored; it does not replace the pre-existing alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if *bytes* is *False* (default), the rgba array will be floats in the 0-1 range; if it is *True*, the returned rgba array will be uint8 in the 0 to 255 range.

If norm is False, no normalization of the input data is performed, and it is assumed to be in the range (0-1).

`update`(*self*, *props*)
Update this artist's properties from the dict *props*.

Parameters

props

[dict]

`property update_dict`

`update_from`(*self*, *other*)
Copy properties from other to self.

`update_scalarmappable`(*self*)
Update colors from the scalar mappable array, if it is not None.

`zorder = 0`

class matplotlib.collections.**RegularPolyCollection**(*numsides*, *rotation=0*, *sizes=1*, ***kwargs*)

Bases: matplotlib.collections._CollectionWithSizes

A collection of n-sided regular polygons.

Parameters

numsides

[int] The number of sides of the polygon.

rotation

[float] The rotation of the polygon in radians.

sizes

[tuple of float] The area of the circle circumscribing the polygon in points^2.

****kwargs**

Forwarded to *Collection*.

Examples

See /gallery/event_handling/lasso_demo for a complete example:

```
offsets = np.random.rand(20, 2)
facecolors = [cm.jet(x) for x in np.random.rand(20)]

collection = RegularPolyCollection(
    numsides=5, # a pentagon
    rotation=0, sizes=(50,),
    facecolors=facecolors,
    edgecolors=("black",),
    linewidths=(1,),
    offsets=offsets,
    transOffset=ax.transData,
    )
```

add_callback(*self*, *func*)

Add a callback function that will be called whenever one of the *Artist*'s properties changes.

Parameters

func

[callable] The callback function. It must have the signature:

```
def func(artist: Artist) -> Any
```

where *artist* is the calling *Artist*. Return values may exist but are ignored.

Returns

int

The observer id associated with the callback. This id can be used for removing the callback with *remove_callback* later.

See also:

remove_callback

`add_checker`(*self*, *checker*)
[*Deprecated*]

Notes

Deprecated since version 3.3:

`autoscale`(*self*)
Autoscale the scalar limits on the norm instance using the current array

`autoscale_None`(*self*)
Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

`property axes`
The *Axes* instance the artist resides in, or *None*.

`changed`(*self*)
Call this whenever the mappable is changed to notify all the callbackSM listeners to the 'changed' signal.

`check_update`(*self*, *checker*)
[*Deprecated*]

Notes

Deprecated since version 3.3:

`contains`(*self*, *mouseevent*)
Test whether the mouse event occurred in the collection.

Returns `bool, dict(ind=itemlist)`, where every item in itemlist contains the event.

`convert_xunits`(*self*, *x*)
Convert *x* using the unit type of the xaxis.

If the artist is not in contained in an Axes or if the xaxis does not have units, *x* itself is returned.

`convert_yunits`(*self*, *y*)
Convert *y* using the unit type of the yaxis.

If the artist is not in contained in an Axes or if the yaxis does not have units, *y* itself is returned.

`draw`(*self*, *renderer*)
Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

Parameters

renderer

[*RendererBase* subclass.]

Notes

This method is overridden in the Artist subclasses.

`findobj`(*self*, *match=None*, *include_self=True*)
Find artist objects.

Recursively find all *Artist* instances contained in the artist.

Parameters

match

A filter criterion for the matches. This can be

- *None*: Return all objects contained in artist.
- A function with signature `def match(artist: Artist) -> bool`. The result will only contain artists for which the function returns *True*.
- A class instance: e.g., *Line2D*. The result will only contain artists of this class or its subclasses (`isinstance` check).

include_self

[bool] Include *self* in the list to be checked for a match.

Returns

list of *Artist*

format_cursor_data(*self*, *data*)
Return a string representation of *data*.

Note: This method is intended to be overridden by artist subclasses. As an end-user of Matplotlib you will most likely not call this method yourself.

The default implementation converts ints and floats and arrays of ints and floats into a comma-separated string enclosed in square brackets.

See also:

get_cursor_data

get_agg_filter(*self*)
Return filter function to be used for agg filter.

get_alpha(*self*)
Return the alpha value used for blending - not supported on all backends.

get_animated(*self*)
Return whether the artist is animated.

get_array(*self*)
Return the data array.

get_capstyle(*self*)

get_children(*self*)
Return a list of the child *Artist*s of this *Artist*.

get_clim(*self*)
Return the values (min, max) that are mapped to the colormap limits.

get_clip_box(*self*)
Return the clipbox.

get_clip_on(*self*)
Return whether the artist uses clipping.

get_clip_path(*self*)
Return the clip path.

get_cmap(*self*)
Return the *Colormap* instance.

get_contains(*self*)
[*Deprecated*] Return the custom contains function of the artist if set, or *None*.

See also:

set_contains

Notes

Deprecated since version 3.3.

get_cursor_data(*self*, *event*)
Return the cursor data for a given event.

Note: This method is intended to be overridden by artist subclasses. As an end-user of Matplotlib you will most likely not call this method yourself.

Cursor data can be used by Artists to provide additional context information for a given event. The default implementation just returns *None*.

Subclasses can override the method and return arbitrary data. However, when doing so, they must ensure that *format_cursor_data* can convert the data to a string representation.

The only current use case is displaying the z-value of an *AxesImage* in the status bar of a plot window, while moving the mouse.

Parameters

event

[*matplotlib.backend_bases.MouseEvent*]

See also:

format_cursor_data

get_dashes(*self*)
Alias for *get_linestyle*.

get_datalim(*self*, *transData*)

get_ec(*self*)
Alias for *get_edgecolor*.

get_edgecolor(*self*)

get_edgecolors(*self*)
Alias for *get_edgecolor*.

get_facecolor(*self*)

get_facecolors(*self*)
Alias for *get_facecolor*.

get_fc(*self*)
Alias for *get_facecolor*.

get_figure(*self*)
Return the *Figure* instance the artist belongs to.

`get_fill`(*self*)
Return whether fill is set.

`get_gid`(*self*)
Return the group id.

`get_hatch`(*self*)
Return the current hatching pattern.

`get_in_layout`(*self*)
Return boolean flag, `True` if artist is included in layout calculations.

E.g. *Constrained Layout Guide*, *Figure.tight_layout()*, and `fig.savefig(fname, bbox_inches='tight')`.

`get_joinstyle`(*self*)

`get_label`(*self*)
Return the label used for this artist in the legend.

`get_linestyle`(*self*)

`get_linestyles`(*self*)
Alias for *get_linestyle*.

`get_linewidth`(*self*)

`get_linewidths`(*self*)
Alias for *get_linewidth*.

`get_ls`(*self*)
Alias for *get_linestyle*.

`get_lw`(*self*)
Alias for *get_linewidth*.

`get_numsides`(*self*)

`get_offset_position`(*self*)
[*Deprecated*] Return how offsets are applied for the collection. If *offset_position* is 'screen', the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is 'data', the offset is applied before the master transform, i.e., the offsets are in data coordinates.

Notes

Deprecated since version 3.3.

`get_offset_transform`(*self*)

`get_offsets`(*self*)
Return the offsets for the collection.

`get_path_effects`(*self*)

get_paths(*self*)

get_picker(*self*)
Return the picking behavior of the artist.

The possible values are described in *set_picker*.

See also:

set_picker, *pickable*, *pick*

get_pickradius(*self*)

get_rasterized(*self*)
Return whether the artist is to be rasterized.

get_rotation(*self*)

get_sizes(*self*)
Return the sizes ('areas') of the elements in the collection.

Returns

array

The 'area' of each element.

get_sketch_params(*self*)
Return the sketch parameters for the artist.

Returns

tuple or None

A 3-tuple with the following elements:

- *scale*: The amplitude of the wiggle perpendicular to the source line.
- *length*: The length of the wiggle along the line.
- *randomness*: The scale factor by which the length is shrunken or expanded.

Returns *None* if no sketch parameters were set.

get_snap(*self*)
Return the snap setting.

See *set_snap* for details.

get_tightbbox(*self*, *renderer*)
Like *Artist.get_window_extent*, but includes any clipping.

Parameters

renderer

[*RendererBase* subclass] renderer that will be used to draw the figures (i.e. `fig.canvas.get_renderer()`)

Returns

Bbox

The enclosing bounding box (in figure pixel coordinates).

`get_transform`(*self*)
Return the *Transform* instance used by this artist.

`get_transformed_clip_path_and_affine`(*self*)
Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

`get_transforms`(*self*)

`get_url`(*self*)
Return the url.

`get_urls`(*self*)
Return a list of URLs, one for each element of the collection.

The list contains *None* for elements without a URL. See /gallery/misc/hyperlinks_sgskip for an example.

`get_visible`(*self*)
Return the visibility.

`get_window_extent`(*self*, *renderer*)
Get the axes bounding box in display space.

The bounding box' width and height are nonnegative.

Subclasses should override for inclusion in the bounding box "tight" calculation. Default is to return an empty bounding box at 0, 0.

Be careful when using this function, the results will not update if the artist window extent of the artist changes. The extent can change due to any changes in the transform stack, such as changing the axes limits, the figure size, or the canvas used (as is done when saving a figure). This can lead to unexpected behavior where interactive figures will look fine on the screen, but will save incorrectly.

`get_zorder`(*self*)
Return the artist's zorder.

`have_units`(*self*)
Return *True* if units are set on any axis.

`is_transform_set`(*self*)
Return whether the Artist has an explicitly set transform.

This is *True* after *set_transform* has been called.

property mouseover
: If this property is set to *True*, the artist will be queried for custom context information when the mouse cursor moves over it.

  See also *get_cursor_data()*, *ToolCursorPosition* and *NavigationToolbar2*.

pchanged(*self*)
: Call all of the registered callbacks.

  This function is triggered internally when a property is changed.

  See also:

  add_callback

  remove_callback

pick(*self, mouseevent*)
: Process a pick event.

  Each child artist will fire a pick event if *mouseevent* is over the artist and the artist has picker set.

  See also:

  set_picker, *get_picker*, *pickable*

pickable(*self*)
: Return whether the artist is pickable.

  See also:

  set_picker, *get_picker*, *pick*

properties(*self*)
: Return a dictionary of all the properties of the artist.

remove(*self*)
: Remove the artist from the figure if possible.

  The effect will not be visible until the figure is redrawn, e.g., with *FigureCanvasBase.draw_idle*. Call *relim* to update the axes limits if desired.

  Note: *relim* will not see collections even if the collection was added to the axes with *autolim* = True.

  Note: there is no support for removing the artist's legend entry.

remove_callback(*self, oid*)
: Remove a callback based on its observer id.

  See also:

add_callback

set(*self*, ***kwargs*)
A property batch setter. Pass *kwargs* to set properties.

set_aa(*self*, *aa*)
Alias for *set_antialiased*.

set_agg_filter(*self*, *filter_func*)
Set the agg filter.

Parameters

filter_func

[callable] A filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array.

set_alpha(*self*, *alpha*)
Set the alpha value used for blending - not supported on all backends.

Parameters

alpha

[float or None]

set_animated(*self*, *b*)
Set the artist's animation state.

Parameters

b

[bool]

set_antialiased(*self*, *aa*)
Set the antialiasing state for rendering.

Parameters

aa

[bool or list of bools]

set_antialiaseds(*self*, *aa*)
Alias for *set_antialiased*.

set_array(*self*, *A*)
Set the image array from numpy array *A*.

Parameters

A

[ndarray]

set_capstyle(*self*, *cs*)
Set the capstyle for the collection (for all its elements).

Parameters

cs

[{'butt', 'round', 'projecting'}] The capstyle.

set_clim(*self*, *vmin=None*, *vmax=None*)
Set the norm limits for image scaling.

Parameters

vmin, vmax

[float] The limits.

The limits may also be passed as a tuple (*vmin*, *vmax*) as a single positional argument.

set_clip_box(*self*, *clipbox*)
Set the artist's clip *Bbox*.

Parameters

clipbox

[*Bbox*]

set_clip_on(*self*, *b*)
Set whether the artist uses clipping.

When False artists will be visible outside of the axes which can lead to unexpected results.

Parameters

b

[bool]

set_clip_path(*self*, *path*, *transform=None*)
Set the artist's clip path.

Parameters

path

[*Patch* or *Path* or *TransformedPath* or None] The clip path. If given a *Path*, *transform* must be provided as well. If *None*, a previously set clip path is removed.

transform

[*Transform*, optional] Only used if *path* is a *Path*, in which case the given *Path* is converted to a *TransformedPath* using *transform*.

Notes

For efficiency, if *path* is a *Rectangle* this method will set the clipping box to the corresponding rectangle and set the clipping path to `None`.

For technical reasons (support of *set*), a tuple (*path*, *transform*) is also accepted as a single positional parameter.

`set_cmap`(*self*, *cmap*)
Set the colormap for luminance data.

Parameters

cmap

[*Colormap* or str or None]

`set_color`(*self*, *c*)
Set both the edgecolor and the facecolor.

Parameters

c

[color or list of rgba tuples]

See also:

Collection.set_facecolor, *Collection.set_edgecolor*

For setting the edge or face color individually.

`set_contains`(*self*, *picker*)
[*Deprecated*] Define a custom contains test for the artist.

The provided callable replaces the default *contains* method of the artist.

Parameters

picker

[callable] A custom picker function to evaluate if an event is within the artist. The function must have the signature:

```
def contains(artist: Artist, event: MouseEvent) -> bool, dict
```

that returns:

- a bool indicating if the event is within the artist
- a dict of additional information. The dict should at least return the same information as the default `contains()` implementation of the respective artist, but may provide additional information.

Notes

Deprecated since version 3.3.

`set_dashes`(*self*, *ls*)
Alias for *set_linestyle*.

`set_ec`(*self*, *c*)
Alias for *set_edgecolor*.

`set_edgecolor`(*self*, *c*)
Set the edgecolor(s) of the collection.

Parameters

c

[color or list of colors or 'face'] The collection edgecolor(s). If a sequence, the patches cycle through it. If 'face', match the facecolor.

`set_edgecolors`(*self*, *c*)
Alias for *set_edgecolor*.

`set_facecolor`(*self*, *c*)
Set the facecolor(s) of the collection. *c* can be a color (all patches have same color), or a sequence of colors; if it is a sequence the patches will cycle through the sequence.

If *c* is 'none', the patch will not be filled.

Parameters

c

[color or list of colors]

`set_facecolors`(*self*, *c*)
Alias for *set_facecolor*.

set_fc(*self*, *c*)
Alias for *set_facecolor*.

set_figure(*self*, *fig*)
Set the *Figure* instance the artist belongs to.

Parameters

fig

[*Figure*]

set_gid(*self*, *gid*)
Set the (group) id for the artist.

Parameters

gid

[str]

set_hatch(*self*, *hatch*)
Set the hatching pattern

hatch can be one of:

```
/   - diagonal hatching
\   - back diagonal
|   - vertical
-   - horizontal
+   - crossed
x   - crossed diagonal
o   - small circle
O   - large circle
.   - dots
*   - stars
```

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Unlike other properties such as linewidth and colors, hatching can only be specified for the collection as a whole, not separately for each member.

Parameters

hatch

[{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}]

set_in_layout(*self*, *in_layout*)
Set if artist is to be included in layout calculations, E.g. *Constrained Layout Guide*, *Figure.tight_layout()*, and `fig.savefig(fname, bbox_inches='tight')`.

Parameters

in_layout

[bool]

`set_joinstyle`(*self*, *js*)
Set the joinstyle for the collection (for all its elements).

Parameters

js

[{'miter', 'round', 'bevel'}] The joinstyle.

`set_label`(*self*, *s*)
Set a label that will be displayed in the legend.

Parameters

s

[object] *s* will be converted to a string by calling `str`.

`set_linestyle`(*self*, *ls*)
Set the linestyle(s) for the collection.

linestyle	description
`'-'` or `'solid'`	solid line
`'--'` or `'dashed'`	dashed line
`'-.'` or `'dashdot'`	dash-dotted line
`':'` or `'dotted'`	dotted line

Alternatively a dash tuple of the following form can be provided:

```
(offset, onoffseq),
```

where `onoffseq` is an even length tuple of on and off ink in points.

Parameters

ls

[str or tuple or list thereof] Valid values for individual linestyles include {'-', '--', '-.', ':', '', (offset, on-off-seq)}. See *Line2D.set_linestyle* for a complete description.

`set_linestyles`(*self*, *ls*)
Alias for *set_linestyle*.

`set_linewidth`(*self*, *lw*)
Set the linewidth(s) for the collection. *lw* can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence

Parameters

lw

[float or list of floats]

`set_linewidths`(*self*, *lw*)
Alias for *set_linewidth*.

`set_ls`(*self*, *ls*)
Alias for *set_linestyle*.

`set_lw`(*self*, *lw*)
Alias for *set_linewidth*.

`set_norm`(*self*, *norm*)
Set the normalization instance.

Parameters

norm

[*Normalize* or None]

Notes

If there are any colorbars using the mappable for this norm, setting the norm of the mappable will reset the norm, locator, and formatters on the colorbar to default.

`set_offset_position`(*self*, *offset_position*)
[*Deprecated*] Set how offsets are applied. If *offset_position* is 'screen' (default) the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is 'data', the offset is applied before the master transform, i.e., the offsets are in data coordinates.

Parameters

offset_position

[{'screen', 'data'}]

Notes

Deprecated since version 3.3.

`set_offsets`(*self*, *offsets*)
Set the offsets for the collection.

Parameters

offsets

[array-like (N, 2) or (2,)]

`set_path_effects`(*self*, *path_effects*)
Set the path effects.

Parameters

path_effects

[*AbstractPathEffect*]

`set_paths`(*self*)

`set_picker`(*self*, *picker*)
Define the picking behavior of the artist.

Parameters

picker

[None or bool or callable] This can be one of the following:

- *None*: Picking is disabled for this artist (default).
- A boolean: If *True* then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist.
- A function: If picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

to determine the hit test. if the mouse event is over the artist, return *hit=True* and props is a dictionary of properties you want added to the PickEvent attributes.

- *deprecated*: For *Line2D* only, *picker* can also be a float that sets the tolerance for checking whether an event occurred "on" the line; this is deprecated. Use *Line2D.set_pickradius* instead.

`set_pickradius`(*self*, *pr*)
Set the pick radius used for containment tests.

Parameters

d

[float] Pick radius, in points.

`set_rasterized`(*self*, *rasterized*)
Force rasterized (bitmap) drawing in vector backend output.

Defaults to None, which implies the backend's default behavior.

Parameters

rasterized

[bool or None]

`set_sizes`(*self*, *sizes*, *dpi=72.0*)
Set the sizes of each member of the collection.

Parameters

sizes

[ndarray or None] The size to set for each element of the collection. The value is the 'area' of the element.

dpi

[float, default: 72] The dpi of the canvas.

`set_sketch_params`(*self*, *scale=None*, *length=None*, *randomness=None*)
Sets the sketch parameters.

Parameters

scale

[float, optional] The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is `None`, or not provided, no sketch filter will be provided.

length

[float, optional] The length of the wiggle along the line, in pixels (default 128.0)

randomness

[float, optional] The scale factor by which the length is shrunken or expanded (default 16.0)

`set_snap`(*self*, *snap*)
Set the snapping behavior.

Snapping aligns positions with the pixel grid, which results in clearer images. For example, if a black line of 1px width was defined at a position in between two pixels, the resulting image would contain the interpolated value of that line in the pixel grid, which would be a grey value on both adjacent pixel positions. In contrast, snapping will move the line to the nearest integer pixel value, so that the resulting image will really contain a 1px wide black line.

Snapping is currently only supported by the Agg and MacOSX backends.

Parameters

snap

[bool or None] Possible values:

- *True*: Snap vertices to the nearest pixel center.
- *False*: Do not modify vertex positions.
- *None*: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center.

`set_transform`(*self*, *t*)
Set the artist transform.

Parameters

t

[*Transform*]

`set_url`(*self*, *url*)
Set the url for the artist.

Parameters

url

[str]

`set_urls`(*self*, *urls*)

Parameters

urls

[list of str or None]

Notes

URLs are currently only implemented by the SVG backend. They are ignored by all other backends.

`set_visible`(*self*, *b*)
Set the artist's visibility.

Parameters

b

[bool]

`set_zorder`(*self*, *level*)
Set the zorder for the artist. Artists with lower zorder values are drawn first.

Parameters

level

[float]

`property stale`
Whether the artist is 'stale' and needs to be re-drawn for the output to match the internal state of the artist.

`property sticky_edges`
`x` and `y` sticky edge lists for autoscaling.

When performing autoscaling, if a data limit coincides with a value in the corresponding sticky_edges list, then no margin will be added--the view limit "sticks" to the edge. A typical use case is histograms, where one usually expects no margin on the bottom edge (0) of the histogram.

This attribute cannot be assigned to; however, the `x` and `y` lists can be modified in place as needed.

Examples

```
>>> artist.sticky_edges.x[:] = (xmin, xmax)
>>> artist.sticky_edges.y[:] = (ymin, ymax)
```

`to_rgba`(*self*, *x*, *alpha=None*, *bytes=False*, *norm=True*)
Return a normalized rgba array corresponding to *x*.

In the normal case, *x* is a 1-D or 2-D sequence of scalars, and the corresponding ndarray of rgba values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already rgb or rgba, such as might have been read from an image file. If *x* is an ndarray with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an rgb or rgba array, and no mapping will be done. The array can be uint8, or it can be floating point with values in the 0-1 range; otherwise a ValueError will be raised. If it is a masked array, the mask will be ignored. If the last dimension is 3, the *alpha* kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the *alpha* kwarg is ignored; it does not replace the pre-existing alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if *bytes* is *False* (default), the rgba array will be floats in the 0-1 range; if it is *True*, the returned rgba array will be uint8 in the 0 to 255 range.

If norm is False, no normalization of the input data is performed, and it is assumed to be in the range (0-1).

`update`(*self*, *props*)
Update this artist's properties from the dict *props*.

Parameters

props

[dict]

`property update_dict`

`update_from`(*self*, *other*)
Copy properties from other to self.

`update_scalarmappable`(*self*)
Update colors from the scalar mappable array, if it is not None.

`zorder = 0`

`class matplotlib.collections.StarPolygonCollection`(*numsides*, *rotation=0*, *sizes=1*, ***kwargs*)
Bases: *matplotlib.collections.RegularPolyCollection*

Draw a collection of regular stars with *numsides* points.

Parameters

numsides

[int] The number of sides of the polygon.

rotation

[float] The rotation of the polygon in radians.

sizes

[tuple of float] The area of the circle circumscribing the polygon in points^2.

****kwargs**

Forwarded to *Collection*.

Examples

See /gallery/event_handling/lasso_demo for a complete example:

```
offsets = np.random.rand(20, 2)
facecolors = [cm.jet(x) for x in np.random.rand(20)]

collection = RegularPolyCollection(
    numsides=5, # a pentagon
    rotation=0, sizes=(50,),
    facecolors=facecolors,
    edgecolors=("black",),
    linewidths=(1,),
    offsets=offsets,
    transOffset=ax.transData,
    )
```

`add_callback`(*self*, *func*)

Add a callback function that will be called whenever one of the *Artist*'s properties changes.

Parameters

func

[callable] The callback function. It must have the signature:

```
def func(artist: Artist) -> Any
```

where *artist* is the calling *Artist*. Return values may exist but are ignored.

Returns

int

The observer id associated with the callback. This id can be used for removing the callback with *remove_callback* later.

See also:

remove_callback

`add_checker`(*self*, *checker*)
[*Deprecated*]

Notes

Deprecated since version 3.3:

`autoscale`(*self*)
Autoscale the scalar limits on the norm instance using the current array

`autoscale_None`(*self*)
Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

`property axes`
The *Axes* instance the artist resides in, or *None*.

`changed`(*self*)
Call this whenever the mappable is changed to notify all the callbackSM listeners to the 'changed' signal.

`check_update`(*self*, *checker*)
[*Deprecated*]

Notes

Deprecated since version 3.3:

`contains`(*self*, *mouseevent*)
Test whether the mouse event occurred in the collection.

Returns `bool, dict(ind=itemlist)`, where every item in itemlist contains the event.

`convert_xunits`(*self*, *x*)
Convert *x* using the unit type of the xaxis.

If the artist is not in contained in an Axes or if the xaxis does not have units, *x* itself is returned.

`convert_yunits`(*self*, *y*)
Convert *y* using the unit type of the yaxis.

If the artist is not in contained in an Axes or if the yaxis does not have units, *y* itself is returned.

`draw`(*self*, *renderer*)
Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

Parameters

renderer

[*RendererBase* subclass.]

Notes

This method is overridden in the Artist subclasses.

`findobj`(*self*, *match=None*, *include_self=True*)
Find artist objects.

Recursively find all *Artist* instances contained in the artist.

Parameters

match

A filter criterion for the matches. This can be

- *None*: Return all objects contained in artist.
- A function with signature `def match(artist: Artist) -> bool`. The result will only contain artists for which the function returns *True*.
- A class instance: e.g., *Line2D*. The result will only contain artists of this class or its subclasses (`isinstance` check).

include_self

[bool] Include *self* in the list to be checked for a match.

Returns

list of *Artist*

`format_cursor_data`(*self*, *data*)
Return a string representation of *data*.

Note: This method is intended to be overridden by artist subclasses. As an end-user of Matplotlib you will most likely not call this method yourself.

The default implementation converts ints and floats and arrays of ints and floats into a comma-separated string enclosed in square brackets.

See also:

get_cursor_data

`get_agg_filter`(*self*)
Return filter function to be used for agg filter.

`get_alpha`(*self*)
Return the alpha value used for blending - not supported on all backends.

`get_animated`(*self*)
Return whether the artist is animated.

`get_array`(*self*)
Return the data array.

`get_capstyle`(*self*)

`get_children`(*self*)
Return a list of the child *Artist*s of this *Artist*.

`get_clim`(*self*)
Return the values (min, max) that are mapped to the colormap limits.

`get_clip_box`(*self*)
Return the clipbox.

`get_clip_on`(*self*)
Return whether the artist uses clipping.

`get_clip_path`(*self*)
Return the clip path.

`get_cmap`(*self*)
Return the *Colormap* instance.

`get_contains`(*self*)
[*Deprecated*] Return the custom contains function of the artist if set, or *None*.

See also:

set_contains

Notes

Deprecated since version 3.3.

`get_cursor_data`(*self*, *event*)
Return the cursor data for a given event.

Note: This method is intended to be overridden by artist subclasses. As an end-user of Matplotlib you will most likely not call this method yourself.

Cursor data can be used by Artists to provide additional context information for a given event. The default implementation just returns *None*.

Subclasses can override the method and return arbitrary data. However, when doing so, they must ensure that *format_cursor_data* can convert the data to a string representation.

The only current use case is displaying the z-value of an *AxesImage* in the status bar of a plot window, while moving the mouse.

Parameters

event

[*matplotlib.backend_bases.MouseEvent*]

See also:

format_cursor_data

`get_dashes`(*self*)
Alias for *get_linestyle*.

`get_datalim`(*self*, *transData*)

`get_ec`(*self*)
Alias for *get_edgecolor*.

`get_edgecolor`(*self*)

`get_edgecolors`(*self*)
Alias for *get_edgecolor*.

`get_facecolor`(*self*)

`get_facecolors`(*self*)
Alias for *get_facecolor*.

`get_fc`(*self*)
Alias for *get_facecolor*.

`get_figure`(*self*)
Return the *Figure* instance the artist belongs to.

`get_fill`(*self*)
Return whether fill is set.

`get_gid`(*self*)
Return the group id.

`get_hatch`(*self*)
Return the current hatching pattern.

`get_in_layout`(*self*)
Return boolean flag, `True` if artist is included in layout calculations.

E.g. *Constrained Layout Guide*, *Figure.tight_layout()*, and `fig.savefig(fname, bbox_inches='tight')`.

get_joinstyle(*self*)

get_label(*self*)
Return the label used for this artist in the legend.

get_linestyle(*self*)

get_linestyles(*self*)
Alias for *get_linestyle*.

get_linewidth(*self*)

get_linewidths(*self*)
Alias for *get_linewidth*.

get_ls(*self*)
Alias for *get_linestyle*.

get_lw(*self*)
Alias for *get_linewidth*.

get_numsides(*self*)

get_offset_position(*self*)
[*Deprecated*] Return how offsets are applied for the collection. If *offset_position* is 'screen', the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is 'data', the offset is applied before the master transform, i.e., the offsets are in data coordinates.

Notes

Deprecated since version 3.3.

get_offset_transform(*self*)

get_offsets(*self*)
Return the offsets for the collection.

get_path_effects(*self*)

get_paths(*self*)

get_picker(*self*)
Return the picking behavior of the artist.

The possible values are described in *set_picker*.

See also:

set_picker, *pickable*, *pick*

get_pickradius(*self*)

get_rasterized(*self*)
Return whether the artist is to be rasterized.

get_rotation(*self*)

get_sizes(*self*)
Return the sizes ('areas') of the elements in the collection.

Returns

array

The 'area' of each element.

get_sketch_params(*self*)
Return the sketch parameters for the artist.

Returns

tuple or None

A 3-tuple with the following elements:

- *scale*: The amplitude of the wiggle perpendicular to the source line.
- *length*: The length of the wiggle along the line.
- *randomness*: The scale factor by which the length is shrunken or expanded.

Returns *None* if no sketch parameters were set.

get_snap(*self*)
Return the snap setting.

See *set_snap* for details.

get_tightbbox(*self*, *renderer*)
Like *Artist.get_window_extent*, but includes any clipping.

Parameters

renderer

[*RendererBase* subclass] renderer that will be used to draw the figures (i.e. `fig.canvas.get_renderer()`)

Returns

Bbox

The enclosing bounding box (in figure pixel coordinates).

`get_transform`(*self*)
Return the *Transform* instance used by this artist.

`get_transformed_clip_path_and_affine`(*self*)
Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

`get_transforms`(*self*)

`get_url`(*self*)
Return the url.

`get_urls`(*self*)
Return a list of URLs, one for each element of the collection.

The list contains *None* for elements without a URL. See /gallery/misc/hyperlinks_sgskip for an example.

`get_visible`(*self*)
Return the visibility.

`get_window_extent`(*self*, *renderer*)
Get the axes bounding box in display space.

The bounding box' width and height are nonnegative.

Subclasses should override for inclusion in the bounding box "tight" calculation. Default is to return an empty bounding box at 0, 0.

Be careful when using this function, the results will not update if the artist window extent of the artist changes. The extent can change due to any changes in the transform stack, such as changing the axes limits, the figure size, or the canvas used (as is done when saving a figure). This can lead to unexpected behavior where interactive figures will look fine on the screen, but will save incorrectly.

`get_zorder`(*self*)
Return the artist's zorder.

`have_units`(*self*)
Return *True* if units are set on any axis.

`is_transform_set`(*self*)
Return whether the Artist has an explicitly set transform.

This is *True* after *set_transform* has been called.

`property mouseover`
If this property is set to *True*, the artist will be queried for custom context information when the mouse cursor moves over it.

See also *get_cursor_data()*, *ToolCursorPosition* and *NavigationToolbar2*.

`pchanged`(*self*)
Call all of the registered callbacks.

This function is triggered internally when a property is changed.

See also:

add_callback

remove_callback

pick(*self*, *mouseevent*)
Process a pick event.

Each child artist will fire a pick event if *mouseevent* is over the artist and the artist has picker set.

See also:

set_picker, *get_picker*, *pickable*

pickable(*self*)
Return whether the artist is pickable.

See also:

set_picker, *get_picker*, *pick*

properties(*self*)
Return a dictionary of all the properties of the artist.

remove(*self*)
Remove the artist from the figure if possible.

The effect will not be visible until the figure is redrawn, e.g., with *FigureCanvasBase.draw_idle*. Call *relim* to update the axes limits if desired.

Note: *relim* will not see collections even if the collection was added to the axes with *autolim* = True.

Note: there is no support for removing the artist's legend entry.

remove_callback(*self*, *oid*)
Remove a callback based on its observer id.

See also:

add_callback

set(*self*, ***kwargs*)
A property batch setter. Pass *kwargs* to set properties.

set_aa(*self*, *aa*)
Alias for *set_antialiased*.

set_agg_filter(*self*, *filter_func*)
Set the agg filter.

Parameters

filter_func

[callable] A filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array.

`set_alpha`(*self*, *alpha*)
Set the alpha value used for blending - not supported on all backends.

Parameters

alpha

[float or None]

`set_animated`(*self*, *b*)
Set the artist's animation state.

Parameters

b

[bool]

`set_antialiased`(*self*, *aa*)
Set the antialiasing state for rendering.

Parameters

aa

[bool or list of bools]

`set_antialiaseds`(*self*, *aa*)
Alias for *set_antialiased*.

`set_array`(*self*, *A*)
Set the image array from numpy array *A*.

Parameters

A

[ndarray]

`set_capstyle`(*self*, *cs*)
Set the capstyle for the collection (for all its elements).

Parameters

cs

[{'butt', 'round', 'projecting'}] The capstyle.

`set_clim`(*self*, *vmin=None*, *vmax=None*)
Set the norm limits for image scaling.

Parameters

vmin, vmax

[float] The limits.

The limits may also be passed as a tuple (*vmin*, *vmax*) as a single positional argument.

`set_clip_box`(*self*, *clipbox*)
Set the artist's clip *Bbox*.

Parameters

clipbox

[*Bbox*]

`set_clip_on`(*self*, *b*)
Set whether the artist uses clipping.

When False artists will be visible outside of the axes which can lead to unexpected results.

Parameters

b

[bool]

`set_clip_path`(*self*, *path*, *transform=None*)
Set the artist's clip path.

Parameters

path

[*Patch* or *Path* or *TransformedPath* or None] The clip path. If given a *Path*, *transform* must be provided as well. If *None*, a previously set clip path is removed.

transform

[*Transform*, optional] Only used if *path* is a *Path*, in which case the given *Path* is converted to a *TransformedPath* using *transform*.

Notes

For efficiency, if *path* is a *Rectangle* this method will set the clipping box to the corresponding rectangle and set the clipping path to `None`.

For technical reasons (support of *set*), a tuple (*path*, *transform*) is also accepted as a single positional parameter.

`set_cmap`(*self*, *cmap*)
Set the colormap for luminance data.

Parameters

cmap

[*Colormap* or str or None]

`set_color`(*self*, *c*)
Set both the edgecolor and the facecolor.

Parameters

c

[color or list of rgba tuples]

See also:

Collection.set_facecolor, *Collection.set_edgecolor*

For setting the edge or face color individually.

`set_contains`(*self*, *picker*)
[*Deprecated*] Define a custom contains test for the artist.

The provided callable replaces the default *contains* method of the artist.

Parameters

picker

[callable] A custom picker function to evaluate if an event is within the artist. The function must have the signature:

```
def contains(artist: Artist, event: MouseEvent) -> bool, dict
```

that returns:

- a bool indicating if the event is within the artist
- a dict of additional information. The dict should at least return the same information as the default `contains()` implementation of the respective artist, but may provide additional information.

Notes

Deprecated since version 3.3.

`set_dashes`(*self*, *ls*)
Alias for *set_linestyle*.

`set_ec`(*self*, *c*)
Alias for *set_edgecolor*.

`set_edgecolor`(*self*, *c*)
Set the edgecolor(s) of the collection.

Parameters

c

[color or list of colors or 'face'] The collection edgecolor(s). If a sequence, the patches cycle through it. If 'face', match the facecolor.

`set_edgecolors`(*self*, *c*)
Alias for *set_edgecolor*.

`set_facecolor`(*self*, *c*)
Set the facecolor(s) of the collection. *c* can be a color (all patches have same color), or a sequence of colors; if it is a sequence the patches will cycle through the sequence.

If *c* is 'none', the patch will not be filled.

Parameters

c

[color or list of colors]

`set_facecolors`(*self*, *c*)
Alias for *set_facecolor*.

`set_fc`(*self*, *c*)
Alias for *set_facecolor*.

`set_figure`(*self*, *fig*)
Set the *Figure* instance the artist belongs to.

Parameters

fig

[*Figure*]

`set_gid`(*self*, *gid*)
Set the (group) id for the artist.

Parameters

gid

[str]

set_hatch(*self*, *hatch*)

Set the hatching pattern

hatch can be one of:

```
/   - diagonal hatching
\   - back diagonal
|   - vertical
-   - horizontal
+   - crossed
x   - crossed diagonal
o   - small circle
O   - large circle
.   - dots
*   - stars
```

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Unlike other properties such as linewidth and colors, hatching can only be specified for the collection as a whole, not separately for each member.

Parameters

hatch

[{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}]

set_in_layout(*self*, *in_layout*)

Set if artist is to be included in layout calculations, E.g. *Constrained Layout Guide*, *`Figure.tight_layout()`*, and `fig.savefig(fname, bbox_inches='tight')`.

Parameters

in_layout

[bool]

set_joinstyle(*self*, *js*)

Set the joinstyle for the collection (for all its elements).

Parameters

js

[{'miter', 'round', 'bevel'}] The joinstyle.

`set_label`(*self*, *s*)
Set a label that will be displayed in the legend.

Parameters

s

[object] *s* will be converted to a string by calling `str`.

`set_linestyle`(*self*, *ls*)
Set the linestyle(s) for the collection.

linestyle	description
`'-'` or `'solid'`	solid line
`'--'` or `'dashed'`	dashed line
`'-.'` or `'dashdot'`	dash-dotted line
`':'` or `'dotted'`	dotted line

Alternatively a dash tuple of the following form can be provided:

```
(offset, onoffseq),
```

where `onoffseq` is an even length tuple of on and off ink in points.

Parameters

ls

[str or tuple or list thereof] Valid values for individual linestyles include {'-', '--', '-.', ':', '', (offset, on-off-seq)}. See *Line2D.set_linestyle* for a complete description.

`set_linestyles`(*self*, *ls*)
Alias for *set_linestyle*.

`set_linewidth`(*self*, *lw*)
Set the linewidth(s) for the collection. *lw* can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence

Parameters

lw

[float or list of floats]

`set_linewidths`(*self*, *lw*)
Alias for *set_linewidth*.

`set_ls`(*self*, *ls*)
Alias for *set_linestyle*.

set_lw(*self*, *lw*)
Alias for *set_linewidth*.

set_norm(*self*, *norm*)
Set the normalization instance.

Parameters

norm

[*Normalize* or None]

Notes

If there are any colorbars using the mappable for this norm, setting the norm of the mappable will reset the norm, locator, and formatters on the colorbar to default.

set_offset_position(*self*, *offset_position*)
[*Deprecated*] Set how offsets are applied. If *offset_position* is 'screen' (default) the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is 'data', the offset is applied before the master transform, i.e., the offsets are in data coordinates.

Parameters

offset_position

[{'screen', 'data'}]

Notes

Deprecated since version 3.3.

set_offsets(*self*, *offsets*)
Set the offsets for the collection.

Parameters

offsets

[array-like (N, 2) or (2,)]

set_path_effects(*self*, *path_effects*)
Set the path effects.

Parameters

path_effects

[*AbstractPathEffect*]

`set_paths`(*self*)

`set_picker`(*self*, *picker*)

Define the picking behavior of the artist.

Parameters

picker

[None or bool or callable] This can be one of the following:

- *None*: Picking is disabled for this artist (default).
- A boolean: If *True* then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist.
- A function: If picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

to determine the hit test. if the mouse event is over the artist, return *hit=True* and props is a dictionary of properties you want added to the PickEvent attributes.

- *deprecated*: For *Line2D* only, *picker* can also be a float that sets the tolerance for checking whether an event occurred "on" the line; this is deprecated. Use *Line2D.set_pickradius* instead.

`set_pickradius`(*self*, *pr*)

Set the pick radius used for containment tests.

Parameters

d

[float] Pick radius, in points.

`set_rasterized`(*self*, *rasterized*)

Force rasterized (bitmap) drawing in vector backend output.

Defaults to None, which implies the backend's default behavior.

Parameters

rasterized

[bool or None]

set_sizes(*self*, *sizes*, *dpi=72.0*)

Set the sizes of each member of the collection.

Parameters

sizes

[ndarray or None] The size to set for each element of the collection. The value is the 'area' of the element.

dpi

[float, default: 72] The dpi of the canvas.

set_sketch_params(*self*, *scale=None*, *length=None*, *randomness=None*)

Sets the sketch parameters.

Parameters

scale

[float, optional] The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is `None`, or not provided, no sketch filter will be provided.

length

[float, optional] The length of the wiggle along the line, in pixels (default 128.0)

randomness

[float, optional] The scale factor by which the length is shrunken or expanded (default 16.0)

set_snap(*self*, *snap*)

Set the snapping behavior.

Snapping aligns positions with the pixel grid, which results in clearer images. For example, if a black line of 1px width was defined at a position in between two pixels, the resulting image would contain the interpolated value of that line in the pixel grid, which would be a grey value on both adjacent pixel positions. In contrast, snapping will move the line to the nearest integer pixel value, so that the resulting image will really contain a 1px wide black line.

Snapping is currently only supported by the Agg and MacOSX backends.

Parameters

snap

[bool or None] Possible values:

- *True*: Snap vertices to the nearest pixel center.

- *False*: Do not modify vertex positions.
- *None*: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center.

`set_transform`(*self*, *t*)
Set the artist transform.

Parameters

t

[*Transform*]

`set_url`(*self*, *url*)
Set the url for the artist.

Parameters

url

[str]

`set_urls`(*self*, *urls*)

Parameters

urls

[list of str or None]

Notes

URLs are currently only implemented by the SVG backend. They are ignored by all other backends.

`set_visible`(*self*, *b*)
Set the artist's visibility.

Parameters

b

[bool]

`set_zorder`(*self*, *level*)
Set the zorder for the artist. Artists with lower zorder values are drawn first.

Parameters

level

[float]

property stale
: Whether the artist is 'stale' and needs to be re-drawn for the output to match the internal state of the artist.

property sticky_edges
: `x` and `y` sticky edge lists for autoscaling.

When performing autoscaling, if a data limit coincides with a value in the corresponding sticky_edges list, then no margin will be added--the view limit "sticks" to the edge. A typical use case is histograms, where one usually expects no margin on the bottom edge (0) of the histogram.

This attribute cannot be assigned to; however, the `x` and `y` lists can be modified in place as needed.

Examples

```
>>> artist.sticky_edges.x[:] = (xmin, xmax)
>>> artist.sticky_edges.y[:] = (ymin, ymax)
```

to_rgba(*self*, *x*, *alpha=None*, *bytes=False*, *norm=True*)
: Return a normalized rgba array corresponding to *x*.

In the normal case, *x* is a 1-D or 2-D sequence of scalars, and the corresponding ndarray of rgba values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already rgb or rgba, such as might have been read from an image file. If *x* is an ndarray with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an rgb or rgba array, and no mapping will be done. The array can be uint8, or it can be floating point with values in the 0-1 range; otherwise a ValueError will be raised. If it is a masked array, the mask will be ignored. If the last dimension is 3, the *alpha* kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the *alpha* kwarg is ignored; it does not replace the pre-existing alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if *bytes* is *False* (default), the rgba array will be floats in the 0-1 range; if it is *True*, the returned rgba array will be uint8 in the 0 to 255 range.

If norm is False, no normalization of the input data is performed, and it is assumed to be in the range (0-1).

update(*self*, *props*)
: Update this artist's properties from the dict *props*.

Parameters

props

[dict]

property update_dict

update_from(*self*, *other*)
Copy properties from other to self.

update_scalarmappable(*self*)
Update colors from the scalar mappable array, if it is not None.

zorder = 0

class matplotlib.collections.TriMesh(*triangulation*, ***kwargs*)
Bases: *matplotlib.collections.Collection*

Class for the efficient drawing of a triangular mesh using Gouraud shading.

A triangular mesh is a *Triangulation* object.

Parameters

edgecolors

[color or list of colors, default: rcParams["patch.edgecolor"] (default: 'black')] Edge color for each patch making up the collection. The special value 'face' can be passed to make the edgecolor match the facecolor.

facecolors

[color or list of colors, default: rcParams["patch.facecolor"] (default: 'C0')] Face color for each patch making up the collection.

linewidths

[float or list of floats, default: rcParams["patch.linewidth"] (default: 1.0)] Line width for each patch making up the collection.

linestyles

[str or tuple or list thereof, default: 'solid'] Valid strings are ['solid', 'dashed', 'dashdot', 'dotted', '-', '--', '-.', ':']. Dash tuples should be of the form:

```
(offset, onoffseq),
```

where *onoffseq* is an even length tuple of on and off ink lengths in points. For examples, see /gallery/lines_bars_and_markers/linestyles.

capstyle

[str, default: rcParams["patch.capstyle"]] Style to use for capping lines for all paths in the collection. See /gallery/lines_bars_and_markers/joinstyle for a demonstration of each of the allowed values.

joinstyle

[str, default: rcParams["patch.joinstyle"]] Style to use for joining lines for all paths in the collection. See /gallery/lines_bars_and_markers/joinstyle for a demonstration of each of the allowed values.

antialiaseds

[bool or list of bool, default: rcParams["patch.antialiased"] (default: True)] Whether each pach in the collection should be drawn with antialiasing.

offsets

[(float, float) or list thereof, default: (0, 0)] A vector by which to translate each patch after rendering (default is no translation). The translation is performed in screen (pixel) coordinates (i.e. after the Artist's transform is applied).

transOffset

[*Transform*, default: *IdentityTransform*] A single transform which will be applied to each *offsets* vector before it is used.

offset_position

[{'screen' (default), 'data' (deprecated)}] If set to 'data' (deprecated), *offsets* will be treated as if it is in data coordinates instead of in screen coordinates.

norm

[*Normalize*, optional] Forwarded to *ScalarMappable*. The default of None means that the first draw call will set vmin and vmax using the minimum and maximum values of the data.

cmap

[*Colormap*, optional] Forwarded to *ScalarMappable*. The default of None will result in rcParams["image.cmap"] (default: 'viridis') being used.

hatch

[str, optional] Hatching pattern to use in filled paths, if any. Valid strings are ['/', '', '|', '-', '+', 'x', 'o', 'O', '.', '*']. See /gallery/shapes_and_collections/hatch_demo for the meaning of each hatch type.

pickradius

[float, default: 5.0] If `pickradius <= 0`, then *Collection.contains* will return `True` whenever the test point is inside of one of the polygons formed by the control points of a Path in the Collection. On the other hand, if it is greater than 0, then we instead check if the test point is contained in a stroke of width `2*pickradius` following any of the Paths in the Collection.

urls

[list of str, default: None] A URL for each patch to link to once drawn. Currently only works for the SVG backend. See /gallery/misc/hyperlinks_sgskip for examples.

zorder

[float, default: 1] The drawing order, shared by all Patches in the Collection. See /gallery/misc/zorder_demo for all defaults and examples.

`add_callback`(*self*, *func*)

Add a callback function that will be called whenever one of the *Artist*'s properties changes.

Parameters

func

[callable] The callback function. It must have the signature:

```
def func(artist: Artist) -> Any
```

where *artist* is the calling *Artist*. Return values may exist but are ignored.

Returns

int

The observer id associated with the callback. This id can be used for removing the callback with *remove_callback* later.

See also:

remove_callback

`add_checker`(*self*, *checker*)

[*Deprecated*]

Notes

Deprecated since version 3.3:

`autoscale`(*self*)
Autoscale the scalar limits on the norm instance using the current array

`autoscale_None`(*self*)
Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

`property axes`
The *Axes* instance the artist resides in, or *None*.

`changed`(*self*)
Call this whenever the mappable is changed to notify all the callbackSM listeners to the 'changed' signal.

`check_update`(*self*, *checker*)
[*Deprecated*]

Notes

Deprecated since version 3.3:

`contains`(*self*, *mouseevent*)
Test whether the mouse event occurred in the collection.

Returns `bool, dict(ind=itemlist)`, where every item in itemlist contains the event.

`static convert_mesh_to_paths`(*tri*)
Convert a given mesh into a sequence of *Path* objects.

This function is primarily of use to implementers of backends that do not directly support meshes.

`convert_xunits`(*self*, *x*)
Convert *x* using the unit type of the xaxis.

If the artist is not in contained in an Axes or if the xaxis does not have units, *x* itself is returned.

`convert_yunits`(*self*, *y*)
Convert *y* using the unit type of the yaxis.

If the artist is not in contained in an Axes or if the yaxis does not have units, *y* itself is returned.

`draw`(*self*, *renderer*)
Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

Parameters

renderer

[*RendererBase* subclass.]

Notes

This method is overridden in the Artist subclasses.

`findobj`(*self*, *match=None*, *include_self=True*)
Find artist objects.

Recursively find all *Artist* instances contained in the artist.

Parameters

match

A filter criterion for the matches. This can be

- *None*: Return all objects contained in artist.
- A function with signature `def match(artist: Artist) -> bool`. The result will only contain artists for which the function returns *True*.
- A class instance: e.g., *Line2D*. The result will only contain artists of this class or its subclasses (`isinstance` check).

include_self

[bool] Include *self* in the list to be checked for a match.

Returns

list of *Artist*

`format_cursor_data`(*self*, *data*)
Return a string representation of *data*.

Note: This method is intended to be overridden by artist subclasses. As an end-user of Matplotlib you will most likely not call this method yourself.

The default implementation converts ints and floats and arrays of ints and floats into a comma-separated string enclosed in square brackets.

See also:

get_cursor_data

`get_agg_filter`(*self*)
Return filter function to be used for agg filter.

`get_alpha`(*self*)
Return the alpha value used for blending - not supported on all backends.

`get_animated`(*self*)
Return whether the artist is animated.

`get_array`(*self*)
Return the data array.

`get_capstyle`(*self*)

`get_children`(*self*)
Return a list of the child *Artist*s of this *Artist*.

`get_clim`(*self*)
Return the values (min, max) that are mapped to the colormap limits.

`get_clip_box`(*self*)
Return the clipbox.

`get_clip_on`(*self*)
Return whether the artist uses clipping.

`get_clip_path`(*self*)
Return the clip path.

`get_cmap`(*self*)
Return the *Colormap* instance.

`get_contains`(*self*)
[*Deprecated*] Return the custom contains function of the artist if set, or *None*.

See also:

set_contains

Notes

Deprecated since version 3.3.

`get_cursor_data`(*self*, *event*)
Return the cursor data for a given event.

Note: This method is intended to be overridden by artist subclasses. As an end-user of Matplotlib you will most likely not call this method yourself.

Cursor data can be used by Artists to provide additional context information for a given event. The default implementation just returns *None*.

Subclasses can override the method and return arbitrary data. However, when doing so, they must ensure that *format_cursor_data* can convert the data to a string representation.

The only current use case is displaying the z-value of an *AxesImage* in the status bar of a plot window, while moving the mouse.

Parameters

event

[*matplotlib.backend_bases.MouseEvent*]

See also:

format_cursor_data

`get_dashes`(*self*)
Alias for *get_linestyle*.

`get_datalim`(*self*, *transData*)

`get_ec`(*self*)
Alias for *get_edgecolor*.

`get_edgecolor`(*self*)

`get_edgecolors`(*self*)
Alias for *get_edgecolor*.

`get_facecolor`(*self*)

`get_facecolors`(*self*)
Alias for *get_facecolor*.

`get_fc`(*self*)
Alias for *get_facecolor*.

`get_figure`(*self*)
Return the *Figure* instance the artist belongs to.

`get_fill`(*self*)
Return whether fill is set.

`get_gid`(*self*)
Return the group id.

`get_hatch`(*self*)
Return the current hatching pattern.

`get_in_layout`(*self*)
Return boolean flag, `True` if artist is included in layout calculations.

E.g. *Constrained Layout Guide*, *Figure.tight_layout()*, and `fig.savefig(fname, bbox_inches='tight')`.

`get_joinstyle`(*self*)

`get_label`(*self*)
Return the label used for this artist in the legend.

`get_linestyle`(*self*)

`get_linestyles`(*self*)
Alias for *get_linestyle*.

`get_linewidth`(*self*)

`get_linewidths`(*self*)
Alias for *get_linewidth*.

`get_ls`(*self*)
Alias for *get_linestyle*.

`get_lw`(*self*)
Alias for *get_linewidth*.

`get_offset_position`(*self*)
[*Deprecated*] Return how offsets are applied for the collection. If *offset_position* is 'screen', the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is 'data', the offset is applied before the master transform, i.e., the offsets are in data coordinates.

Notes

Deprecated since version 3.3.

`get_offset_transform`(*self*)

`get_offsets`(*self*)
Return the offsets for the collection.

`get_path_effects`(*self*)

`get_paths`(*self*)

`get_picker`(*self*)
Return the picking behavior of the artist.

The possible values are described in *set_picker*.

See also:

set_picker, *pickable*, *pick*

`get_pickradius`(*self*)

`get_rasterized`(*self*)
Return whether the artist is to be rasterized.

get_sketch_params(*self*)
Return the sketch parameters for the artist.

Returns

tuple or None

A 3-tuple with the following elements:

- *scale*: The amplitude of the wiggle perpendicular to the source line.
- *length*: The length of the wiggle along the line.
- *randomness*: The scale factor by which the length is shrunken or expanded.

Returns *None* if no sketch parameters were set.

get_snap(*self*)
Return the snap setting.

See *set_snap* for details.

get_tightbbox(*self*, *renderer*)
Like *Artist.get_window_extent*, but includes any clipping.

Parameters

renderer

[*RendererBase* subclass] renderer that will be used to draw the figures (i.e. `fig.canvas.get_renderer()`)

Returns

Bbox

The enclosing bounding box (in figure pixel coordinates).

get_transform(*self*)
Return the *Transform* instance used by this artist.

get_transformed_clip_path_and_affine(*self*)
Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

get_transforms(*self*)

get_url(*self*)
Return the url.

get_urls(*self*)
Return a list of URLs, one for each element of the collection.

The list contains *None* for elements without a URL. See /gallery/misc/hyperlinks_sgskip for an example.

`get_visible`(*self*)
Return the visibility.

`get_window_extent`(*self*, *renderer*)
Get the axes bounding box in display space.

The bounding box' width and height are nonnegative.

Subclasses should override for inclusion in the bounding box "tight" calculation. Default is to return an empty bounding box at 0, 0.

Be careful when using this function, the results will not update if the artist window extent of the artist changes. The extent can change due to any changes in the transform stack, such as changing the axes limits, the figure size, or the canvas used (as is done when saving a figure). This can lead to unexpected behavior where interactive figures will look fine on the screen, but will save incorrectly.

`get_zorder`(*self*)
Return the artist's zorder.

`have_units`(*self*)
Return *True* if units are set on any axis.

`is_transform_set`(*self*)
Return whether the Artist has an explicitly set transform.

This is *True* after *set_transform* has been called.

`property mouseover`
If this property is set to *True*, the artist will be queried for custom context information when the mouse cursor moves over it.

See also *get_cursor_data()*, *ToolCursorPosition* and *NavigationToolbar2*.

`pchanged`(*self*)
Call all of the registered callbacks.

This function is triggered internally when a property is changed.

See also:

add_callback

remove_callback

`pick`(*self*, *mouseevent*)
Process a pick event.

Each child artist will fire a pick event if *mouseevent* is over the artist and the artist has picker set.

See also:

set_picker, *get_picker*, *pickable*

pickable(*self*)
Return whether the artist is pickable.

See also:

set_picker, *get_picker*, *pick*

properties(*self*)
Return a dictionary of all the properties of the artist.

remove(*self*)
Remove the artist from the figure if possible.

The effect will not be visible until the figure is redrawn, e.g., with *FigureCanvasBase.draw_idle*. Call *relim* to update the axes limits if desired.

Note: *relim* will not see collections even if the collection was added to the axes with *autolim* = True.

Note: there is no support for removing the artist's legend entry.

remove_callback(*self*, *oid*)
Remove a callback based on its observer id.

See also:

add_callback

set(*self*, ***kwargs*)
A property batch setter. Pass *kwargs* to set properties.

set_aa(*self*, *aa*)
Alias for *set_antialiased*.

set_agg_filter(*self*, *filter_func*)
Set the agg filter.

Parameters

filter_func

[callable] A filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array.

set_alpha(*self*, *alpha*)
Set the alpha value used for blending - not supported on all backends.

Parameters

alpha

[float or None]

set_animated(*self*, *b*)
Set the artist's animation state.

Parameters

b

[bool]

set_antialiased(*self*, *aa*)
Set the antialiasing state for rendering.

Parameters

aa

[bool or list of bools]

set_antialiaseds(*self*, *aa*)
Alias for *set_antialiased*.

set_array(*self*, *A*)
Set the image array from numpy array *A*.

Parameters

A

[ndarray]

set_capstyle(*self*, *cs*)
Set the capstyle for the collection (for all its elements).

Parameters

cs

[{'butt', 'round', 'projecting'}] The capstyle.

set_clim(*self*, *vmin=None*, *vmax=None*)
Set the norm limits for image scaling.

Parameters

vmin, vmax

[float] The limits.

The limits may also be passed as a tuple (*vmin*, *vmax*) as a single positional argument.

set_clip_box(*self*, *clipbox*)
Set the artist's clip *Bbox*.

Parameters

clipbox

[*Bbox*]

set_clip_on(*self*, *b*)
Set whether the artist uses clipping.

When False artists will be visible outside of the axes which can lead to unexpected results.

Parameters

b

[bool]

set_clip_path(*self*, *path*, *transform=None*)
Set the artist's clip path.

Parameters

path

[*Patch* or *Path* or *TransformedPath* or None] The clip path. If given a *Path*, *transform* must be provided as well. If *None*, a previously set clip path is removed.

transform

[*Transform*, optional] Only used if *path* is a *Path*, in which case the given *Path* is converted to a *TransformedPath* using *transform*.

Notes

For efficiency, if *path* is a *Rectangle* this method will set the clipping box to the corresponding rectangle and set the clipping path to `None`.

For technical reasons (support of *set*), a tuple (*path*, *transform*) is also accepted as a single positional parameter.

set_cmap(*self*, *cmap*)
Set the colormap for luminance data.

Parameters

cmap

[*Colormap* or str or None]

set_color(*self*, *c*)
Set both the edgecolor and the facecolor.

Parameters

c

[color or list of rgba tuples]

See also:

Collection.set_facecolor, *Collection.set_edgecolor*

For setting the edge or face color individually.

set_contains(*self*, *picker*)
[*Deprecated*] Define a custom contains test for the artist.

The provided callable replaces the default *contains* method of the artist.

Parameters

picker

[callable] A custom picker function to evaluate if an event is within the artist. The function must have the signature:

```
def contains(artist: Artist, event: MouseEvent) -> bool, dict
```

that returns:

- a bool indicating if the event is within the artist
- a dict of additional information. The dict should at least return the same information as the default `contains()` implementation of the respective artist, but may provide additional information.

Notes

Deprecated since version 3.3.

set_dashes(*self*, *ls*)
Alias for *set_linestyle*.

set_ec(*self*, *c*)
Alias for *set_edgecolor*.

set_edgecolor(*self*, *c*)
Set the edgecolor(s) of the collection.

Parameters

c

[color or list of colors or 'face'] The collection edgecolor(s). If a sequence, the patches cycle through it. If 'face', match the facecolor.

`set_edgecolors`(*self*, *c*)
Alias for *set_edgecolor*.

`set_facecolor`(*self*, *c*)
Set the facecolor(s) of the collection. *c* can be a color (all patches have same color), or a sequence of colors; if it is a sequence the patches will cycle through the sequence.

If *c* is 'none', the patch will not be filled.

Parameters

c

[color or list of colors]

`set_facecolors`(*self*, *c*)
Alias for *set_facecolor*.

`set_fc`(*self*, *c*)
Alias for *set_facecolor*.

`set_figure`(*self*, *fig*)
Set the *Figure* instance the artist belongs to.

Parameters

fig

[*Figure*]

`set_gid`(*self*, *gid*)
Set the (group) id for the artist.

Parameters

gid

[str]

`set_hatch`(*self*, *hatch*)
Set the hatching pattern

hatch can be one of:

```
/   - diagonal hatching
\   - back diagonal
|   - vertical
-   - horizontal
+   - crossed
x   - crossed diagonal
o   - small circle
O   - large circle
.   - dots
*   - stars
```

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Unlike other properties such as linewidth and colors, hatching can only be specified for the collection as a whole, not separately for each member.

Parameters

hatch

[{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}]

`set_in_layout`(*self*, *in_layout*)

Set if artist is to be included in layout calculations, E.g. *Constrained Layout Guide*, *`Figure.tight_layout()`*, and `fig.savefig(fname, bbox_inches='tight')`.

Parameters

in_layout

[bool]

`set_joinstyle`(*self*, *js*)

Set the joinstyle for the collection (for all its elements).

Parameters

js

[{'miter', 'round', 'bevel'}] The joinstyle.

`set_label`(*self*, *s*)

Set a label that will be displayed in the legend.

Parameters

s

[object] *s* will be converted to a string by calling `str`.

set_linestyle(*self*, *ls*)
Set the linestyle(s) for the collection.

linestyle	description
'-' or 'solid'	solid line
'--' or 'dashed'	dashed line
'-.' or 'dashdot'	dash-dotted line
':' or 'dotted'	dotted line

Alternatively a dash tuple of the following form can be provided:

```
(offset, onoffseq),
```

where `onoffseq` is an even length tuple of on and off ink in points.

Parameters

ls

[str or tuple or list thereof] Valid values for individual linestyles include {'-', '--', '-.', ':', '', (offset, on-off-seq)}. See *Line2D.set_linestyle* for a complete description.

set_linestyles(*self*, *ls*)
Alias for *set_linestyle*.

set_linewidth(*self*, *lw*)
Set the linewidth(s) for the collection. *lw* can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence

Parameters

lw

[float or list of floats]

set_linewidths(*self*, *lw*)
Alias for *set_linewidth*.

set_ls(*self*, *ls*)
Alias for *set_linestyle*.

set_lw(*self*, *lw*)
Alias for *set_linewidth*.

set_norm(*self*, *norm*)
Set the normalization instance.

Parameters

norm

[*Normalize* or None]

Notes

If there are any colorbars using the mappable for this norm, setting the norm of the mappable will reset the norm, locator, and formatters on the colorbar to default.

`set_offset_position`(*self*, *offset_position*)
[*Deprecated*] Set how offsets are applied. If *offset_position* is 'screen' (default) the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is 'data', the offset is applied before the master transform, i.e., the offsets are in data coordinates.

Parameters

offset_position

[{'screen', 'data'}]

Notes

Deprecated since version 3.3.

`set_offsets`(*self*, *offsets*)
Set the offsets for the collection.

Parameters

offsets

[array-like (N, 2) or (2,)]

`set_path_effects`(*self*, *path_effects*)
Set the path effects.

Parameters

path_effects

[*AbstractPathEffect*]

`set_paths`(*self*)

`set_picker`(*self*, *picker*)
Define the picking behavior of the artist.

Parameters

picker

[None or bool or callable] This can be one of the following:

- *None*: Picking is disabled for this artist (default).
- A boolean: If *True* then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist.
- A function: If picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

to determine the hit test. if the mouse event is over the artist, return *hit=True* and props is a dictionary of properties you want added to the PickEvent attributes.

- *deprecated*: For *Line2D* only, *picker* can also be a float that sets the tolerance for checking whether an event occurred "on" the line; this is deprecated. Use *Line2D.set_pickradius* instead.

`set_pickradius`(*self*, *pr*)

Set the pick radius used for containment tests.

Parameters

d

[float] Pick radius, in points.

`set_rasterized`(*self*, *rasterized*)

Force rasterized (bitmap) drawing in vector backend output.

Defaults to None, which implies the backend's default behavior.

Parameters

rasterized

[bool or None]

`set_sketch_params`(*self*, *scale=None*, *length=None*, *randomness=None*)

Sets the sketch parameters.

Parameters

scale

[float, optional] The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is `None`, or not provided, no sketch filter will be provided.

length

[float, optional] The length of the wiggle along the line, in pixels (default 128.0)

randomness

[float, optional] The scale factor by which the length is shrunken or expanded (default 16.0)

`set_snap`(*self*, *snap*)
Set the snapping behavior.

Snapping aligns positions with the pixel grid, which results in clearer images. For example, if a black line of 1px width was defined at a position in between two pixels, the resulting image would contain the interpolated value of that line in the pixel grid, which would be a grey value on both adjacent pixel positions. In contrast, snapping will move the line to the nearest integer pixel value, so that the resulting image will really contain a 1px wide black line.

Snapping is currently only supported by the Agg and MacOSX backends.

Parameters

snap

[bool or None] Possible values:

- *True*: Snap vertices to the nearest pixel center.
- *False*: Do not modify vertex positions.
- *None*: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center.

`set_transform`(*self*, *t*)
Set the artist transform.

Parameters

t

[*Transform*]

`set_url`(*self*, *url*)
Set the url for the artist.

Parameters

url

[str]

`set_urls`(*self*, *urls*)

Parameters

urls

[list of str or None]

Notes

URLs are currently only implemented by the SVG backend. They are ignored by all other backends.

`set_visible`(*self*, *b*)
Set the artist's visibility.

Parameters

b

[bool]

`set_zorder`(*self*, *level*)
Set the zorder for the artist. Artists with lower zorder values are drawn first.

Parameters

level

[float]

`property stale`
Whether the artist is 'stale' and needs to be re-drawn for the output to match the internal state of the artist.

`property sticky_edges`
`x` and `y` sticky edge lists for autoscaling.

When performing autoscaling, if a data limit coincides with a value in the corresponding sticky_edges list, then no margin will be added--the view limit "sticks" to the edge. A typical use case is histograms, where one usually expects no margin on the bottom edge (0) of the histogram.

This attribute cannot be assigned to; however, the `x` and `y` lists can be modified in place as needed.

Examples

```
>>> artist.sticky_edges.x[:] = (xmin, xmax)
>>> artist.sticky_edges.y[:] = (ymin, ymax)
```

to_rgba(*self*, *x*, *alpha=None*, *bytes=False*, *norm=True*)
Return a normalized rgba array corresponding to *x*.

In the normal case, *x* is a 1-D or 2-D sequence of scalars, and the corresponding ndarray of rgba values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already rgb or rgba, such as might have been read from an image file. If *x* is an ndarray with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an rgb or rgba array, and no mapping will be done. The array can be uint8, or it can be floating point with values in the 0-1 range; otherwise a ValueError will be raised. If it is a masked array, the mask will be ignored. If the last dimension is 3, the *alpha* kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the *alpha* kwarg is ignored; it does not replace the pre-existing alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if *bytes* is *False* (default), the rgba array will be floats in the 0-1 range; if it is *True*, the returned rgba array will be uint8 in the 0 to 255 range.

If norm is False, no normalization of the input data is performed, and it is assumed to be in the range (0-1).

update(*self*, *props*)
Update this artist's properties from the dict *props*.

Parameters

props

[dict]

property update_dict

update_from(*self*, *other*)
Copy properties from other to self.

update_scalarmappable(*self*)
Update colors from the scalar mappable array, if it is not None.

zorder = 0

18.17 matplotlib.colorbar

Colorbars are a visualization of the mapping from scalar values to colors. In Matplotlib they are drawn into a dedicated *Axes*.

Note: Colorbars are typically created through *Figure.colorbar* or its pyplot wrapper *pyplot.colorbar*, which use *make_axes* and *Colorbar* internally.

As an end-user, you most likely won't have to call the methods or instantiate the classes in this module explicitly.

ColorbarBase

The base class with full colorbar drawing functionality. It can be used as-is to make a colorbar for a given colormap; a mappable object (e.g., image) is not needed.

Colorbar

On top of *ColorbarBase* this connects the colorbar with a *ScalarMappable* such as an image or contour plot.

ColorbarPatch

A specialized *Colorbar* to support hatched contour plots.

make_axes()

Create an *Axes* suitable for a colorbar. This functions can be used with figures containing a single axes or with freely placed axes.

make_axes_gridspec()

Create a *SubplotBase* suitable for a colorbar. This function should be used for adding a colorbar to a *GridSpec*.

class matplotlib.colorbar.**Colorbar**(*ax*, *mappable*, ***kwargs*)

Bases: *matplotlib.colorbar.ColorbarBase*

This class connects a *ColorbarBase* to a *ScalarMappable* such as an *AxesImage* generated via *imshow*.

Note: This class is not intended to be instantiated directly; instead, use *Figure.colorbar* or *pyplot.colorbar* to create a colorbar.

---

add_lines(*self*, *CS*, *erase=True*)

Add the lines from a non-filled *ContourSet* to the colorbar.

Parameters

CS

[*ContourSet*] The line positions are taken from the ContourSet levels. The ContourSet must not be filled.

erase

[bool, default: True] Whether to remove any previously added lines.

`on_mappable_changed`(*self*, *mappable*)

[*Deprecated*] Update this colorbar to match the mappable's properties.

Typically this is automatically registered as an event handler by *colorbar_factory()* and should not be called manually.

Notes

Deprecated since version 3.3.

`remove`(*self*)

Remove this colorbar from the figure.

If the colorbar was created with `use_gridspec=True` the previous gridspec is restored.

`update_bruteforce`(*self*, *mappable*)

[*Deprecated*] Destroy and rebuild the colorbar. This is intended to become obsolete, and will probably be deprecated and then removed. It is not called when the pyplot.colorbar function or the Figure.colorbar method are used to create the colorbar.

Notes

Deprecated since version 3.3.

`update_normal`(*self*, *mappable*)

Update solid patches, lines, etc.

This is meant to be called when the norm of the image or contour plot to which this colorbar belongs changes.

If the norm on the mappable is different than before, this resets the locator and formatter for the axis, so if these have been customized, they will need to be customized again. However, if the norm only changes values of *vmin*, *vmax* or *cmap* then the old formatter and locator will be preserved.

class matplotlib.colorbar.**ColorbarBase**(*ax*, ***, *cmap=None*, *norm=None*, *alpha=None*, *values=None*, *boundaries=None*, *orientation='vertical'*, *ticklocation='auto'*, *extend=None*, *spacing='uniform'*, *ticks=None*, *format=None*, *drawedges=False*, *filled=True*, *extendfrac=None*, *extendrect=False*, *label=''*)

Bases: `object`

Draw a colorbar in an existing axes.

There are only some rare cases in which you would work directly with a *ColorbarBase* as an end-user. Typically, colorbars are used with *ScalarMappable*s such as an *AxesImage* generated via *imshow*. For these cases you will use *Colorbar* and likely create it via *pyplot.colorbar* or *Figure.colorbar*.

The main application of using a *ColorbarBase* explicitly is drawing colorbars that are not associated with other elements in the figure, e.g. when showing a colormap by itself.

If the *cmap* kwarg is given but *boundaries* and *values* are left as None, then the colormap will be displayed on a 0-1 scale. To show the under- and over-value colors, specify the *norm* as:

```
norm=colors.Normalize(clip=False)
```

To show the colors versus index instead of on the 0-1 scale, use:

```
norm=colors.NoNorm()
```

Useful public methods are *set_label()* and *add_lines()*.

Parameters

ax

[*Axes*] The *Axes* instance in which the colorbar is drawn.

cmap

[*Colormap*, default: `rcParams["image.cmap"]` (default: `'viridis'`)] The colormap to use.

norm

[*Normalize*]

alpha

[float] The colorbar transparency between 0 (transparent) and 1 (opaque).

values

boundaries

orientation

[{'vertical', 'horizontal'}]

ticklocation

[{'auto', 'left', 'right', 'top', 'bottom'}]

extend

[{'neither', 'both', 'min', 'max'}]

spacing

[{'uniform', 'proportional'}]

ticks

[*Locator* or array-like of float]

format

[str or *Formatter*]

drawedges

[bool]

filled

[bool]

extendfrac

extendrec

label

[str]

Attributes

ax

[*Axes*] The *Axes* instance in which the colorbar is drawn.

lines

[list] A list of *LineCollection* if lines were drawn, otherwise an empty list.

dividers

[*LineCollection*] A LineCollection if *drawedges* is `True`, otherwise `None`.

`add_lines`(*self*, *levels*, *colors*, *linewidths*, *erase=True*)

Draw lines on the colorbar.

The lines are appended to the list `lines`.

Parameters

levels

[array-like] The positions of the lines.

colors

[color or list of colors] Either a single color applying to all lines or one color value for each line.

linewidths

[float or array-like] Either a single linewidth applying to all lines or one linewidth for each line.

erase

[bool, default: True] Whether to remove any previously added lines.

`config_axis`(*self*)
[*Deprecated*]

Notes

Deprecated since version 3.3:

`draw_all`(*self*)
Calculate any free parameters based on the current cmap and norm, and do all the drawing.

`get_ticks`(*self*, *minor=False*)
Return the x ticks as a list of locations.

`minorticks_off`(*self*)
Turn the minor ticks of the colorbar off.

`minorticks_on`(*self*)
Turn the minor ticks of the colorbar on without extruding into the "extend regions".

`n_rasterize = 50`

`remove`(*self*)
Remove this colorbar from the figure.

`set_alpha`(*self*, *alpha*)
Set the transparency between 0 (transparent) and 1 (opaque).

`set_label`(*self*, *label*, *, *loc=None*, ***kwargs*)
Add a label to the long axis of the colorbar.

set_ticklabels(*self*, *ticklabels*, *update_ticks=True*)
Set tick labels.

Tick labels are updated immediately unless *update_ticks* is *False*, in which case one should call *update_ticks* explicitly.

set_ticks(*self*, *ticks*, *update_ticks=True*)
Set tick locations.

Parameters

ticks

[array-like or *Locator* or None] The tick positions can be hard-coded by an array of values; or they can be defined by a *Locator*. Setting to *None* reverts to using a default locator.

update_ticks

[bool, default: True] If True, tick locations are updated immediately. If False, the user has to call *update_ticks* later to update the ticks.

update_ticks(*self*)
Force the update of the ticks and ticklabels. This must be called whenever the tick locator and/or tick formatter changes.

class matplotlib.colorbar.ColorbarPatch(*ax*, *mappable*, ***kw*)
Bases: *matplotlib.colorbar.Colorbar*

A Colorbar that uses a list of *Patch* instances rather than the default *PatchCollection* created by *pcolor*, because the latter does not allow the hatch pattern to vary among the members of the collection.

matplotlib.colorbar.colorbar_factory(*cax*, *mappable*, ***kwargs*)
Create a colorbar on the given axes for the given mappable.

Note: This is a low-level function to turn an existing axes into a colorbar axes. Typically, you'll want to use *colorbar* instead, which automatically handles creation and placement of a suitable axes as well.

Parameters

cax

[*Axes*] The *Axes* to turn into a colorbar.

mappable

[*ScalarMappable*] The mappable to be described by the colorbar.

****kwargs**

Keyword arguments are passed to the respective colorbar class.

Returns

Colorbar **or** *ColorbarPatch*

The created colorbar instance. *ColorbarPatch* is only used if *mappable* is a *ContourSet* with hatches.

matplotlib.colorbar.**make_axes**(*parents*, *location=None*, *orientation=None*, *fraction=0.15*, *shrink=1.0*, *aspect=20*, ***kw*)

Create an *Axes* suitable for a colorbar.

The axes is placed in the figure of the *parents* axes, by resizing and repositioning *parents*.

Parameters

parents

[*Axes* or list of *Axes*] The Axes to use as parents for placing the colorbar.

location

[None or {'left', 'right', 'top', 'bottom'}] The position, relative to *parents*, where the colorbar axes should be created. If None, the value will either come from the given `orientation`, else it will default to 'right'.

orientation

[None or {'vertical', 'horizontal'}] The orientation of the colorbar. Typically, this keyword shouldn't be used, as it can be derived from the `location` keyword.

fraction

[float, default: 0.15] Fraction of original axes to use for colorbar.

shrink

[float, default: 1.0] Fraction by which to multiply the size of the colorbar.

aspect

[float, default: 20] Ratio of long to short dimensions.

Returns

cax

[*Axes*] The child axes.

kw

[dict] The reduced keyword dictionary to be passed when creating the colorbar instance.

Other Parameters

pad

[float, default: 0.05 if vertical, 0.15 if horizontal] Fraction of original axes between colorbar and new image axes.

anchor

[(float, float), optional] The anchor point of the colorbar axes. Defaults to (0.0, 0.5) if vertical; (0.5, 1.0) if horizontal.

panchor

[(float, float), or *False*, optional] The anchor point of the colorbar parent axes. If *False*, the parent axes' anchor will be unchanged. Defaults to (1.0, 0.5) if vertical; (0.5, 0.0) if horizontal.

matplotlib.colorbar.**make_axes_gridspec**(*parent*, ***, *fraction=0.15*, *shrink=1.0*, *aspect=20*, ***kw*)

Create a *SubplotBase* suitable for a colorbar.

The axes is placed in the figure of the *parent* axes, by resizing and repositioning *parent*.

This function is similar to *make_axes*. Primary differences are

- *make_axes_gridspec* only handles the *orientation* keyword and cannot handle the *location* keyword.
- *make_axes_gridspec* should only be used with a *SubplotBase* parent.
- *make_axes* creates an *Axes*; *make_axes_gridspec* creates a *SubplotBase*.
- *make_axes* updates the position of the parent. *make_axes_gridspec* replaces the `grid_spec` attribute of the parent with a new one.

While this function is meant to be compatible with *make_axes*, there could be some minor differences.

Parameters

parent

[*Axes*] The Axes to use as parent for placing the colorbar.

fraction

[float, default: 0.15] Fraction of original axes to use for colorbar.

shrink

[float, default: 1.0] Fraction by which to multiply the size of the colorbar.

aspect

[float, default: 20] Ratio of long to short dimensions.

Returns

cax

[*SubplotBase*] The child axes.

kw

[dict] The reduced keyword dictionary to be passed when creating the colorbar instance.

Other Parameters

orientation

[{'vertical', 'horizontal'}, default: 'vertical'] The orientation of the colorbar.

pad

[float, default: 0.05 if vertical, 0.15 if horizontal] Fraction of original axes between colorbar and new image axes.

anchor

[(float, float), optional] The anchor point of the colorbar axes. Defaults to (0.0, 0.5) if vertical; (0.5, 1.0) if horizontal.

panchor

[(float, float), or *False*, optional] The anchor point of the colorbar parent axes. If *False*, the parent axes' anchor will be unchanged. Defaults to (1.0, 0.5) if vertical; (0.5, 0.0) if horizontal.

18.18 matplotlib.colors

The Color *tutorials* and examples demonstrate how to set colors and colormaps.

A module for converting numbers or color arguments to *RGB* or *RGBA*.

RGB and *RGBA* are sequences of, respectively, 3 or 4 floats in the range 0-1.

This module includes functions and classes for color specification conversions, and for mapping numbers to colors in a 1-D array of colors called a colormap.

Mapping data onto colors using a colormap typically involves two steps: a data array is first mapped onto the range 0-1 using a subclass of *Normalize*, then this number is mapped to a color using a subclass of *Colormap*. Two sublasses of *Colormap* provided here: *LinearSegmentedColormap*, which uses piecewise-linear interpolation to define colormaps, and *ListedColormap*, which makes a colormap from a list of colors.

See also:

Creating Colormaps in Matplotlib for examples of how to make colormaps and

Choosing Colormaps in Matplotlib for a list of built-in colormaps.

Colormap Normalization for more details about data normalization

More colormaps are available at palettable.

The module also provides functions for checking whether an object can be interpreted as a color (*is_color_like*), for converting such an object to an RGBA tuple (*to_rgba*) or to an HTML-like hex string in the "#rrggbb" format (*to_hex*), and a sequence of colors to an (n, 4) RGBA array (*to_rgba_array*). Caching is used for efficiency.

Matplotlib recognizes the following formats to specify a color:

- an RGB or RGBA (red, green, blue, alpha) tuple of float values in closed interval `[0, 1]` (e.g., `(0.1, 0.2, 0.5)` or `(0.1, 0.2, 0.5, 0.3)`);
- a hex RGB or RGBA string (e.g., `'#0f0f0f'` or `'#0f0f0f80'`; case-insensitive);
- a shorthand hex RGB or RGBA string, equivalent to the hex RGB or RGBA string obtained by duplicating each character, (e.g., `'#abc'`, equivalent to `'#aabbcc'`, or `'#abcd'`, equivalent to `'#aabbccdd'`; case-insensitive);
- a string representation of a float value in `[0, 1]` inclusive for gray level (e.g., `'0.5'`);
- one of the characters `{'b', 'g', 'r', 'c', 'm', 'y', 'k', 'w'}`, which are shorthand notations for shades of blue, green, red, cyan, magenta, yellow, black, and white. Note that the colors `'g', 'c', 'm', 'y'` do not coincide with the X11/CSS4 colors. Their particular shades were chosen for better visibility of colored lines against typical backgrounds.
- a X11/CSS4 color name (case-insensitive);
- a name from the xkcd color survey, prefixed with `'xkcd:'` (e.g., `'xkcd:sky blue'`; case insensitive);
- one of the Tableau Colors from the 'T10' categorical palette (the default color cycle): `{'tab:blue', 'tab:orange', 'tab:green', 'tab:red', 'tab:purple', 'tab:brown', 'tab:pink', 'tab:gray', 'tab:olive', 'tab:cyan'}` (case-insensitive);
- a "CN" color spec, i.e. 'C' followed by a number, which is an index into the default property cycle (`rcParams["axes.prop_cycle"]` (default: `cycler('color', ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728', '#9467bd',`

`'#8c564b', '#e377c2', '#7f7f7f', '#bcbd22', '#17becf']))`); the indexing is intended to occur at rendering time, and defaults to black if the cycle does not include color.

18.18.1 Classes

BoundaryNorm(boundaries, ncolors[, clip, extend])	Generate a colormap index based on discrete intervals.
Colormap(name[, N])	Baseclass for all scalar to RGBA mappings.
DivergingNorm(**kwargs)	[*Deprecated*]
LightSource([azdeg, altdeg, hsv_min_val, ...])	Create a light source coming from the specified azimuth and elevation.
LinearSegmentedColormap(name, segmentdata[, ...])	Colormap objects based on lookup tables using linear segments.
ListedColormap(colors[, name, N])	Colormap object generated from a list of colors.
LogNorm([vmin, vmax, clip])	Normalize a given value to the 0-1 range on a log scale.
NoNorm([vmin, vmax, clip])	Dummy replacement for *Normalize*, for the case where we want to use indices directly in a *ScalarMappable*.
Normalize([vmin, vmax, clip])	A class which, when called, linearly normalizes data into the `[0.0, 1.0]` interval.
PowerNorm(gamma[, vmin, vmax, clip])	Linearly map a given value to the 0-1 range and then apply a power-law normalization over that range.
SymLogNorm(linthresh[, linscale, vmin, ...])	The symmetrical logarithmic scale is logarithmic in both the positive and negative directions from the origin.
TwoSlopeNorm(vcenter[, vmin, vmax])	Normalize data with a set center.

matplotlib.colors.BoundaryNorm

class matplotlib.colors.**BoundaryNorm**(*boundaries*, *ncolors*, *clip=False*, ***, *extend='neither'*)

Bases: *matplotlib.colors.Normalize*

Generate a colormap index based on discrete intervals.

Unlike *Normalize* or *LogNorm*, *BoundaryNorm* maps values to integers instead of to the interval 0-1.

Mapping to the 0-1 interval could have been done via piece-wise linear interpolation, but using integers seems simpler, and reduces the number of conversions back and forth between integer and floating point.

Parameters

boundaries

[array-like] Monotonically increasing sequence of boundaries

ncolors

[int] Number of colors in the colormap to be used

clip

[bool, optional] If clip is `True`, out of range values are mapped to 0 if they are below `boundaries[0]` or mapped to `ncolors - 1` if they are above `boundaries[-1]`.

If clip is `False`, out of range values are mapped to -1 if they are below `boundaries[0]` or mapped to *ncolors* if they are above `boundaries[-1]`. These are then converted to valid indices by *Colormap.__call__*.

extend

[{'neither', 'both', 'min', 'max'}, default: 'neither'] Extend the number of bins to include one or both of the regions beyond the boundaries. For example, if `extend` is 'min', then the color to which the region between the first pair of boundaries is mapped will be distinct from the first color in the colormap, and by default a *Colorbar* will be drawn with the triangle extension on the left or lower end.

Returns

int16 scalar or array

Notes

boundaries defines the edges of bins, and data falling within a bin is mapped to the color with the same index.

If the number of bins, including any extensions, is less than *ncolors*, the color index is chosen by linear interpolation, mapping the `[0, nbins - 1]` range onto the `[0, ncolors - 1]` range.

`__call__`(*self*, *value*, *clip=None*)

Normalize *value* data in the `[vmin, vmax]` interval into the `[0.0, 1.0]` interval and return it.

Parameters

value

Data to normalize.

clip

[bool] If `None`, defaults to `self.clip` (which defaults to `False`).

Notes

If not already initialized, `self.vmin` and `self.vmax` are initialized using `self.autoscale_None(value)`.

`__init__`(*self*, *boundaries*, *ncolors*, *clip=False*, *, *extend='neither'*)

Parameters

boundaries

[array-like] Monotonically increasing sequence of boundaries

ncolors

[int] Number of colors in the colormap to be used

clip

[bool, optional] If clip is `True`, out of range values are mapped to 0 if they are below `boundaries[0]` or mapped to `ncolors - 1` if they are above `boundaries[-1]`.

If clip is `False`, out of range values are mapped to -1 if they are below `boundaries[0]` or mapped to *ncolors* if they are above `boundaries[-1]`. These are then converted to valid indices by *Colormap.__call__*.

extend

[{'neither', 'both', 'min', 'max'}, default: 'neither'] Extend the number of bins to include one or both of the regions beyond the boundaries. For example, if `extend` is 'min', then the color to which the region between the first pair of boundaries is mapped will be distinct from the first color in the colormap, and by default a *Colorbar* will be drawn with the triangle extension on the left or lower end.

Returns

int16 scalar or array

Notes

boundaries defines the edges of bins, and data falling within a bin is mapped to the color with the same index.

If the number of bins, including any extensions, is less than *ncolors*, the color index is chosen by linear interpolation, mapping the `[0, nbins - 1]` range onto the `[0, ncolors - 1]` range.

`__module__ = 'matplotlib.colors'`

`__slotnames__ = []`

`inverse`(*self*, *value*)

Raises

ValueError

BoundaryNorm is not invertible, so calling this method will always raise an error

Examples using `matplotlib.colors.BoundaryNorm`

- sphx_glr_gallery_lines_bars_and_markers_multicolored_line.py
- sphx_glr_gallery_images_contours_and_fields_image_annotated_heatmap.py
- sphx_glr_gallery_images_contours_and_fields_image_masked.py
- sphx_glr_gallery_images_contours_and_fields_pcolormesh_levels.py
- sphx_glr_gallery_specialty_plots_leftventricle_bulleye.py
- sphx_glr_gallery_userdemo_colormap_normalizations.py
- *Customized Colorbars Tutorial*
- *Colormap Normalization*

matplotlib.colors.Colormap

`class matplotlib.colors.Colormap`(*name*, *N=256*)

Bases: `object`

Baseclass for all scalar to RGBA mappings.

Typically, Colormap instances are used to convert data values (floats) from the interval `[0, 1]` to the RGBA color that the respective Colormap represents. For scaling of data into the `[0, 1]` interval see *matplotlib.colors.Normalize*. Subclasses of *matplotlib.cm.ScalarMappable* make heavy use of this `data -> normalize -> map-to-color` processing chain.

Parameters

name

[str] The name of the colormap.

N

[int] The number of rgb quantization levels.

`__call__`(*self*, *X*, *alpha=None*, *bytes=False*)

Parameters

X

[float or int, ndarray or scalar] The data value(s) to convert to RGBA. For floats, X should be in the interval `[0.0, 1.0]` to return the RGBA values `X*100` percent along the Colormap line. For integers, X should be in the interval `[0, Colormap.N)` to return RGBA values *indexed* from the Colormap with index `X`.

alpha

[float, None] Alpha must be a scalar between 0 and 1, or None.

bytes

[bool] If False (default), the returned RGBA values will be floats in the interval `[0, 1]` otherwise they will be uint8s in the interval `[0, 255]`.

Returns

Tuple of RGBA values if X is scalar, otherwise an array of RGBA values with a shape of `X.shape + (4, )`.

`__copy__`(*self*)

`__dict__ = mappingproxy({'__module__': 'matplotlib.colors', '__doc__': '\n Baseclass for`

`__init__`(*self*, *name*, *N=256*)

Parameters

name

[str] The name of the colormap.

N

[int] The number of rgb quantization levels.

`__module__ = 'matplotlib.colors'`

`__weakref__`
list of weak references to the object (if defined)

`colorbar_extend`
When this colormap exists on a scalar mappable and colorbar_extend is not False, colorbar creation will pick up `colorbar_extend` as the default value for the `extend` keyword in the *matplotlib.colorbar.Colorbar* constructor.

`is_gray`(*self*)

`reversed`(*self*, *name=None*)
Return a reversed instance of the Colormap.

Note: This function is not implemented for base class.

Parameters

name

[str, optional] The name for the reversed colormap. If it's None the name will be the name of the parent colormap + "_r".

See also:

LinearSegmentedColormap.reversed

ListedColormap.reversed

`set_bad`(*self*, *color='k'*, *alpha=None*)
Set the color for masked values.

`set_over`(*self*, *color='k'*, *alpha=None*)
Set the color for high out-of-range values when `norm.clip = False`.

`set_under`(*self*, *color='k'*, *alpha=None*)
Set the color for low out-of-range values when `norm.clip = False`.

Examples using `matplotlib.colors.Colormap`

- sphx_glr_gallery_lines_bars_and_markers_multicolored_line.py
- sphx_glr_gallery_images_contours_and_fields_contourf_demo.py
- sphx_glr_gallery_color_custom_cmap.py
- sphx_glr_gallery_specialty_plots_leftventricle_bulleye.py
- *Customized Colorbars Tutorial*
- *Creating Colormaps in Matplotlib*

matplotlib.colors.DivergingNorm

class matplotlib.colors.DivergingNorm(**kwargs)

Bases: *matplotlib.colors.TwoSlopeNorm*

[*Deprecated*]

Notes

Deprecated since version 3.2:

Normalize data with a set center.

Useful when mapping data with an unequal rates of change around a conceptual center, e.g., data that range from -2 to 4, with 0 as the midpoint.

Parameters

vcenter

[float] The data value that defines `0.5` in the normalization.

vmin

[float, optional] The data value that defines `0.0` in the normalization. Defaults to the min value of the dataset.

vmax

[float, optional] The data value that defines `1.0` in the normalization. Defaults to the the max value of the dataset.

Examples

This maps data value -4000 to 0., 0 to 0.5, and +10000 to 1.0; data between is linearly interpolated:

```
>>> import matplotlib.colors as mcolors
>>> offset = mcolors.TwoSlopeNorm(vmin=-4000.,
                                  vcenter=0., vmax=10000)
>>> data = [-4000., -2000., 0., 2500., 5000., 7500., 10000.]
>>> offset(data)
array([0., 0.25, 0.5, 0.625, 0.75, 0.875, 1.0])
```

__init__(*self*, *vcenter*, *vmin=None*, *vmax=None*)

Normalize data with a set center.

Useful when mapping data with an unequal rates of change around a conceptual center, e.g., data that range from -2 to 4, with 0 as the midpoint.

Parameters

vcenter

[float] The data value that defines `0.5` in the normalization.

vmin

[float, optional] The data value that defines `0.0` in the normalization. Defaults to the min value of the dataset.

vmax

[float, optional] The data value that defines `1.0` in the normalization. Defaults to the the max value of the dataset.

Examples

This maps data value -4000 to 0., 0 to 0.5, and +10000 to 1.0; data between is linearly interpolated:

```
>>> import matplotlib.colors as mcolors
>>> offset = mcolors.TwoSlopeNorm(vmin=-4000.,
                                  vcenter=0., vmax=10000)
>>> data = [-4000., -2000., 0., 2500., 5000., 7500., 10000.]
>>> offset(data)
array([0., 0.25, 0.5, 0.625, 0.75, 0.875, 1.0])
```

__module__ = 'matplotlib.colors'

Examples using `matplotlib.colors.DivergingNorm`

matplotlib.colors.LightSource

class matplotlib.colors.**LightSource**(*azdeg=315*, *altdeg=45*, *hsv_min_val=0*, *hsv_max_val=1*, *hsv_min_sat=1*, *hsv_max_sat=0*)

Bases: `object`

Create a light source coming from the specified azimuth and elevation. Angles are in degrees, with the azimuth measured clockwise from north and elevation up from the zero plane of the surface.

shade is used to produce "shaded" rgb values for a data array. *shade_rgb* can be used to combine an rgb image with an elevation map. *hillshade* produces an illumination map of a surface.

Specify the azimuth (measured clockwise from south) and altitude (measured up from the plane of the surface) of the light source in degrees.

Parameters

azdeg

[float, default: 315 degrees (from the northwest)] The azimuth (0-360, degrees clockwise from North) of the light source.

altdeg

[float, default: 45 degrees] The altitude (0-90, degrees up from horizontal) of the light source.

Notes

For backwards compatibility, the parameters *hsv_min_val*, *hsv_max_val*, *hsv_min_sat*, and *hsv_max_sat* may be supplied at initialization as well. However, these parameters will only be used if "blend_mode='hsv'" is passed into *shade* or *shade_rgb*. See the documentation for *blend_hsv* for more details.

`__dict__ = mappingproxy({'__module__': 'matplotlib.colors', '__doc__': '\n Create a light`

`__init__`(*self*, *azdeg=315*, *altdeg=45*, *hsv_min_val=0*, *hsv_max_val=1*, *hsv_min_sat=1*, *hsv_max_sat=0*)

Specify the azimuth (measured clockwise from south) and altitude (measured up from the plane of the surface) of the light source in degrees.

Parameters

azdeg

[float, default: 315 degrees (from the northwest)] The azimuth (0-360, degrees clockwise from North) of the light source.

altdeg

[float, default: 45 degrees] The altitude (0-90, degrees up from horizontal) of the light source.

Notes

For backwards compatibility, the parameters *hsv_min_val*, *hsv_max_val*, *hsv_min_sat*, and *hsv_max_sat* may be supplied at initialization as well. However, these parameters will only be used if "blend_mode='hsv'" is passed into *shade* or *shade_rgb*. See the documentation for *blend_hsv* for more details.

`__module__ = 'matplotlib.colors'`

`__weakref__`

list of weak references to the object (if defined)

`blend_hsv`(*self*, *rgb*, *intensity*, *hsv_max_sat=None*, *hsv_max_val=None*, *hsv_min_val=None*, *hsv_min_sat=None*)

Take the input data array, convert to HSV values in the given colormap, then adjust those color values to give the impression of a shaded relief map with

a specified light source. RGBA values are returned, which can then be used to plot the shaded image with imshow.

The color of the resulting image will be darkened by moving the (s, v) values (in hsv colorspace) toward (hsv_min_sat, hsv_min_val) in the shaded regions, or lightened by sliding (s, v) toward (hsv_max_sat, hsv_max_val) in regions that are illuminated. The default extremes are chose so that completely shaded points are nearly black (s = 1, v = 0) and completely illuminated points are nearly white (s = 0, v = 1).

Parameters

rgb

[ndarray] An MxNx3 RGB array of floats ranging from 0 to 1 (color image).

intensity

[ndarray] An MxNx1 array of floats ranging from 0 to 1 (grayscale image).

hsv_max_sat

[number, default: 1] The maximum saturation value that the *intensity* map can shift the output image to.

hsv_min_sat

[number, optional] The minimum saturation value that the *intensity* map can shift the output image to. Defaults to 0.

hsv_max_val

[number, optional] The maximum value ("v" in "hsv") that the *intensity* map can shift the output image to. Defaults to 1.

hsv_min_val

[number, optional] The minimum value ("v" in "hsv") that the *intensity* map can shift the output image to. Defaults to 0.

Returns

ndarray

An MxNx3 RGB array representing the combined images.

`blend_overlay`(*self*, *rgb*, *intensity*)

Combines an rgb image with an intensity map using "overlay" blending.

Parameters

rgb

[ndarray] An MxNx3 RGB array of floats ranging from 0 to 1 (color image).

intensity

[ndarray] An MxNx1 array of floats ranging from 0 to 1 (grayscale image).

Returns

ndarray

An MxNx3 RGB array representing the combined images.

`blend_soft_light`(*self*, *rgb*, *intensity*)

Combine an rgb image with an intensity map using "soft light" blending, using the "pegtop" formula.

Parameters

rgb

[ndarray] An MxNx3 RGB array of floats ranging from 0 to 1 (color image).

intensity

[ndarray] An MxNx1 array of floats ranging from 0 to 1 (grayscale image).

Returns

ndarray

An MxNx3 RGB array representing the combined images.

`property direction`

The unit vector direction towards the light source.

`hillshade`(*self*, *elevation*, *vert_exag=1*, *dx=1*, *dy=1*, *fraction=1.0*)

Calculate the illumination intensity for a surface using the defined azimuth and elevation for the light source.

This computes the normal vectors for the surface, and then passes them on to *shade_normals*

Parameters

elevation

[array-like] A 2d array (or equivalent) of the height values used to generate an illumination map

vert_exag

[number, optional] The amount to exaggerate the elevation values by when calculating illumination. This can be used either to correct for differences in units between the x-y coordinate system and the elevation coordinate system (e.g. decimal degrees vs. meters) or to exaggerate or de-emphasize topographic effects.

dx

[number, optional] The x-spacing (columns) of the input *elevation* grid.

dy

[number, optional] The y-spacing (rows) of the input *elevation* grid.

fraction

[number, optional] Increases or decreases the contrast of the hillshade. Values greater than one will cause intermediate values to move closer to full illumination or shadow (and clipping any values that move beyond 0 or 1). Note that this is not visually or mathematically the same as vertical exaggeration.

Returns

ndarray

A 2d array of illumination values between 0-1, where 0 is completely in shadow and 1 is completely illuminated.

`shade`(*self*, *data*, *cmap*, *norm=None*, *blend_mode='overlay'*, *vmin=None*, *vmax=None*, *vert_exag=1*, *dx=1*, *dy=1*, *fraction=1*, ***kwargs*)

Combine colormapped data values with an illumination intensity map (a.k.a. "hillshade") of the values.

Parameters

data

[array-like] A 2d array (or equivalent) of the height values used to generate a shaded map.

cmap

[*Colormap*] The colormap used to color the *data* array. Note that this must be a *Colormap* instance. For example, rather than passing in `cmap='gist_earth'`, use `cmap=plt.get_cmap('gist_earth')` instead.

norm

[*Normalize* instance, optional] The normalization used to scale values before colormapping. If None, the input will be linearly scaled between its min and max.

blend_mode

[{'hsv', 'overlay', 'soft'} or callable, optional] The type of blending used to combine the colormapped data values with the illumination intensity. Default is "overlay". Note that for most topographic surfaces, "overlay" or "soft" appear more visually realistic. If a user-defined function is supplied, it is expected to combine an MxNx3 RGB array of floats (ranging 0 to 1) with an MxNx1 hillshade array (also 0 to 1). (Call signature `func(rgb, illum, **kwargs)`) Additional kwargs supplied to this function will be passed on to the *blend_mode* function.

vmin

[float or None, optional] The minimum value used in colormapping *data*. If *None* the minimum value in *data* is used. If *norm* is specified, then this argument will be ignored.

vmax

[float or None, optional] The maximum value used in colormapping *data*. If *None* the maximum value in *data* is used. If *norm* is specified, then this argument will be ignored.

vert_exag

[number, optional] The amount to exaggerate the elevation values by when calculating illumination. This can be used either to correct for differences in units between the x-y coordinate system and the elevation coordinate system (e.g. decimal degrees vs. meters) or to exaggerate or de-emphasize topography.

dx

[number, optional] The x-spacing (columns) of the input *elevation* grid.

dy

[number, optional] The y-spacing (rows) of the input *elevation* grid.

fraction

[number, optional] Increases or decreases the contrast of the hillshade. Values greater than one will cause intermediate values to move closer to full illumination or shadow (and clipping any values that move beyond 0 or 1). Note that this is not visually or mathematically the same as vertical exaggeration.

Additional kwargs are passed on to the *blend_mode* function.

Returns

ndarray

An MxNx4 array of floats ranging between 0-1.

`shade_normals`(*self*, *normals*, *fraction=1.0*)

Calculate the illumination intensity for the normal vectors of a surface using the defined azimuth and elevation for the light source.

Imagine an artificial sun placed at infinity in some azimuth and elevation position illuminating our surface. The parts of the surface that slope toward the sun should brighten while those sides facing away should become darker.

Parameters

fraction

[number, optional] Increases or decreases the contrast of the hillshade. Values greater than one will cause intermediate values to move closer to full illumination or shadow (and clipping any values that move beyond 0 or 1). Note that this is not visually or mathematically the same as vertical exaggeration.

Returns

ndarray

A 2d array of illumination values between 0-1, where 0 is completely in shadow and 1 is completely illuminated.

`shade_rgb`(*self*, *rgb*, *elevation*, *fraction=1.0*, *blend_mode='hsv'*, *vert_exag=1*, *dx=1*, *dy=1*, ***kwargs*)

Use this light source to adjust the colors of the *rgb* input array to give the impression of a shaded relief map with the given *elevation*.

Parameters

rgb

[array-like] An (M, N, 3) RGB array, assumed to be in the range of 0 to 1.

elevation

[array-like] An (M, N) array of the height values used to generate a shaded map.

fraction

[number] Increases or decreases the contrast of the hillshade. Values greater than one will cause intermediate values to move closer to full illumination or shadow (and clipping any values that move beyond 0 or 1). Note that this is not visually or mathematically the same as vertical exaggeration.

blend_mode

[{'hsv', 'overlay', 'soft'} or callable, optional] The type of blending used to combine the colormapped data values with the illumination intensity. For backwards compatibility, this defaults to "hsv". Note that for most topographic surfaces, "overlay" or "soft" appear more visually realistic. If a user-defined function is supplied, it is expected to combine an MxNx3 RGB array of floats (ranging 0 to 1) with an MxNx1 hillshade array (also 0 to 1). (Call signature `func(rgb, illum, **kwargs)`) Additional kwargs supplied to this function will be passed on to the *blend_mode* function.

vert_exag

[number, optional] The amount to exaggerate the elevation values by when calculating illumination. This can be used either to correct for differences in units between the x-y coordinate system and the elevation coordinate system (e.g. decimal degrees vs. meters) or to exaggerate or de-emphasize topography.

dx

[number, optional] The x-spacing (columns) of the input *elevation* grid.

dy

[number, optional] The y-spacing (rows) of the input *elevation* grid.

Additional kwargs are passed on to the *blend_mode* function.

Returns

ndarray

An (m, n, 3) array of floats ranging between 0-1.

Examples using `matplotlib.colors.LightSource`

- sphx_glr_gallery_images_contours_and_fields_shading_example.py
- sphx_glr_gallery_showcase_mandelbrot.py
- sphx_glr_gallery_frontpage_3D.py
- sphx_glr_gallery_misc_demo_agg_filter.py
- sphx_glr_gallery_mplot3d_custom_shaded_3d_surface.py
- sphx_glr_gallery_specialty_plots_advanced_hillshading.py
- sphx_glr_gallery_specialty_plots_topographic_hillshading.py

matplotlib.colors.LinearSegmentedColormap

class matplotlib.colors.**LinearSegmentedColormap**(*name*, *segmentdata*, *N=256*, *gamma=1.0*)

Bases: *matplotlib.colors.Colormap*

Colormap objects based on lookup tables using linear segments.

The lookup table is generated using linear interpolation for each primary color, with the 0-1 domain divided into any number of segments.

Create color map from linear mapping segments

segmentdata argument is a dictionary with a red, green and blue entries. Each entry should be a list of *x*, *y0*, *y1* tuples, forming rows in a table. Entries for alpha are optional.

Example: suppose you want red to increase from 0 to 1 over the bottom half, green to do the same over the middle half, and blue over the top half. Then you would use:

```
cdict = {'red':   [(0.0,  0.0, 0.0),
                   (0.5,  1.0, 1.0),
                   (1.0,  1.0, 1.0)],

         'green': [(0.0,  0.0, 0.0),
                   (0.25, 0.0, 0.0),
                   (0.75, 1.0, 1.0),
                   (1.0,  1.0, 1.0)],

         'blue':  [(0.0,  0.0, 0.0),
                   (0.5,  0.0, 0.0),
                   (1.0,  1.0, 1.0)]}
```

Each row in the table for a given color is a sequence of *x*, *y0*, *y1* tuples. In each sequence, *x* must increase monotonically from 0 to 1. For any input value *z* falling between *x[i]* and *x[i+1]*, the output value of a given color will be linearly interpolated between *y1[i]* and *y0[i+1]*:

```
row i:   x  y0  y1
               /
              /
row i+1: x  y0  y1
```

Hence y0 in the first row and y1 in the last row are never used.

See also:

LinearSegmentedColormap.from_list

Static method; factory function for generating a smoothly-varying LinearSegmentedColormap.

makeMappingArray

For information about making a mapping array.

`__init__`(*self*, *name*, *segmentdata*, *N=256*, *gamma=1.0*)
Create color map from linear mapping segments

segmentdata argument is a dictionary with a red, green and blue entries. Each entry should be a list of *x*, *y0*, *y1* tuples, forming rows in a table. Entries for alpha are optional.

Example: suppose you want red to increase from 0 to 1 over the bottom half, green to do the same over the middle half, and blue over the top half. Then you would use:

```
cdict = {'red':   [(0.0,  0.0, 0.0),
                   (0.5,  1.0, 1.0),
                   (1.0,  1.0, 1.0)],

         'green': [(0.0,  0.0, 0.0),
                   (0.25, 0.0, 0.0),
                   (0.75, 1.0, 1.0),
                   (1.0,  1.0, 1.0)],

         'blue':  [(0.0,  0.0, 0.0),
                   (0.5,  0.0, 0.0),
                   (1.0,  1.0, 1.0)]}
```

Each row in the table for a given color is a sequence of *x*, *y0*, *y1* tuples. In each sequence, *x* must increase monotonically from 0 to 1. For any input value *z* falling between *x[i]* and *x[i+1]*, the output value of a given color will be linearly interpolated between *y1[i]* and *y0[i+1]*:

```
row i:   x  y0  y1
               /
              /
row i+1: x  y0  y1
```

Hence y0 in the first row and y1 in the last row are never used.

See also:

LinearSegmentedColormap.from_list

Static method; factory function for generating a smoothly-varying LinearSegmentedColormap.

makeMappingArray

For information about making a mapping array.

```
__module__ = 'matplotlib.colors'
```

`static from_list`(*name*, *colors*, *N=256*, *gamma=1.0*)

Create a *LinearSegmentedColormap* from a list of colors.

Parameters

name

[str] The name of the colormap.

colors

[array-like of colors or array-like of (value, color)] If only colors are given, they are equidistantly mapped from the range $[0, 1]$; i.e. 0 maps to `colors[0]` and 1 maps to `colors[-1]`. If (value, color) pairs are given, the mapping is from *value* to *color*. This can be used to divide the range unevenly.

N

[int] The number of rgb quantization levels.

gamma

[float]

`reversed`(*self*, *name=None*)

Return a reversed instance of the Colormap.

Parameters

name

[str, optional] The name for the reversed colormap. If it's None the name will be the name of the parent colormap + "_r".

Returns

LinearSegmentedColormap

The reversed colormap.

`set_gamma`(*self*, *gamma*)
Set a new gamma value and regenerate color map.

Examples using `matplotlib.colors.LinearSegmentedColormap`

- sphx_glr_gallery_lines_bars_and_markers_gradient_bar.py
- sphx_glr_gallery_lines_bars_and_markers_scatter_with_legend.py
- sphx_glr_gallery_images_contours_and_fields_contour_demo.py
- sphx_glr_gallery_images_contours_and_fields_contour_image.py
- sphx_glr_gallery_images_contours_and_fields_contourf_demo.py
- sphx_glr_gallery_images_contours_and_fields_contourf_log.py
- sphx_glr_gallery_images_contours_and_fields_image_demo.py
- sphx_glr_gallery_images_contours_and_fields_image_masked.py
- sphx_glr_gallery_images_contours_and_fields_image_nonuniform.py
- sphx_glr_gallery_images_contours_and_fields_layer_images.py
- sphx_glr_gallery_images_contours_and_fields_pcolormesh_levels.py
- sphx_glr_gallery_images_contours_and_fields_shading_example.py
- sphx_glr_gallery_images_contours_and_fields_tricontour_smooth_delaunay.py
- sphx_glr_gallery_images_contours_and_fields_tricontour_smooth_user.py
- sphx_glr_gallery_images_contours_and_fields_trigradient_demo.py
- sphx_glr_gallery_subplots_axes_and_figures_axes_box_aspect.py
- sphx_glr_gallery_subplots_axes_and_figures_subplots_adjust.py
- sphx_glr_gallery_text_labels_and_annotations_custom_legends.py
- sphx_glr_gallery_text_labels_and_annotations_demo_text_path.py
- sphx_glr_gallery_color_custom_cmap.py
- sphx_glr_gallery_shapes_and_collections_artist_reference.py
- sphx_glr_gallery_shapes_and_collections_dolphin.py
- sphx_glr_gallery_axisartist_demo_curvelinear_grid2.py
- sphx_glr_gallery_showcase_mandelbrot.py
- sphx_glr_gallery_frontpage_3D.py
- sphx_glr_gallery_misc_contour_manual.py
- sphx_glr_gallery_misc_demo_agg_filter.py
- sphx_glr_gallery_misc_logos2.py

- sphx_glr_gallery_misc_table_demo.py
- sphx_glr_gallery_mplot3d_contour3d.py
- sphx_glr_gallery_mplot3d_contour3d_2.py
- sphx_glr_gallery_mplot3d_contour3d_3.py
- sphx_glr_gallery_mplot3d_contourf3d.py
- sphx_glr_gallery_mplot3d_contourf3d_2.py
- sphx_glr_gallery_mplot3d_custom_shaded_3d_surface.py
- sphx_glr_gallery_mplot3d_subplot3d.py
- sphx_glr_gallery_mplot3d_surface3d.py
- sphx_glr_gallery_mplot3d_surface3d_radial.py
- sphx_glr_gallery_mplot3d_tricontour3d.py
- sphx_glr_gallery_mplot3d_tricontourf3d.py
- sphx_glr_gallery_mplot3d_trisurf3d_2.py
- sphx_glr_gallery_specialty_plots_advanced_hillshading.py
- sphx_glr_gallery_specialty_plots_leftventricle_bulleye.py
- sphx_glr_gallery_specialty_plots_mri_with_eeg.py
- sphx_glr_gallery_specialty_plots_topographic_hillshading.py
- sphx_glr_gallery_ticks_and_spines_colorbar_tick_labelling_demo.py
- sphx_glr_gallery_ticks_and_spines_spines_dropped.py
- *Customized Colorbars Tutorial*
- *Creating Colormaps in Matplotlib*
- *Colormap Normalization*

matplotlib.colors.ListedColormap

class matplotlib.colors.**ListedColormap**(*colors*, *name='from_list'*, *N=None*)

Bases: *matplotlib.colors.Colormap*

Colormap object generated from a list of colors.

This may be most useful when indexing directly into a colormap, but it can also be used to generate special colormaps for ordinary mapping.

Parameters

colors

[list, array] List of Matplotlib color specifications, or an equivalent Nx3 or Nx4 floating point array (*N* rgb or rgba values).

name

[str, optional] String to identify the colormap.

N

[int, optional] Number of entries in the map. The default is *None*, in which case there is one colormap entry for each element in the list of colors. If

```
N < len(colors)
```

the list will be truncated at *N*. If

```
N > len(colors)
```

the list will be extended by repetition.

Parameters

name

[str] The name of the colormap.

N

[int] The number of rgb quantization levels.

`__init__`(*self*, *colors*, *name='from_list'*, *N=None*)

Parameters

name

[str] The name of the colormap.

N

[int] The number of rgb quantization levels.

`__module__ = 'matplotlib.colors'`

`reversed`(*self*, *name=None*)

Return a reversed instance of the Colormap.

Parameters

name

[str, optional] The name for the reversed colormap. If it's None the name will be the name of the parent colormap + "_r".

Returns

ListedColormap

A reversed instance of the colormap.

Examples using `matplotlib.colors.ListedColormap`

- sphx_glr_gallery_lines_bars_and_markers_multicolored_line.py
- sphx_glr_gallery_images_contours_and_fields_layer_images.py
- sphx_glr_gallery_images_contours_and_fields_quadmesh_demo.py
- sphx_glr_gallery_statistics_hist.py
- sphx_glr_gallery_pie_and_polar_charts_nested_pie.py
- sphx_glr_gallery_pie_and_polar_charts_polar_bar.py
- sphx_glr_gallery_pyplots_whats_new_1_subplot3d.py
- sphx_glr_gallery_pyplots_whats_new_99_mplot3d.py
- sphx_glr_gallery_specialty_plots_leftventricle_bulleye.py
- *Customized Colorbars Tutorial*
- *Creating Colormaps in Matplotlib*

matplotlib.colors.LogNorm

`class matplotlib.colors.LogNorm(vmin=None, vmax=None, clip=False)`

Bases: *matplotlib.colors.Normalize*

Normalize a given value to the 0-1 range on a log scale.

Parameters

vmin, vmax

[float or None] If *vmin* and/or *vmax* is not given, they are initialized from the minimum and maximum value, respectively, of the first input processed; i.e., `__call__(A)` calls `autoscale_None(A)`.

clip

[bool, default: False] If `True` values falling outside the range `[vmin, vmax]`, are mapped to 0 or 1, whichever is closer, and masked values are set to 1. If `False` masked values remain masked.

Clipping silently defeats the purpose of setting the over, under, and masked colors in a colormap, so it is likely to lead to surprises; therefore the default is `clip=False`.

Notes

Returns 0 if `vmin == vmax`.

`__call__`(*self*, *value*, *clip=None*)
Normalize *value* data in the `[vmin, vmax]` interval into the `[0.0, 1.0]` interval and return it.

Parameters

value

Data to normalize.

clip

[bool] If `None`, defaults to `self.clip` (which defaults to `False`).

Notes

If not already initialized, `self.vmin` and `self.vmax` are initialized using `self.autoscale_None(value)`.

`__module__ = 'matplotlib.colors'`

`__slotnames__ = []`

`autoscale`(*self*, *A*)
Set *vmin*, *vmax* to min, max of *A*.

`autoscale_None`(*self*, *A*)
If vmin or vmax are not set, use the min/max of *A* to set them.

`inverse`(*self*, *value*)

Examples using `matplotlib.colors.LogNorm`

- sphx_glr_gallery_images_contours_and_fields_pcolor_demo.py
- sphx_glr_gallery_statistics_hist.py
- sphx_glr_gallery_userdemo_colormap_normalizations.py
- *Colormap Normalization*

matplotlib.colors.NoNorm

class matplotlib.colors.**NoNorm**(*vmin=None*, *vmax=None*, *clip=False*)

Bases: *matplotlib.colors.Normalize*

Dummy replacement for *Normalize*, for the case where we want to use indices directly in a *ScalarMappable*.

Parameters

vmin, vmax

[float or None] If *vmin* and/or *vmax* is not given, they are initialized from the minimum and maximum value, respectively, of the first input processed; i.e., `__call__(A)` calls `autoscale_None(A)`.

clip

[bool, default: False] If `True` values falling outside the range `[vmin, vmax]`, are mapped to 0 or 1, whichever is closer, and masked values are set to 1. If `False` masked values remain masked.

Clipping silently defeats the purpose of setting the over, under, and masked colors in a colormap, so it is likely to lead to surprises; therefore the default is `clip=False`.

Notes

Returns 0 if `vmin == vmax`.

`__call__`(*self*, *value*, *clip=None*)

Normalize *value* data in the `[vmin, vmax]` interval into the `[0.0, 1.0]` interval and return it.

Parameters

value

Data to normalize.

clip

[bool] If `None`, defaults to `self.clip` (which defaults to `False`).

Notes

If not already initialized, `self.vmin` and `self.vmax` are initialized using `self.autoscale_None(value)`.

`__module__ = 'matplotlib.colors'`

`__slotnames__ = []`

`inverse`(*self*, *value*)

Examples using `matplotlib.colors.NoNorm`

matplotlib.colors.Normalize

`class matplotlib.colors.Normalize`(*vmin=None*, *vmax=None*, *clip=False*)

Bases: `object`

A class which, when called, linearly normalizes data into the `[0.0, 1.0]` interval.

Parameters

vmin, vmax

[float or None] If *vmin* and/or *vmax* is not given, they are initialized from the minimum and maximum value, respectively, of the first input processed; i.e., `__call__(A)` calls `autoscale_None(A)`.

clip

[bool, default: False] If `True` values falling outside the range `[vmin, vmax]`, are mapped to 0 or 1, whichever is closer, and masked values are set to 1. If `False` masked values remain masked.

Clipping silently defeats the purpose of setting the over, under, and masked colors in a colormap, so it is likely to lead to surprises; therefore the default is `clip=False`.

Notes

Returns 0 if `vmin == vmax`.

`__call__`(*self*, *value*, *clip=None*)

Normalize *value* data in the `[vmin, vmax]` interval into the `[0.0, 1.0]` interval and return it.

Parameters

value

Data to normalize.

clip

[bool] If `None`, defaults to `self.clip` (which defaults to `False`).

Notes

If not already initialized, `self.vmin` and `self.vmax` are initialized using `self.autoscale_None(value)`.

`__dict__ = mappingproxy({'__module__': 'matplotlib.colors', '__doc__': '\n A class which,`

`__init__`(*self*, *vmin=None*, *vmax=None*, *clip=False*)

Parameters

vmin, vmax

[float or None] If *vmin* and/or *vmax* is not given, they are initialized from the minimum and maximum value, respectively, of the first input processed; i.e., `__call__(A)` calls `autoscale_None(A)`.

clip

[bool, default: False] If `True` values falling outside the range `[vmin, vmax]`, are mapped to 0 or 1, whichever is closer, and masked values are set to 1. If `False` masked values remain masked.

Clipping silently defeats the purpose of setting the over, under, and masked colors in a colormap, so it is likely to lead to surprises; therefore the default is `clip=False`.

Notes

Returns 0 if `vmin == vmax`.

`__module__ = 'matplotlib.colors'`

`__slotnames__ = []`

`__weakref__`
list of weak references to the object (if defined)

`autoscale`(*self*, *A*)
Set *vmin*, *vmax* to min, max of *A*.

`autoscale_None`(*self*, *A*)
If vmin or vmax are not set, use the min/max of *A* to set them.

`inverse`(*self*, *value*)

`static process_value`(*value*)

Homogenize the input *value* for easy and efficient normalization.

value can be a scalar or sequence.

Returns

result

[masked array] Masked array with the same shape as *value*.

is_scalar

[bool] Whether *value* is a scalar.

Notes

Float dtypes are preserved; integer types with two bytes or smaller are converted to np.float32, and larger types are converted to np.float64. Preserving float32 when possible, and using in-place operations, greatly improves speed for large arrays.

`scaled`(*self*)

Return whether vmin and vmax are set.

Examples using `matplotlib.colors.Normalize`

- sphx_glr_gallery_lines_bars_and_markers_multicolored_line.py
- sphx_glr_gallery_images_contours_and_fields_contour_image.py
- sphx_glr_gallery_images_contours_and_fields_image_annotated_heatmap.py
- sphx_glr_gallery_images_contours_and_fields_image_masked.py
- sphx_glr_gallery_images_contours_and_fields_image_transparency_blend.py
- sphx_glr_gallery_images_contours_and_fields_multi_image.py
- sphx_glr_gallery_images_contours_and_fields_pcolor_demo.py
- sphx_glr_gallery_images_contours_and_fields_pcolormesh_levels.py
- sphx_glr_gallery_statistics_hist.py
- sphx_glr_gallery_axes_grid1_demo_axes_grid2.py
- sphx_glr_gallery_showcase_mandelbrot.py
- sphx_glr_gallery_frontpage_contour.py
- sphx_glr_gallery_scales_power_norm.py

- sphx_glr_gallery_specialty_plots_advanced_hillshading.py
- sphx_glr_gallery_specialty_plots_leftventricle_bulleye.py
- sphx_glr_gallery_userdemo_colormap_normalizations.py
- sphx_glr_gallery_userdemo_colormap_normalizations_symlognorm.py
- *Constrained Layout Guide*
- *Customized Colorbars Tutorial*
- *Colormap Normalization*

matplotlib.colors.PowerNorm

class matplotlib.colors.**PowerNorm**(*gamma*, *vmin=None*, *vmax=None*, *clip=False*)

Bases: *matplotlib.colors.Normalize*

Linearly map a given value to the 0-1 range and then apply a power-law normalization over that range.

Parameters

vmin, vmax

[float or None] If *vmin* and/or *vmax* is not given, they are initialized from the minimum and maximum value, respectively, of the first input processed; i.e., `__call__(A)` calls `autoscale_None(A)`.

clip

[bool, default: False] If `True` values falling outside the range `[vmin, vmax]`, are mapped to 0 or 1, whichever is closer, and masked values are set to 1. If `False` masked values remain masked.

Clipping silently defeats the purpose of setting the over, under, and masked colors in a colormap, so it is likely to lead to surprises; therefore the default is `clip=False`.

Notes

Returns 0 if `vmin == vmax`.

`__call__`(*self*, *value*, *clip=None*)

Normalize *value* data in the `[vmin, vmax]` interval into the `[0.0, 1.0]` interval and return it.

Parameters

value

Data to normalize.

clip

[bool] If `None`, defaults to `self.clip` (which defaults to `False`).

Notes

If not already initialized, `self.vmin` and `self.vmax` are initialized using `self.autoscale_None(value)`.

`__init__`(*self*, *gamma*, *vmin=None*, *vmax=None*, *clip=False*)

Parameters

vmin, vmax

[float or None] If *vmin* and/or *vmax* is not given, they are initialized from the minimum and maximum value, respectively, of the first input processed; i.e., `__call__(A)` calls `autoscale_None(A)`.

clip

[bool, default: False] If `True` values falling outside the range `[vmin, vmax]`, are mapped to 0 or 1, whichever is closer, and masked values are set to 1. If `False` masked values remain masked.

Clipping silently defeats the purpose of setting the over, under, and masked colors in a colormap, so it is likely to lead to surprises; therefore the default is `clip=False`.

Notes

Returns 0 if `vmin == vmax`.

`__module__ = 'matplotlib.colors'`

`__slotnames__ = []`

`inverse`(*self*, *value*)

Examples using `matplotlib.colors.PowerNorm`

- sphx_glr_gallery_showcase_mandelbrot.py
- sphx_glr_gallery_scales_power_norm.py
- sphx_glr_gallery_userdemo_colormap_normalizations.py
- *Colormap Normalization*

matplotlib.colors.SymLogNorm

class matplotlib.colors.**SymLogNorm**(*linthresh*, *linscale=1.0*, *vmin=None*, *vmax=None*, *clip=False*, *, *base=None*)

Bases: *matplotlib.colors.Normalize*

The symmetrical logarithmic scale is logarithmic in both the positive and negative directions from the origin.

Since the values close to zero tend toward infinity, there is a need to have a range around zero that is linear. The parameter *linthresh* allows the user to specify the size of this range (*-linthresh*, *linthresh*).

Parameters

linthresh

[float] The range within which the plot is linear (to avoid having the plot go to infinity around zero).

linscale

[float, default: 1] This allows the linear range (*-linthresh* to *linthresh*) to be stretched relative to the logarithmic range. Its value is the number of powers of *base* to use for each half of the linear range.

For example, when *linscale* == 1.0 (the default) and `base=10`, then space used for the positive and negative halves of the linear range will be equal to a decade in the logarithmic.

base

[float, default: None] If not given, defaults to `np.e` (consistent with prior behavior) and warns.

In v3.3 the default value will change to 10 to be consistent with *SymLogNorm*.

To suppress the warning pass *base* as a keyword argument.

`__call__`(*self*, *value*, *clip=None*)

Normalize *value* data in the `[vmin, vmax]` interval into the `[0.0, 1.0]` interval and return it.

Parameters

value

Data to normalize.

clip

[bool] If `None`, defaults to `self.clip` (which defaults to `False`).

Notes

If not already initialized, `self.vmin` and `self.vmax` are initialized using `self.autoscale_None(value)`.

`__init__`(*self*, *linthresh*, *linscale=1.0*, *vmin=None*, *vmax=None*, *clip=False*, **, base=None*)

Parameters

linthresh

[float] The range within which the plot is linear (to avoid having the plot go to infinity around zero).

linscale

[float, default: 1] This allows the linear range (*-linthresh* to *linthresh*) to be stretched relative to the logarithmic range. Its value is the number of powers of *base* to use for each half of the linear range.

For example, when *linscale* == 1.0 (the default) and `base=10`, then space used for the positive and negative halves of the linear range will be equal to a decade in the logarithmic.

base

[float, default: None] If not given, defaults to `np.e` (consistent with prior behavior) and warns.

In v3.3 the default value will change to 10 to be consistent with *SymLogNorm*.

To suppress the warning pass *base* as a keyword argument.

`__module__ = 'matplotlib.colors'`

`__slotnames__ = []`

`autoscale`(*self*, *A*)
Set *vmin*, *vmax* to min, max of *A*.

`autoscale_None`(*self*, *A*)
If vmin or vmax are not set, use the min/max of *A* to set them.

`inverse`(*self*, *value*)

Examples using `matplotlib.colors.SymLogNorm`

- sphx_glr_gallery_userdemo_colormap_normalizations.py
- sphx_glr_gallery_userdemo_colormap_normalizations_symlognorm.py
- *Colormap Normalization*

matplotlib.colors.TwoSlopeNorm

`class matplotlib.colors.TwoSlopeNorm`(*vcenter*, *vmin=None*, *vmax=None*)
Bases: *matplotlib.colors.Normalize*

Normalize data with a set center.

Useful when mapping data with an unequal rates of change around a conceptual center, e.g., data that range from -2 to 4, with 0 as the midpoint.

Parameters

vcenter

[float] The data value that defines `0.5` in the normalization.

vmin

[float, optional] The data value that defines `0.0` in the normalization. Defaults to the min value of the dataset.

vmax

[float, optional] The data value that defines `1.0` in the normalization. Defaults to the the max value of the dataset.

Examples

This maps data value -4000 to 0., 0 to 0.5, and +10000 to 1.0; data between is linearly interpolated:

```
>>> import matplotlib.colors as mcolors
>>> offset = mcolors.TwoSlopeNorm(vmin=-4000.,
                                  vcenter=0., vmax=10000)
>>> data = [-4000., -2000., 0., 2500., 5000., 7500., 10000.]
>>> offset(data)
array([0., 0.25, 0.5, 0.625, 0.75, 0.875, 1.0])
```

__call__(*self*, *value*, *clip=None*)
Map value to the interval [0, 1]. The clip argument is unused.

__init__(*self*, *vcenter*, *vmin=None*, *vmax=None*)
Normalize data with a set center.

Useful when mapping data with an unequal rates of change around a conceptual center, e.g., data that range from -2 to 4, with 0 as the midpoint.

Parameters

vcenter

[float] The data value that defines `0.5` in the normalization.

vmin

[float, optional] The data value that defines `0.0` in the normalization. Defaults to the min value of the dataset.

vmax

[float, optional] The data value that defines `1.0` in the normalization. Defaults to the the max value of the dataset.

Examples

This maps data value -4000 to 0., 0 to 0.5, and +10000 to 1.0; data between is linearly interpolated:

```
>>> import matplotlib.colors as mcolors
>>> offset = mcolors.TwoSlopeNorm(vmin=-4000.,
                                  vcenter=0., vmax=10000)
>>> data = [-4000., -2000., 0., 2500., 5000., 7500., 10000.]
>>> offset(data)
array([0., 0.25, 0.5, 0.625, 0.75, 0.875, 1.0])
```

__module__ = 'matplotlib.colors'

__slotnames__ = []

autoscale_None(*self*, *A*)
Get vmin and vmax, and then clip at vcenter

Examples using `matplotlib.colors.TwoSlopeNorm`

- *Colormap Normalization*

18.18.2 Functions

from_levels_and_colors(levels, colors[, extend])	A helper routine to generate a cmap and a norm instance which behave similar to contourf's levels and colors arguments.
hsv_to_rgb(hsv)	Convert hsv values to rgb.
rgb_to_hsv(arr)	Convert float rgb values (in the range [0, 1]), in a numpy array to hsv values.
to_hex(c[, keep_alpha])	Convert *c* to a hex color.
to_rgb(c)	Convert *c* to an RGB color, silently dropping the alpha channel.
to_rgba(c[, alpha])	Convert *c* to an RGBA color.
to_rgba_array(c[, alpha])	Convert *c* to a (n, 4) array of RGBA colors.
is_color_like(c)	Return whether *c* can be interpreted as an RGB(A) color.
same_color(c1, c2)	Return whether the colors *c1* and *c2* are the same.
makeMappingArray(N, data[, gamma])	[*Deprecated*] Create an *N* -element 1-d lookup table.
get_named_colors_mapping()	Return the global mapping of names to named colors.

matplotlib.colors.from_levels_and_colors

`matplotlib.colors.`**`from_levels_and_colors`**(*levels*, *colors*, *extend='neither'*)

A helper routine to generate a cmap and a norm instance which behave similar to contourf's levels and colors arguments.

Parameters

levels

[sequence of numbers] The quantization levels used to construct the *BoundaryNorm*. Value `v` is quantized to level `i` if `lev[i] <= v < lev[i+1]`.

colors

[sequence of colors] The fill color to use for each level. If *extend* is "neither" there must be `n_level - 1` colors. For an *extend* of

"min" or "max" add one extra color, and for an *extend* of "both" add two colors.

extend

[{'neither', 'min', 'max', 'both'}, optional] The behaviour when a value falls out of range of the given levels. See *contourf* for details.

Returns

cmap

[*Normalize*]

norm

[*Colormap*]

Examples using `matplotlib.colors.from_levels_and_colors`

matplotlib.colors.hsv_to_rgb

`matplotlib.colors.hsv_to_rgb(hsv)`

Convert hsv values to rgb.

Parameters

hsv

[(..., 3) array-like] All values assumed to be in range [0, 1]

Returns

(..., 3) ndarray

Colors converted to RGB values in range [0, 1]

Examples using `matplotlib.colors.hsv_to_rgb`

- sphx_glr_gallery_mplot3d_voxels_torus.py

matplotlib.colors.rgb_to_hsv

matplotlib.colors.**rgb_to_hsv**(*arr*)

Convert float rgb values (in the range [0, 1]), in a numpy array to hsv values.

Parameters

arr

[(..., 3) array-like] All values must be in the range [0, 1]

Returns

(..., 3) ndarray

Colors converted to hsv values in range [0, 1]

Examples using `matplotlib.colors.rgb_to_hsv`

- sphx_glr_gallery_color_named_colors.py

matplotlib.colors.to_hex

matplotlib.colors.**to_hex**(*c*, *keep_alpha=False*)

Convert *c* to a hex color.

Uses the `#rrggbb` format if *keep_alpha* is False (the default), `#rrggbbaa` otherwise.

Examples using `matplotlib.colors.to_hex`

matplotlib.colors.to_rgb

matplotlib.colors.**to_rgb**(*c*)

Convert *c* to an RGB color, silently dropping the alpha channel.

Examples using `matplotlib.colors.to_rgb`

- sphx_glr_gallery_color_named_colors.py

matplotlib.colors.to_rgba

matplotlib.colors.to_rgba(*c*, *alpha=None*)

Convert *c* to an RGBA color.

Parameters

c

[Matplotlib color or `np.ma.masked`]

alpha

[float, optional] If *alpha* is not `None`, it forces the alpha value, except if *c* is `"none"` (case-insensitive), which always maps to `(0, 0, 0, 0)`.

Returns

tuple

Tuple of `(r, g, b, a)` scalars.

Examples using `matplotlib.colors.to_rgba`

- sphx_glr_gallery_color_named_colors.py
- sphx_glr_gallery_shapes_and_collections_collections.py
- sphx_glr_gallery_shapes_and_collections_line_collection.py
- sphx_glr_gallery_event_handling_lasso_demo.py
- sphx_glr_gallery_misc_demo_ribbon_box.py
- sphx_glr_gallery_widgets_menu.py

matplotlib.colors.to_rgba_array

matplotlib.colors.to_rgba_array(*c*, *alpha=None*)

Convert *c* to a (n, 4) array of RGBA colors.

If *alpha* is not `None`, it forces the alpha value. If *c* is `"none"` (case-insensitive) or an empty list, an empty array is returned. If *c* is a masked array, an ndarray is returned with a (0, 0, 0, 0) row for each masked value or row in *c*.

Examples using `matplotlib.colors.to_rgba_array`

matplotlib.colors.is_color_like

`matplotlib.colors.is_color_like(c)`
Return whether *c* can be interpreted as an RGB(A) color.

Examples using `matplotlib.colors.is_color_like`

matplotlib.colors.same_color

`matplotlib.colors.same_color(c1, c2)`
Return whether the colors *c1* and *c2* are the same.

c1, *c2* can be single colors or lists/arrays of colors.

Examples using `matplotlib.colors.same_color`

matplotlib.colors.makeMappingArray

`matplotlib.colors.makeMappingArray(N, data, gamma=1.0)`
[*Deprecated*] Create an *N* -element 1-d lookup table.

This assumes a mapping $f : [0, 1] \to [0, 1]$. The returned data is an array of N values $y = f(x)$ where x is sampled from [0, 1].

By default (*gamma* = 1) x is equidistantly sampled from [0, 1]. The *gamma* correction factor γ distorts this equidistant sampling by $x \to x^\gamma$.

Parameters

N

[int] The number of elements of the created lookup table; at least 1.

data

[Mx3 array-like or callable] Defines the mapping f.

If a Mx3 array-like, the rows define values (x, y0, y1). The x values must start with x=0, end with x=1, and all x values be in increasing order.

A value between x_i and x_{i+1} is mapped to the range $y^1_{i-1} \ldots y^0_i$ by linear interpolation.

For the simple case of a y-continuous mapping, y0 and y1 are identical.

The two values of y are to allow for discontinuous mapping functions. E.g. a sawtooth with a period of 0.2 and an amplitude of 1 would be:

```
[(0, 1, 0), (0.2, 1, 0), (0.4, 1, 0), ..., [(1, 1, 0)]
```

In the special case of `N == 1`, by convention the returned value is y0 for x == 1.

If *data* is a callable, it must accept and return numpy arrays:

```
data(x : ndarray) -> ndarray
```

and map values between 0 - 1 to 0 - 1.

gamma

[float] Gamma correction factor for input distribution x of the mapping.

See also https://en.wikipedia.org/wiki/Gamma_correction.

Returns

array

The lookup table where `lut[x * (N-1)]` gives the closest value for values of x between 0 and 1.

Notes

This function is internally used for *LinearSegmentedColormap*.

Deprecated since version 3.2.

Examples using `matplotlib.colors.makeMappingArray`

matplotlib.colors.get_named_colors_mapping

matplotlib.colors.**get_named_colors_mapping**()

Return the global mapping of names to named colors.

Examples using `matplotlib.colors.get_named_colors_mapping`

18.19 matplotlib.container

class matplotlib.container.**BarContainer**(**args*, ***kwargs*)

Bases: *matplotlib.container.Container*

Container for the artists of bar plots (e.g. created by *Axes.bar*).

The container can be treated as a tuple of the *patches* themselves. Additionally, you can access these and further parameters by the attributes.

Attributes

patches

[list of *Rectangle*] The artists of the bars.

errorbar

[None or *ErrorbarContainer*] A container for the error bar artists if error bars are present. *None* otherwise.

class matplotlib.container.**Container**(**args*, ***kwargs*)

Bases: `tuple`

Base class for containers.

Containers are classes that collect semantically related Artists such as the bars of a bar plot.

add_callback(*self*, *func*)

Add a callback function that will be called whenever one of the *Artist*'s properties changes.

Parameters

func

[callable] The callback function. It must have the signature:

```
def func(artist: Artist) -> Any
```

where *artist* is the calling *Artist*. Return values may exist but are ignored.

Returns

int

The observer id associated with the callback. This id can be used for removing the callback with *remove_callback* later.

See also:

remove_callback

get_children(*self*)

get_label(*self*)
Return the label used for this artist in the legend.

pchanged(*self*)
Call all of the registered callbacks.

This function is triggered internally when a property is changed.

See also:

add_callback

remove_callback

remove(*self*)

remove_callback(*self*, *oid*)
Remove a callback based on its observer id.

See also:

add_callback

set_label(*self*, *s*)
Set a label that will be displayed in the legend.

Parameters

s

[object] *s* will be converted to a string by calling `str`.

class matplotlib.container.**ErrorbarContainer**(**args*, ***kwargs*)
Bases: *matplotlib.container.Container*

Container for the artists of error bars (e.g. created by *Axes.errorbar*).

The container can be treated as the *lines* tuple itself. Additionally, you can access these and further parameters by the attributes.

Attributes

lines

[tuple] Tuple of `(data_line, caplines, barlinecols)`.

- data_line : *Line2D* instance of x, y plot markers and/or line.
- caplines : tuple of *Line2D* instances of the error bar caps.

- barlinecols : list of *LineCollection* with the horizontal and vertical error ranges.

has_xerr, has_yerr

[bool] `True` if the errorbar has x/y errors.

class matplotlib.container.**StemContainer**(**args*, ***kwargs*)

Bases: *matplotlib.container.Container*

Container for the artists created in a *Axes.stem()* plot.

The container can be treated like a namedtuple `(markerline, stemlines, baseline)`.

Attributes

markerline

[*Line2D*] The artist of the markers at the stem heads.

stemlines

[list of *Line2D*] The artists of the vertical lines for all stems.

baseline

[*Line2D*] The artist of the horizontal baseline.

Parameters

markerline_stemlines_baseline

[tuple] Tuple of `(markerline, stemlines, baseline)`. `markerline` contains the *LineCollection* of the markers, `stemlines` is a *LineCollection* of the main lines, `baseline` is the *Line2D* of the baseline.

18.20 matplotlib.contour

Classes to support contour plotting and labelling for the Axes class.

class matplotlib.contour.**ClabelText**(*x=0*, *y=0*, *text=''*, *color=None*, *verticalalignment='baseline'*, *horizontalalignment='left'*, *multialignment=None*, *fontproperties=None*, *rotation=None*, *linespacing=None*, *rotation_mode=None*, *usetex=None*, *wrap=False*, ***kwargs*)

Bases: *matplotlib.text.Text*

Unlike the ordinary text, the get_rotation returns an updated angle in the pixel coordinate assuming that the input rotation is an angle in data coordinate (or whatever transform set).

Create a *Text* instance at *x*, *y* with string *text*.

Valid keyword arguments are:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array an
alpha	float or None
animated	bool
backgroundcolor	color
bbox	dict with properties for *patches.FancyBboxPatch*
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color or c	color
contains	unknown
figure	*Figure*
fontfamily or family	{FONTNAME, 'serif', 'sans-serif', 'cursive', 'fantasy', '
fontproperties or font or font_properties	*font_manager.FontProperties* or `str` or `pathlib.Path`
fontsize or size	float or {'xx-small', 'x-small', 'small', 'medium', 'large',
fontstretch or stretch	{a numeric value in range 0-1000, 'ultra-condensed',
fontstyle or style	{'normal', 'italic', 'oblique'}
fontvariant or variant	{'normal', 'small-caps'}
fontweight or weight	{a numeric value in range 0-1000, 'ultralight', 'light',
gid	str
horizontalalignment or ha	{'center', 'right', 'left'}
in_layout	bool
label	object
linespacing	float (multiple of font size)
multialignment or ma	{'left', 'right', 'center'}
path_effects	*AbstractPathEffect*
picker	None or bool or callable
position	(float, float)
rasterized	bool or None
rotation	float or {'vertical', 'horizontal'}
rotation_mode	{None, 'default', 'anchor'}
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
text	object
transform	*Transform*
url	str
usetex	bool or None
verticalalignment or va	{'center', 'top', 'bottom', 'baseline', 'center_baseline'}

Tabl

Property	Description
visible	bool
wrap	bool
x	float
y	float
zorder	float

get_rotation(*self*)
Return the text angle in degrees between 0 and 360.

class matplotlib.contour.ContourLabeler
Bases: object

Mixin to provide labelling capability to *ContourSet*.

add_label(*self*, *x*, *y*, *rotation*, *lev*, *cvalue*)
Add contour label using *Text* class.

add_label_clabeltext(*self*, *x*, *y*, *rotation*, *lev*, *cvalue*)
Add contour label using *ClabelText* class.

add_label_near(*self*, *x*, *y*, *inline=True*, *inline_spacing=5*, *transform=None*)
Add a label near the point (x, y). If transform is None (default), (x, y) is in data coordinates; if transform is False, (x, y) is in display coordinates; otherwise, the specified transform will be used to translate (x, y) into display coordinates.

Parameters

x, y

[float] The approximate location of the label.

inline

[bool, default: True] If *True* remove the segment of the contour beneath the label.

inline_spacing

[int, default: 5] Space in pixels to leave on each side of label when placing inline. This spacing will be exact for labels at locations where the contour is straight, less so for labels on curved contours.

calc_label_rot_and_inline(*self*, *slc*, *ind*, *lw*, *lc=None*, *spacing=5*)
Calculate the appropriate label rotation given the linecontour coordinates in screen units, the index of the label location and the label width.

If *lc* is not None or empty, also break contours and compute inlining.

spacing is the empty space to leave around the label, in pixels.

Both tasks are done together to avoid calculating path lengths multiple times, which is relatively costly.

The method used here involves computing the path length along the contour in pixel coordinates and then looking approximately (label width / 2) away from central point to determine rotation and then to break contour if desired.

`clabel`(*self*, *levels=None*, ***, *fontsize=None*, *inline=True*, *inline_spacing=5*, *fmt='%1.3f'*, *colors=None*, *use_clabeltext=False*, *manual=False*, *rightside_up=True*, *zorder=None*)

Label a contour plot.

Adds labels to line contours in this *ContourSet* (which inherits from this mixin class).

Parameters

levels

[array-like, optional] A list of level values, that should be labeled. The list must be a subset of `cs.levels`. If not given, all levels are labeled.

fontsize

[str or float, default: `rcParams["font.size"]` (default: `10.0`)] Size in points or relative size e.g., 'smaller', 'x-large'. See *Text.set_size* for accepted string values.

colors

[color or colors or None, default: None] The label colors:

- If *None*, the color of each label matches the color of the corresponding contour.
- If one string color, e.g., *colors* = 'r' or *colors* = 'red', all labels will be plotted in this color.
- If a tuple of colors (string, float, rgb, etc), different labels will be plotted in different colors in the order specified.

inline

[bool, default: True] If `True` the underlying contour is removed where the label is placed.

inline_spacing

[float, default: 5] Space in pixels to leave on each side of label when placing inline.

This spacing will be exact for labels at locations where the contour is straight, less so for labels on curved contours.

fmt

[str or dict, default: '%1.3f'] A format string for the label.

Alternatively, this can be a dictionary matching contour levels with arbitrary strings to use for each contour level (i.e., fmt[level]=string), or it can be any callable, such as a *Formatter* instance, that returns a string when called with a numeric contour level.

manual

[bool or iterable, default: False] If `True`, contour labels will be placed manually using mouse clicks. Click the first button near a contour to add a label, click the second button (or potentially both mouse buttons at once) to finish adding labels. The third button can be used to remove the last label added, but only if labels are not inline. Alternatively, the keyboard can be used to select label locations (enter to end label placement, delete or backspace act like the third mouse button, and any other key will select a label location).

manual can also be an iterable object of (x, y) tuples. Contour labels will be created as if mouse is clicked at each (x, y) position.

rightside_up

[bool, default: True] If `True`, label rotations will always be plus or minus 90 degrees from level.

use_clabeltext

[bool, default: False] If `True`, *ClabelText* class (instead of *Text*) is used to create labels. *ClabelText* recalculates rotation angles of texts during the drawing time, therefore this can be used if aspect of the axes changes.

zorder

[float or None, default: `(2 + contour.get_zorder())`] zorder of the contour labels.

Returns

labels

A list of *Text* instances for the labels.

`get_label_coords`(*self*, *distances*, *XX*, *YY*, *ysize*, *lw*)

Return x, y, and the index of a label location.

Labels are plotted at a location with the smallest deviation of the contour from a straight line unless there is another label nearby, in which case the

next best place on the contour is picked up. If all such candidates are rejected, the beginning of the contour is chosen.

`get_label_width`(*self*, *lev*, *fmt*, *fsize*)
Return the width of the label in points.

`get_text`(*self*, *lev*, *fmt*)
Get the text of the label.

`labels`(*self*, *inline*, *inline_spacing*)

`locate_label`(*self*, *linecontour*, *labelwidth*)
Find good place to draw a label (relatively flat part of the contour).

`pop_label`(*self*, *index=- 1*)
Defaults to removing last label, but any index can be supplied

`print_label`(*self*, *linecontour*, *labelwidth*)
Return whether a contour is long enough to hold a label.

`set_label_props`(*self*, *label*, *text*, *color*)
Set the label properties - color, fontsize, text.

`too_close`(*self*, *x*, *y*, *lw*)
Return *True* if a label is already near this location.

class matplotlib.contour.**ContourSet**(*ax*, **args*, *levels=None*, *filled=False*, *linewidths=None*, *linestyles=None*, *hatches=None*, *alpha=None*, *origin=None*, *extent=None*, *cmap=None*, *colors=None*, *norm=None*, *vmin=None*, *vmax=None*, *extend='neither'*, *antialiased=None*, *nchunk=0*, *locator=None*, *transform=None*, ***kwargs*)

Bases: *matplotlib.cm.ScalarMappable*, *matplotlib.contour.ContourLabeler*

Store a set of contour lines or filled regions.

User-callable method: *clabel*

Parameters

ax

[*Axes*]

levels

[[level0, level1, ..., leveln]] A list of floating point numbers indicating the contour levels.

allsegs

[[level0segs, level1segs, ...]] List of all the polygon segments for all the *levels*. For contour lines `len(allsegs) == len(levels)`, and

for filled contour regions `len(allsegs) = len(levels)-1`. The lists should look like

```
level0segs = [polygon0, polygon1, ...]
polygon0 = [[x0, y0], [x1, y1], ...]
```

allkinds

[`None` or [level0kinds, level1kinds, ...]] Optional list of all the polygon vertex kinds (code types), as described and used in Path. This is used to allow multiply- connected paths such as holes within filled polygons. If not `None`, `len(allkinds) == len(allsegs)`. The lists should look like

```
level0kinds = [polygon0kinds, ...]
polygon0kinds = [vertexcode0, vertexcode1, ...]
```

If *allkinds* is not `None`, usually all polygons for a particular contour level are grouped together so that `level0segs = [polygon0]` and `level0kinds = [polygon0kinds]`.

****kwargs**

Keyword arguments are as described in the docstring of *contour*.

Attributes

ax

The axes object in which the contours are drawn.

collections

A silent_list of LineCollections or PolyCollections.

levels

Contour levels.

layers

Same as levels for line contours; half-way between levels for filled contours. See `_process_colors()`.

Draw contour lines or filled regions, depending on whether keyword arg *filled* is `False` (default) or `True`.

Call signature:

```
ContourSet(ax, levels, allsegs, [allkinds], **kwargs)
```

Parameters

ax

[*Axes*] The *Axes* object to draw on.

levels

[[level0, level1, ..., leveln]] A list of floating point numbers indicating the contour levels.

allsegs

[[level0segs, level1segs, ...]] List of all the polygon segments for all the *levels*. For contour lines `len(allsegs) == len(levels)`, and for filled contour regions `len(allsegs) = len(levels)-1`. The lists should look like

```
level0segs = [polygon0, polygon1, ...]
polygon0 = [[x0, y0], [x1, y1], ...]
```

allkinds

[[level0kinds, level1kinds, ...], optional] Optional list of all the polygon vertex kinds (code types), as described and used in Path. This is used to allow multiply- connected paths such as holes within filled polygons. If not `None`, `len(allkinds) == len(allsegs)`. The lists should look like

```
level0kinds = [polygon0kinds, ...]
polygon0kinds = [vertexcode0, vertexcode1, ...]
```

If *allkinds* is not `None`, usually all polygons for a particular contour level are grouped together so that `level0segs = [polygon0]` and `level0kinds = [polygon0kinds]`.

****kwargs**

Keyword arguments are as described in the docstring of *contour*.

`property ax`

`changed`(*self*)
Call this whenever the mappable is changed to notify all the callbackSM listeners to the 'changed' signal.

`find_nearest_contour`(*self*, *x*, *y*, *indices=None*, *pixel=True*)
Find the point in the contour plot that is closest to `(x, y)`.

Parameters

x, y: float

The reference point.

indices

[list of int or None, default: None] Indices of contour levels to consider. If None (the default), all levels are considered.

pixel

[bool, default: True] If *True*, measure distance in pixel (screen) space, which is useful for manual contour labeling; else, measure distance in axes space.

Returns

contour

[*Collection*] The contour that is closest to `(x, y)`.

segment

[int] The index of the *Path* in *contour* that is closest to `(x, y)`.

index

[int] The index of the path segment in *segment* that is closest to `(x, y)`.

xmin, ymin

[float] The point in the contour plot that is closest to `(x, y)`.

d

[float] The distance from `(xmin, ymin)` to `(x, y)`.

`get_alpha`(*self*)
Return alpha to be applied to all ContourSet artists.

`get_transform`(*self*)
Return the *Transform* instance used by this ContourSet.

`legend_elements`(*self*, *variable_name='x'*, *str_format=<class 'str'>*)
Return a list of artists and labels suitable for passing through to *legend* which represent this ContourSet.

The labels have the form "0 < x <= 1" stating the data ranges which the artists represent.

Parameters

variable_name

[str] The string used inside the inequality used on the labels.

str_format

[function: float -> str] Function used to format the numbers in the labels.

Returns

artists

[List[*Artist*]] A list of the artists.

labels

[List[str]] A list of the labels.

`set_alpha`(*self*, *alpha*)

Set the alpha blending value for all ContourSet artists. *alpha* must be between 0 (transparent) and 1 (opaque).

`class matplotlib.contour.`**`QuadContourSet`**(*ax*, **args*, *levels=None*, *filled=False*, *linewidths=None*, *linestyles=None*, *hatches=None*, *alpha=None*, *origin=None*, *extent=None*, *cmap=None*, *colors=None*, *norm=None*, *vmin=None*, *vmax=None*, *extend='neither'*, *antialiased=None*, *nchunk=0*, *locator=None*, *transform=None*, ***kwargs*)

Bases: *matplotlib.contour.ContourSet*

Create and store a set of contour lines or filled regions.

User-callable method: *clabel*

Attributes

ax

The axes object in which the contours are drawn.

collections

A silent_list of LineCollections or PolyCollections.

levels

Contour levels.

layers

Same as levels for line contours; half-way between levels for filled contours. See `_process_colors()` method.

Draw contour lines or filled regions, depending on whether keyword arg *filled* is `False` (default) or `True`.

Call signature:

```
ContourSet(ax, levels, allsegs, [allkinds], **kwargs)
```

Parameters

ax

[*Axes*] The *Axes* object to draw on.

levels

[[level0, level1, ..., leveln]] A list of floating point numbers indicating the contour levels.

allsegs

[[level0segs, level1segs, ...]] List of all the polygon segments for all the *levels*. For contour lines `len(allsegs) == len(levels)`, and for filled contour regions `len(allsegs) = len(levels)-1`. The lists should look like

```
level0segs = [polygon0, polygon1, ...]
polygon0 = [[x0, y0], [x1, y1], ...]
```

allkinds

[[level0kinds, level1kinds, ...], optional] Optional list of all the polygon vertex kinds (code types), as described and used in Path. This is used to allow multiply- connected paths such as holes within filled polygons. If not `None`, `len(allkinds) == len(allsegs)`. The lists should look like

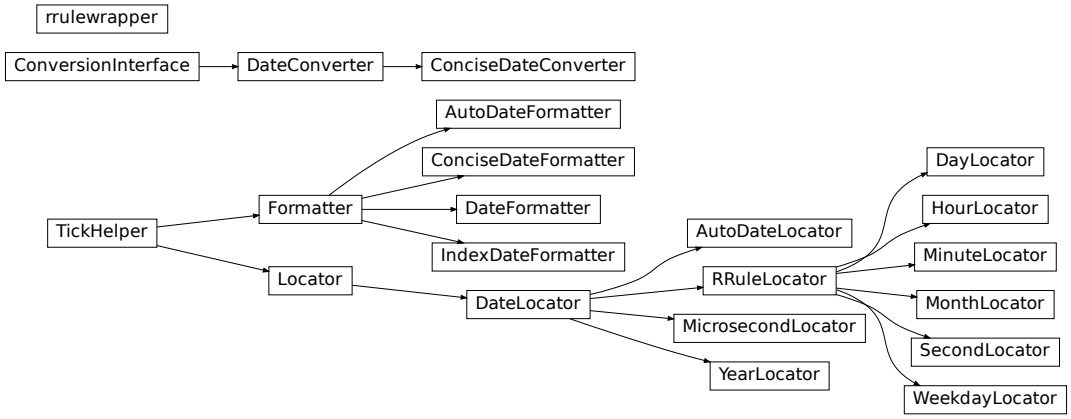
```
level0kinds = [polygon0kinds, ...]
polygon0kinds = [vertexcode0, vertexcode1, ...]
```

If *allkinds* is not `None`, usually all polygons for a particular contour level are grouped together so that `level0segs = [polygon0]` and `level0kinds = [polygon0kinds]`.

****kwargs**

Keyword arguments are as described in the docstring of *contour*.

18.21 matplotlib.dates



Matplotlib provides sophisticated date plotting capabilities, standing on the shoulders of python `datetime` and the add-on module `dateutil`.

18.21.1 Matplotlib date format

Matplotlib represents dates using floating point numbers specifying the number of days since a default epoch of 1970-01-01 UTC; for example, 1970-01-01, 06:00 is the floating point number 0.25. The formatters and locators require the use of `datetime.datetime` objects, so only dates between year 0001 and 9999 can be represented. Microsecond precision is achievable for (approximately) 70 years on either side of the epoch, and 20 microseconds for the rest of the allowable range of dates (year 0001 to 9999). The epoch can be changed at import time via *`dates.set_epoch`* or `rcParams["dates.epoch"]` to other dates if necessary; see /gallery/ticks_and_spines/date_precision_and_epochs for a discussion.

Note: Before Matplotlib 3.3, the epoch was 0000-12-31 which lost modern microsecond precision and also made the default axis limit of 0 an invalid datetime. In 3.3 the epoch was changed as above. To convert old ordinal floats to the new epoch, users can do:

```
new_ordinal = old_ordinal + mdates.date2num(np.datetime64('0000-12-31'))
```

There are a number of helper functions to convert between `datetime` objects and Matplotlib dates:

datestr2num	Convert a date string to a datenum using `dateutil.parser.parse`.
date2num	Convert datetime objects to Matplotlib dates.
num2date	Convert Matplotlib dates to `datetime` objects.
num2timedelta	Convert number of days to a `timedelta` object.
drange	Return a sequence of equally spaced Matplotlib dates.
set_epoch	Set the epoch (origin for dates) for datetime calculations.
get_epoch	Get the epoch used by *dates*.

Note: Like Python's `datetime.datetime`, Matplotlib uses the Gregorian calendar for all conversions between dates and floating point numbers. This practice is not universal, and calendar differences can cause confusing differences between what Python and Matplotlib give as the number of days since 0001-01-01 and what other software and databases yield. For example, the US Naval Observatory uses a calendar that switches from Julian to Gregorian in October, 1582. Hence, using their calculator, the number of days between 0001-01-01 and 2006-04-01 is 732403, whereas using the Gregorian calendar via the datetime module we find:

```
In [1]: date(2006, 4, 1).toordinal() - date(1, 1, 1).toordinal()
Out[1]: 732401
```

All the Matplotlib date converters, tickers and formatters are timezone aware. If no explicit timezone is provided, `rcParams["timezone"]` (default: `'UTC'`) is assumed. If you want to use a custom time zone, pass a `datetime.tzinfo` instance with the tz keyword argument to *num2date*, *plot_date*, and any custom date tickers or locators you create.

A wide range of specific and general purpose date tick locators and formatters are provided in this module. See *matplotlib.ticker* for general information on tick locators and formatters. These are described below.

The dateutil module provides additional code to handle date ticking, making it easy to place ticks on any kinds of dates. See examples below.

18.21.2 Date tickers

Most of the date tickers can locate single or multiple values. For example:

```
# import constants for the days of the week
from matplotlib.dates import MO, TU, WE, TH, FR, SA, SU

# tick on mondays every week
loc = WeekdayLocator(byweekday=MO, tz=tz)

# tick on mondays and saturdays
loc = WeekdayLocator(byweekday=(MO, SA))
```

In addition, most of the constructors take an interval argument:

```
# tick on mondays every second week
loc = WeekdayLocator(byweekday=MO, interval=2)
```

The rrule locator allows completely general date ticking:

```
# tick every 5th easter
rule = rrulewrapper(YEARLY, byeaster=1, interval=5)
loc = RRuleLocator(rule)
```

The available date tickers are:

- *MicrosecondLocator*: Locate microseconds.
- *SecondLocator*: Locate seconds.
- *MinuteLocator*: Locate minutes.
- *HourLocator*: Locate hours.
- *DayLocator*: Locate specified days of the month.
- *WeekdayLocator*: Locate days of the week, e.g., MO, TU.
- *MonthLocator*: Locate months, e.g., 7 for July.
- *YearLocator*: Locate years that are multiples of base.
- *RRuleLocator*: Locate using a `matplotlib.dates.rrulewrapper`. `rrulewrapper` is a simple wrapper around dateutil's `dateutil.rrule` which allow almost arbitrary date tick specifications. See rrule example.
- *AutoDateLocator*: On autoscale, this class picks the best *DateLocator* (e.g., *RRuleLocator*) to set the view limits and the tick locations. If called with `interval_multiples=True` it will make ticks line up with sensible multiples of the tick intervals. E.g. if the interval is 4 hours, it will pick hours 0, 4, 8, etc as ticks. This behaviour is not guaranteed by default.

18.21.3 Date formatters

The available date formatters are:

- *AutoDateFormatter*: attempts to figure out the best format to use. This is most useful when used with the *AutoDateLocator*.
- *ConciseDateFormatter*: also attempts to figure out the best format to use, and to make the format as compact as possible while still having complete date information. This is most useful when used with the *AutoDateLocator*.
- *DateFormatter*: use `strftime` format strings.
- *IndexDateFormatter*: date plots with implicit *x* indexing.

class matplotlib.dates.**AutoDateFormatter**(*locator*, *tz=None*, *defaultfmt='%Y-%m-%d'*)

Bases: *matplotlib.ticker.Formatter*

A *Formatter* which attempts to figure out the best format to use. This is most useful when used with the *AutoDateLocator*.

The AutoDateFormatter has a scale dictionary that maps the scale of the tick (the distance in days between one major tick) and a format string. The default looks like this:

```
self.scaled = {
    DAYS_PER_YEAR: rcParams['date.autoformat.year'],
    DAYS_PER_MONTH: rcParams['date.autoformat.month'],
    1.0: rcParams['date.autoformat.day'],
    1. / HOURS_PER_DAY: rcParams['date.autoformat.hour'],
    1. / (MINUTES_PER_DAY): rcParams['date.autoformat.minute'],
    1. / (SEC_PER_DAY): rcParams['date.autoformat.second'],
    1. / (MUSECONDS_PER_DAY): rcParams['date.autoformat.microsecond'],
}
```

The algorithm picks the key in the dictionary that is >= the current scale and uses that format string. You can customize this dictionary by doing:

```
>>> locator = AutoDateLocator()
>>> formatter = AutoDateFormatter(locator)
>>> formatter.scaled[1/(24.*60.)] = '%M:%S' # only show min and sec
```

A custom *FuncFormatter* can also be used. The following example shows how to use a custom format function to strip trailing zeros from decimal seconds and adds the date to the first ticklabel:

```
>>> def my_format_function(x, pos=None):
...     x = matplotlib.dates.num2date(x)
...     if pos == 0:
...         fmt = '%D %H:%M:%S.%f'
...     else:
...         fmt = '%H:%M:%S.%f'
```

(continues on next page)

(continued from previous page)

```
...     label = x.strftime(fmt)
...     label = label.rstrip("0")
...     label = label.rstrip(".")
...     return label
>>> from matplotlib.ticker import FuncFormatter
>>> formatter.scaled[1/(24.*60.)] = FuncFormatter(my_format_function)
```

Autoformat the date labels. The default format is the one to use if none of the values in `self.scaled` are greater than the unit returned by `locator._get_unit()`.

class matplotlib.dates.**AutoDateLocator**(*tz=None*, *minticks=5*, *maxticks=None*, *interval_multiples=True*)

Bases: *matplotlib.dates.DateLocator*

On autoscale, this class picks the best *DateLocator* to set the view limits and the tick locations.

Attributes

intervald

[dict] Mapping of tick frequencies to multiples allowed for that ticking. The default is

```
self.intervald = {
    YEARLY  : [1, 2, 4, 5, 10, 20, 40, 50, 100, 200, 400, 500,
               1000, 2000, 4000, 5000, 10000],
    MONTHLY : [1, 2, 3, 4, 6],
    DAILY   : [1, 2, 3, 7, 14, 21],
    HOURLY  : [1, 2, 3, 4, 6, 12],
    MINUTELY: [1, 5, 10, 15, 30],
    SECONDLY: [1, 5, 10, 15, 30],
    MICROSECONDLY: [1, 2, 5, 10, 20, 50, 100, 200, 500,
                    1000, 2000, 5000, 10000, 20000, 50000,
                    100000, 200000, 500000, 1000000],
}
```

where the keys are defined in `dateutil.rrule`.

The interval is used to specify multiples that are appropriate for the frequency of ticking. For instance, every 7 days is sensible for daily ticks, but for minutes/seconds, 15 or 30 make sense.

When customizing, you should only modify the values for the existing keys. You should not add or delete entries.

Example for forcing ticks every 3 hours:

```
locator = AutoDateLocator()
locator.intervald[HOURLY] = [3]  # only show every 3 hours
```

Parameters

tz

[datetime.tzinfo] Ticks timezone.

minticks

[int] The minimum number of ticks desired; controls whether ticks occur yearly, monthly, etc.

maxticks

[int] The maximum number of ticks desired; controls the interval between ticks (ticking every other, every 3, etc.). For fine-grained control, this can be a dictionary mapping individual rrule frequency constants (YEARLY, MONTHLY, etc.) to their own maximum number of ticks. This can be used to keep the number of ticks appropriate to the format chosen in *AutoDateFormatter*. Any frequency not specified in this dictionary is given a default value.

interval_multiples

[bool, default: True] Whether ticks should be chosen to be multiple of the interval, locking them to 'nicer' locations. For example, this will force the ticks to be at hours 0, 6, 12, 18 when hourly ticking is done at 6 hour intervals.

`autoscale`(*self*)
[*Deprecated*] Try to choose the view limits intelligently.

Notes

Deprecated since version 3.2.

`get_locator`(*self*, *dmin*, *dmax*)
Pick the best locator based on a distance.

`nonsingular`(*self*, *vmin*, *vmax*)
Given the proposed upper and lower extent, adjust the range if it is too close to being singular (i.e. a range of ~0).

`tick_values`(*self*, *vmin*, *vmax*)
Return the values of the located ticks given **vmin** and **vmax**.

Note: To get tick locations with the vmin and vmax values defined automatically for the associated `axis` simply call the Locator instance:

```
>>> print(type(loc))
<type 'Locator'>
>>> print(loc())
[1, 2, 3, 4]
```

class matplotlib.dates.**ConciseDateConverter**(*formats=None*, *zero_formats=None*, *offset_formats=None*, *show_offset=True*)

Bases: *matplotlib.dates.DateConverter*

axisinfo(*self*, *unit*, *axis*)

Return the *AxisInfo* for *unit*.

unit is a tzinfo instance or None. The *axis* argument is required but not used.

class matplotlib.dates.**ConciseDateFormatter**(*locator*, *tz=None*, *formats=None*, *offset_formats=None*, *zero_formats=None*, *show_offset=True*)

Bases: *matplotlib.ticker.Formatter*

A *Formatter* which attempts to figure out the best format to use for the date, and to make it as compact as possible, but still be complete. This is most useful when used with the *AutoDateLocator*:

```
>>> locator = AutoDateLocator()
>>> formatter = ConciseDateFormatter(locator)
```

Parameters

locator

[*ticker.Locator*] Locator that this axis is using.

tz

[str, optional] Passed to *dates.date2num*.

formats

[list of 6 strings, optional] Format strings for 6 levels of tick labelling: mostly years, months, days, hours, minutes, and seconds. Strings use the same format codes as `strftime`. Default is `['%Y', '%b', '%d', '%H:%M', '%H:%M', '%S.%f']`

zero_formats

[list of 6 strings, optional] Format strings for tick labels that are "zeros" for a given tick level. For instance, if most ticks are months, ticks around 1 Jan 2005 will be labeled "Dec", "2005", "Feb". The default is `['', '%Y', '%b', '%b-%d', '%H:%M', '%H:%M']`

offset_formats

[list of 6 strings, optional] Format strings for the 6 levels that is applied to the "offset" string found on the right side of an x-axis, or top of a y-axis. Combined with the tick labels this should completely specify the date. The default is:

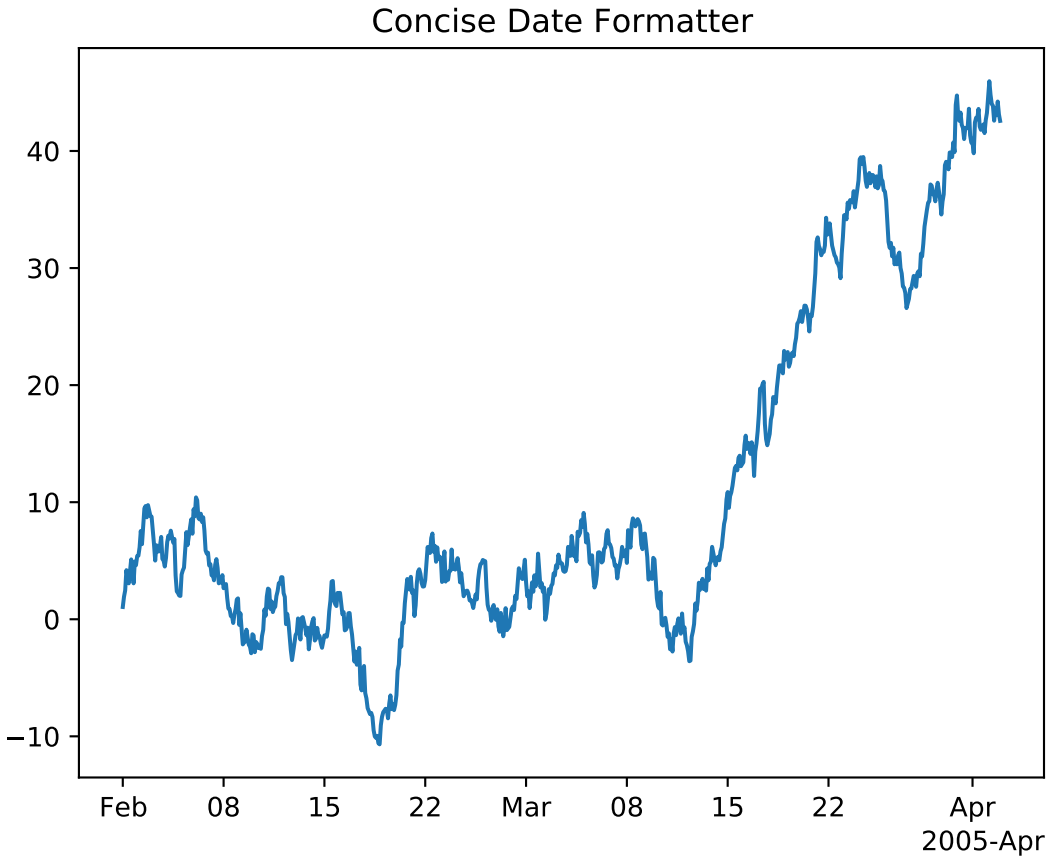
```
['', '%Y', '%Y-%b', '%Y-%b-%d', '%Y-%b-%d', '%Y-%b-%d %H:%M']
```

show_offset

[bool, default: True] Whether to show the offset or not.

Examples

See /gallery/ticks_and_spines/date_concise_formatter



Autoformat the date labels. The default format is used to form an initial string, and then redundant elements are removed.

`format_data_short`(*self*, *value*)
: Return a short string version of the tick value.

  Defaults to the position-independent long value.

`format_ticks`(*self*, *values*)
: Return the tick labels for all the ticks at once.

get_offset(*self*)

class matplotlib.dates.DateConverter

Bases: *matplotlib.units.ConversionInterface*

Converter for `datetime.date` and `datetime.datetime` data, or for date/time data represented as it would be converted by *date2num*.

The 'unit' tag for such data is None or a tzinfo instance.

static axisinfo(*unit*, *axis*)

Return the *AxisInfo* for *unit*.

unit is a tzinfo instance or None. The *axis* argument is required but not used.

static convert(*value*, *unit*, *axis*)

If *value* is not already a number or sequence of numbers, convert it with *date2num*.

The *unit* and *axis* arguments are not used.

static default_units(*x*, *axis*)

Return the tzinfo instance of *x* or of its first element, or None

class matplotlib.dates.DateFormatter(*fmt*, *tz=None*)

Bases: *matplotlib.ticker.Formatter*

Format a tick (in days since the epoch) with a `strftime` format string.

Parameters

fmt

[str] `strftime` format string

tz

[`datetime.tzinfo`, default: `rcParams["timezone"]` (default: `'UTC'`)] Ticks timezone.

property illegal_s

set_tzinfo(*self*, *tz*)

class matplotlib.dates.DateLocator(*tz=None*)

Bases: *matplotlib.ticker.Locator*

Determines the tick locations when plotting dates.

This class is subclassed by other Locators and is not meant to be used on its own.

Parameters

tz

[`datetime.tzinfo`]

`datalim_to_dt`(*self*)
Convert axis data interval to datetime objects.

`hms0d = {'byhour': 0, 'byminute': 0, 'bysecond': 0}`

`nonsingular`(*self*, *vmin*, *vmax*)
Given the proposed upper and lower extent, adjust the range if it is too close to being singular (i.e. a range of ~0).

`set_tzinfo`(*self*, *tz*)
Set time zone info.

`viewlim_to_dt`(*self*)
Convert the view interval to datetime objects.

`class matplotlib.dates.DayLocator`(*bymonthday=None*, *interval=1*, *tz=None*)
Bases: *matplotlib.dates.RRuleLocator*

Make ticks on occurrences of each day of the month. For example, 1, 15, 30.

Mark every day in *bymonthday*; *bymonthday* can be an int or sequence.

Default is to tick every day of the month: `bymonthday=range(1, 32)`.

`class matplotlib.dates.HourLocator`(*byhour=None*, *interval=1*, *tz=None*)
Bases: *matplotlib.dates.RRuleLocator*

Make ticks on occurrences of each hour.

Mark every hour in *byhour*; *byhour* can be an int or sequence. Default is to tick every hour: `byhour=range(24)`

interval is the interval between each iteration. For example, if `interval=2`, mark every second occurrence.

`class matplotlib.dates.IndexDateFormatter`(***kwargs*)
Bases: *matplotlib.ticker.Formatter*

[*Deprecated*] Use with *IndexLocator* to cycle format strings by index.

Notes

Deprecated since version 3.3.

Parameters

t

[list of float] A sequence of dates (floating point days).

fmt

[str] A `strftime` format string.

class matplotlib.dates.MicrosecondLocator(*interval=1*, *tz=None*)

Bases: *matplotlib.dates.DateLocator*

Make ticks on regular intervals of one or more microsecond(s).

Note: By default, Matplotlib uses a floating point representation of time in days since the epoch, so plotting data with microsecond time resolution does not work well for dates that are far (about 70 years) from the epoch (check with *get_epoch*).

If you want sub-microsecond resolution time plots, it is strongly recommended to use floating point seconds, not datetime-like time representation.

If you really must use datetime.datetime() or similar and still need microsecond precision, change the time origin via *dates.set_epoch* to something closer to the dates being plotted. See /gallery/ticks_and_spines/date_precision_and_epochs.

---

interval is the interval between each iteration. For example, if `interval=2`, mark every second microsecond.

set_axis(*self*, *axis*)

set_data_interval(*self*, *vmin*, *vmax*)

set_view_interval(*self*, *vmin*, *vmax*)

tick_values(*self*, *vmin*, *vmax*)

Return the values of the located ticks given **vmin** and **vmax**.

Note: To get tick locations with the vmin and vmax values defined automatically for the associated `axis` simply call the Locator instance:

```
>>> print(type(loc))
<type 'Locator'>
>>> print(loc())
[1, 2, 3, 4]
```

---

class matplotlib.dates.MinuteLocator(*byminute=None*, *interval=1*, *tz=None*)

Bases: *matplotlib.dates.RRuleLocator*

Make ticks on occurrences of each minute.

Mark every minute in *byminute*; *byminute* can be an int or sequence. Default is to tick every minute: `byminute=range(60)`

interval is the interval between each iteration. For example, if `interval=2`, mark every second occurrence.

class matplotlib.dates.MonthLocator(*bymonth=None*, *bymonthday=1*, *interval=1*, *tz=None*)

Bases: *matplotlib.dates.RRuleLocator*

Make ticks on occurrences of each month, e.g., 1, 3, 12.

Mark every month in *bymonth*; *bymonth* can be an int or sequence. Default is `range(1, 13)`, i.e. every month.

interval is the interval between each iteration. For example, if `interval=2`, mark every second occurrence.

`class matplotlib.dates.RRuleLocator(o, tz=None)`

Bases: *matplotlib.dates.DateLocator*

Parameters

tz

[datetime.tzinfo]

`autoscale(self)`

[*Deprecated*] Set the view limits to include the data range.

Notes

Deprecated since version 3.2.

`static get_unit_generic(freq)`

`tick_values(self, vmin, vmax)`

Return the values of the located ticks given **vmin** and **vmax**.

Note: To get tick locations with the vmin and vmax values defined automatically for the associated `axis` simply call the Locator instance:

```
>>> print(type(loc))
<type 'Locator'>
>>> print(loc())
[1, 2, 3, 4]
```

`class matplotlib.dates.SecondLocator(bysecond=None, interval=1, tz=None)`

Bases: *matplotlib.dates.RRuleLocator*

Make ticks on occurrences of each second.

Mark every second in *bysecond*; *bysecond* can be an int or sequence. Default is to tick every second: `bysecond = range(60)`

interval is the interval between each iteration. For example, if `interval=2`, mark every second occurrence.

`class matplotlib.dates.WeekdayLocator(byweekday=1, interval=1, tz=None)`

Bases: *matplotlib.dates.RRuleLocator*

Make ticks on occurrences of each weekday.

Mark every weekday in *byweekday*; *byweekday* can be a number or sequence.

Elements of *byweekday* must be one of MO, TU, WE, TH, FR, SA, SU, the constants from `dateutil.rrule`, which have been imported into the *matplotlib.dates* namespace.

interval specifies the number of weeks to skip. For example, `interval=2` plots every second week.

`class matplotlib.dates.YearLocator(`*base=1*, *month=1*, *day=1*, *tz=None*`)`

Bases: *matplotlib.dates.DateLocator*

Make ticks on a given day of each year that is a multiple of base.

Examples:

```
# Tick every year on Jan 1st
locator = YearLocator()

# Tick every 5 years on July 4th
locator = YearLocator(5, month=7, day=4)
```

Mark years that are multiple of base on a given month and day (default jan 1).

`autoscale(`*self*`)`

[*Deprecated*] Set the view limits to include the data range.

Notes

Deprecated since version 3.2.

`tick_values(`*self*, *vmin*, *vmax*`)`

Return the values of the located ticks given **vmin** and **vmax**.

Note: To get tick locations with the vmin and vmax values defined automatically for the associated `axis` simply call the Locator instance:

```
>>> print(type(loc))
<type 'Locator'>
>>> print(loc())
[1, 2, 3, 4]
```

`matplotlib.dates.date2num(`*d*`)`

Convert datetime objects to Matplotlib dates.

Parameters

d

[`datetime.datetime` or `numpy.datetime64` or sequences of these]

Returns

float or sequence of floats

Number of days since the epoch. See *get_epoch* for the epoch, which can be changed by rcParams["date.epoch"] (default: '1970-01-01T00:00:00') or *set_epoch*. If the epoch is "1970-01-01T00:00:00" (default) then noon Jan 1 1970 ("1970-01-01T12:00:00") returns 0.5.

Notes

The Gregorian calendar is assumed; this is not universal practice. For details see the module docstring.

matplotlib.dates.**datestr2num**(*d*, *default=None*)

Convert a date string to a datenum using dateutil.parser.parse.

Parameters

d

[str or sequence of str] The dates to convert.

default

[datetime.datetime, optional] The default date to use when fields are missing in *d*.

matplotlib.dates.**drange**(*dstart*, *dend*, *delta*)

Return a sequence of equally spaced Matplotlib dates.

The dates start at *dstart* and reach up to, but not including *dend*. They are spaced by *delta*.

Parameters

dstart, dend

[datetime] The date limits.

delta

[datetime.timedelta] Spacing of the dates.

Returns

numpy.array

A list floats representing Matplotlib dates.

matplotlib.dates.**epoch2num**(*e*)

Convert UNIX time to days since Matplotlib epoch.

Parameters

e

[list of floats] Time in seconds since 1970-01-01.

Returns

`numpy.array`

Time in days since Matplotlib epoch (see *get_epoch()*).

`matplotlib.dates.get_epoch()`
Get the epoch used by *dates*.

Returns

epoch: str

String for the epoch (parsable by `numpy.datetime64`).

`matplotlib.dates.mx2num(mxdates)`
[*Deprecated*] Convert mx `datetime` instance (or sequence of mx instances) to the new date format.

Notes

Deprecated since version 3.2.

`matplotlib.dates.num2date(x, tz=None)`
Convert Matplotlib dates to `datetime` objects.

Parameters

x

[float or sequence of floats] Number of days (fraction part represents hours, minutes, seconds) since the epoch. See *get_epoch* for the epoch, which can be changed by `rcParams["date.epoch"]` (default: `'1970-01-01T00:00:00'`) or *set_epoch*.

tz

[str, optional] Timezone of *x* (defaults to `rcParams["timezone"]` (default: `'UTC'`)).

Returns

`datetime` **or sequence of** `datetime`

Dates are returned in timezone *tz*.

If *x* is a sequence, a sequence of `datetime` objects will be returned.

Notes

The addition of one here is a historical artifact. Also, note that the Gregorian calendar is assumed; this is not universal practice. For details, see the module docstring.

`matplotlib.dates.`**num2epoch**(*d*)

Convert days since Matplotlib epoch to UNIX time.

Parameters

d

[list of floats] Time in days since Matplotlib epoch (see *get_epoch()*).

Returns

`numpy.array`

Time in seconds since 1970-01-01.

`matplotlib.dates.`**num2timedelta**(*x*)

Convert number of days to a `timedelta` object.

If *x* is a sequence, a sequence of `timedelta` objects will be returned.

Parameters

x

[float, sequence of floats] Number of days. The fraction part represents hours, minutes, seconds.

Returns

`datetime.timedelta` **or list[**`datetime.timedelta`**]**

`class matplotlib.dates.`**relativedelta**(*dt1=None*, *dt2=None*, *years=0*, *months=0*, *days=0*, *leapdays=0*, *weeks=0*, *hours=0*, *minutes=0*, *seconds=0*, *microseconds=0*, *year=None*, *month=None*, *day=None*, *weekday=None*, *yearday=None*, *nlyearday=None*, *hour=None*, *minute=None*, *second=None*, *microsecond=None*)

Bases: `object`

The relativedelta type is designed to be applied to an existing datetime and can replace specific components of that datetime, or represents an interval of time.

It is based on the specification of the excellent work done by M.-A. Lemburg in his mx.DateTime extension. However, notice that this type does *NOT* implement the same algorithm as his work. Do *NOT* expect it to behave like mx.DateTime's counterpart.

There are two different ways to build a relativedelta instance. The first one is passing it two date/datetime classes:

```
relativedelta(datetime1, datetime2)
```

The second one is passing it any number of the following keyword arguments:

```
relativedelta(arg1=x,arg2=y,arg3=z...)

year, month, day, hour, minute, second, microsecond:
    Absolute information (argument is singular); adding or subtracting a
    relativedelta with absolute information does not perform an arithmetic
    operation, but rather REPLACES the corresponding value in the
    original datetime with the value(s) in relativedelta.

years, months, weeks, days, hours, minutes, seconds, microseconds:
    Relative information, may be negative (argument is plural); adding
    or subtracting a relativedelta with relative information performs
    the corresponding arithmetic operation on the original datetime value
    with the information in the relativedelta.

weekday:
    One of the weekday instances (MO, TU, etc) available in the
    relativedelta module. These instances may receive a parameter N,
    specifying the Nth weekday, which could be positive or negative
    (like MO(+1) or MO(-2)). Not specifying it is the same as specifying
    +1. You can also use an integer, where 0=MO. This argument is always
    relative e.g. if the calculated date is already Monday, using MO(1)
    or MO(-1) won't change the day. To effectively make it absolute, use
    it in combination with the day argument (e.g. day=1, MO(1) for first
    Monday of the month).

leapdays:
    Will add given days to the date found, if year is a leap
    year, and the date found is post 28 of february.

yearday, nlyearday:
    Set the yearday or the non-leap year day (jump leap days).
    These are converted to day/month/leapdays information.
```

There are relative and absolute forms of the keyword arguments. The plural is relative, and the singular is absolute. For each argument in the order below, the absolute form is applied first (by setting each attribute to that value) and then the relative form (by adding the value to the attribute).

The order of attributes considered when this relativedelta is added to a datetime is:

1. Year
2. Month
3. Day
4. Hours
5. Minutes
6. Seconds
7. Microseconds

Finally, weekday is applied, using the rule described above.

For example

```
>>> from datetime import datetime
>>> from dateutil.relativedelta import relativedelta, MO
>>> dt = datetime(2018, 4, 9, 13, 37, 0)
>>> delta = relativedelta(hours=25, day=1, weekday=MO(1))
>>> dt + delta
datetime.datetime(2018, 4, 2, 14, 37)
```

First, the day is set to 1 (the first of the month), then 25 hours are added, to get to the 2nd day and 14th hour, finally the weekday is applied, but since the 2nd is already a Monday there is no effect.

normalized(*self*)

Return a version of this object represented entirely using integer values for the relative attributes.

```
>>> relativedelta(days=1.5, hours=2).normalized()
relativedelta(days=+1, hours=+14)
```

Returns

Returns a `dateutil.relativedelta.relativedelta` object.

property weeks

class matplotlib.dates.**rrule**(*freq*, *dtstart=None*, *interval=1*, *wkst=None*, *count=None*, *until=None*, *bysetpos=None*, *bymonth=None*, *bymonthday=None*, *byyearday=None*, *byeaster=None*, *byweekno=None*, *byweekday=None*, *byhour=None*, *byminute=None*, *bysecond=None*, *cache=False*)

Bases: `dateutil.rrule.rrulebase`

That's the base of the rrule operation. It accepts all the keywords defined in the RFC as its constructor parameters (except byday, which was renamed to

byweekday) and more. The constructor prototype is:

```
rrule(freq)
```

Where freq must be one of YEARLY, MONTHLY, WEEKLY, DAILY, HOURLY, MINUTELY, or SECONDLY.

Note: Per RFC section 3.3.10, recurrence instances falling on invalid dates and times are ignored rather than coerced:

> Recurrence rules may generate recurrence instances with an invalid date (e.g., February 30) or nonexistent local time (e.g., 1:30 AM on a day where the local time is moved forward by an hour at 1:00 AM). Such recurrence instances MUST be ignored and MUST NOT be counted as part of the recurrence set.

This can lead to possibly surprising behavior when, for example, the start date occurs at the end of the month:

```
>>> from dateutil.rrule import rrule, MONTHLY
>>> from datetime import datetime
>>> start_date = datetime(2014, 12, 31)
>>> list(rrule(freq=MONTHLY, count=4, dtstart=start_date))
...
[datetime.datetime(2014, 12, 31, 0, 0),
 datetime.datetime(2015, 1, 31, 0, 0),
 datetime.datetime(2015, 3, 31, 0, 0),
 datetime.datetime(2015, 5, 31, 0, 0)]
```

Additionally, it supports the following keyword arguments:

Parameters

- `dtstart` -- The recurrence start. Besides being the base for the recurrence, missing parameters in the final recurrence instances will also be extracted from this date. If not given, datetime.now() will be used instead.
- `interval` -- The interval between each freq iteration. For example, when using YEARLY, an interval of 2 means once every two years, but with HOURLY, it means once every two hours. The default interval is 1.
- `wkst` -- The week start day. Must be one of the MO, TU, WE constants, or an integer, specifying the first day of the week. This will affect recurrences based on weekly periods. The default week start is got from calendar.firstweekday(), and may be modified by calendar.setfirstweekday().
- `count` -- If given, this determines how many occurrences will be generated.

Note: As of version 2.5.0, the use of the keyword `until` in conjunction with `count` is deprecated, to make sure `dateutil` is fully compliant with RFC-5545 Sec. 3.3.10. Therefore, `until` and `count` **must not** occur in the same call to `rrule`.

- `until` -- If given, this must be a datetime instance specifying the upper-bound limit of the recurrence. The last recurrence in the rule is the greatest datetime that is less than or equal to the value specified in the `until` parameter.

Note: As of version 2.5.0, the use of the keyword `until` in conjunction with `count` is deprecated, to make sure `dateutil` is fully compliant with RFC-5545 Sec. 3.3.10. Therefore, `until` and `count` **must not** occur in the same call to `rrule`.

- `bysetpos` -- If given, it must be either an integer, or a sequence of integers, positive or negative. Each given integer will specify an occurrence number, corresponding to the nth occurrence of the rule inside the frequency period. For example, a bysetpos of -1 if combined with a MONTHLY frequency, and a byweekday of (MO, TU, WE, TH, FR), will result in the last work day of every month.
- `bymonth` -- If given, it must be either an integer, or a sequence of integers, meaning the months to apply the recurrence to.
- `bymonthday` -- If given, it must be either an integer, or a sequence of integers, meaning the month days to apply the recurrence to.
- `byyearday` -- If given, it must be either an integer, or a sequence of integers, meaning the year days to apply the recurrence to.
- `byeaster` -- If given, it must be either an integer, or a sequence of integers, positive or negative. Each integer will define an offset from the Easter Sunday. Passing the offset 0 to byeaster will yield the Easter Sunday itself. This is an extension to the RFC specification.
- `byweekno` -- If given, it must be either an integer, or a sequence of integers, meaning the week numbers to apply the recurrence to. Week numbers have the meaning described in ISO8601, that is, the first week of the year is that containing at least four days of the new year.
- `byweekday` -- If given, it must be either an integer (0 == MO), a sequence of integers, one of the weekday constants (MO, TU, etc), or a sequence of these constants. When given, these variables will define the weekdays where the recurrence will be ap-

plied. It's also possible to use an argument n for the weekday instances, which will mean the nth occurrence of this weekday in the period. For example, with MONTHLY, or with YEARLY and BYMONTH, using FR(+1) in byweekday will specify the first friday of the month where the recurrence happens. Notice that in the RFC documentation, this is specified as BYDAY, but was renamed to avoid the ambiguity of that keyword.

- `byhour` -- If given, it must be either an integer, or a sequence of integers, meaning the hours to apply the recurrence to.
- `byminute` -- If given, it must be either an integer, or a sequence of integers, meaning the minutes to apply the recurrence to.
- `bysecond` -- If given, it must be either an integer, or a sequence of integers, meaning the seconds to apply the recurrence to.
- `cache` -- If given, it must be a boolean value specifying to enable or disable caching of results. If you will use the same rrule instance multiple times, enabling caching will improve the performance considerably.

`replace`(*self*, ***kwargs*)

Return new rrule with same attributes except for those attributes given new values by whichever keyword arguments are specified.

`matplotlib.dates.set_epoch`(*epoch*)

Set the epoch (origin for dates) for datetime calculations.

The default epoch is `rcParams["dates.epoch"]` (by default 1970-01-01T00:00).

If microsecond accuracy is desired, the date being plotted needs to be within approximately 70 years of the epoch. Matplotlib internally represents dates as days since the epoch, so floating point dynamic range needs to be within a factor fo 2^52.

set_epoch must be called before any dates are converted (i.e. near the import section) or a RuntimeError will be raised.

See also /gallery/ticks_and_spines/date_precision_and_epochs.

Parameters

epoch

[str] valid UTC date parsable by `numpy.datetime64` (do not include timezone).

18.22 matplotlib.dviread

A module for reading dvi files output by TeX. Several limitations make this not (currently) useful as a general-purpose dvi preprocessor, but it is currently used by the pdf backend for processing usetex text.

Interface:

```
with Dvi(filename, 72) as dvi:
    # iterate over pages:
    for page in dvi:
        w, h, d = page.width, page.height, page.descent
        for x, y, font, glyph, width in page.text:
            fontname = font.texname
            pointsize = font.size
            ...
        for x, y, height, width in page.boxes:
            ...
```

class matplotlib.dviread.Dvi(*filename*, *dpi*)

Bases: object

A reader for a dvi ("device-independent") file, as produced by TeX. The current implementation can only iterate through pages in order, and does not even attempt to verify the postamble.

This class can be used as a context manager to close the underlying file upon exit. Pages can be read via iteration. Here is an overly simple way to extract text without trying to detect whitespace:

```
>>> with matplotlib.dviread.Dvi('input.dvi', 72) as dvi:
...     for page in dvi:
...         print(''.join(chr(t.glyph) for t in page.text))
```

Read the data from the file named *filename* and convert TeX's internal units to units of *dpi* per inch. *dpi* only sets the units and does not limit the resolution. Use None to return TeX's internal units.

close(*self*)

Close the underlying file if it is open.

class matplotlib.dviread.DviFont(*scale*, *tfm*, *texname*, *vf*)

Bases: object

Encapsulation of a font that a DVI file can refer to.

This class holds a font's texname and size, supports comparison, and knows the widths of glyphs in the same units as the AFM file. There are also internal attributes (for use by dviread.py) that are *not* used for comparison.

The size is in Adobe points (converted from TeX points).

Parameters

scale

[float] Factor by which the font is scaled from its natural size.

tfm

[Tfm] TeX font metrics for this font

texname

[bytes] Name of the font as used internally by TeX and friends, as an ASCII bytestring. This is usually very different from any external font names, and `dviread.PsfontsMap` can be used to find the external name of the font.

vf

[Vf] A TeX "virtual font" file, or None if this font is not virtual.

Attributes

texname

[bytes]

size

[float] Size of the font in Adobe points, converted from the slightly smaller TeX points.

widths

[list] Widths of glyphs in glyph-space units, typically 1/1000ths of the point size.

size

texname

widths

class matplotlib.dviread.**Encoding**(***kwargs*)

Bases: `object`

[*Deprecated*] Parse a *.enc file referenced from a psfonts.map style file.

The format this class understands is a very limited subset of PostScript.

Usage (subject to change):

```
for name in Encoding(filename):
    whatever(name)
```

Parameters

filename

[str or path-like]

Notes

Deprecated since version 3.3.

Attributes

encoding

[list] List of character names

`encoding`

`class matplotlib.dviread.PsFont(`*texname*, *psname*, *effects*, *encoding*, *filename*`)`

Bases: `tuple`

Create new instance of PsFont(texname, psname, effects, encoding, filename)

`effects`
Alias for field number 2

`encoding`
Alias for field number 3

`filename`
Alias for field number 4

`psname`
Alias for field number 1

`texname`
Alias for field number 0

`class matplotlib.dviread.PsfontsMap(`*filename*`)`

Bases: `object`

A psfonts.map formatted file, mapping TeX fonts to PS fonts.

Parameters

filename

[str or path-like]

Notes

For historical reasons, TeX knows many Type-1 fonts by different names than the outside world. (For one thing, the names have to fit in eight characters.) Also, TeX's native fonts are not Type-1 but Metafont, which is nontrivial to convert to PostScript except as a bitmap. While high-quality conversions to Type-1 format exist and are shipped with modern TeX distributions, we need to know which Type-1 fonts are the counterparts of which native fonts. For these reasons a mapping is needed from internal font names to font file names.

A texmf tree typically includes mapping files called e.g. `psfonts.map`, `pdftex.map`, or `dvipdfm.map`. The file `psfonts.map` is used by **`dvips`**, `pdftex.map` by **`pdfTeX`**, and `dvipdfm.map` by **`dvipdfm`**. `psfonts.map` might avoid embedding the 35 PostScript fonts (i.e., have no filename for them, as in the Times-Bold example above), while the pdf-related files perhaps only avoid the "Base 14" pdf fonts. But the user may have configured these files differently.

Examples

```
>>> map = PsfontsMap(find_tex_file('pdftex.map'))
>>> entry = map[b'ptmbo8r']
>>> entry.texname
b'ptmbo8r'
>>> entry.psname
b'Times-Bold'
>>> entry.encoding
'/usr/local/texlive/2008/texmf-dist/fonts/enc/dvips/base/8r.enc'
>>> entry.effects
{'slant': 0.16700000000000001}
>>> entry.filename
```

class matplotlib.dviread.**Tfm**(*filename*)

Bases: `object`

A TeX Font Metric file.

This implementation covers only the bare minimum needed by the Dvi class.

Parameters

filename

[str or path-like]

Attributes

checksum

[int] Used for verifying against the dvi file.

design_size

[int] Design size of the font (unknown units)

width, height, depth

[dict] Dimensions of each character, need to be scaled by the factor specified in the dvi file. These are dicts because indexing may not start from 0.

checksum

depth

design_size

height

width

class matplotlib.dviread.**Vf**(*filename*)

Bases: *matplotlib.dviread.Dvi*

A virtual font (*.vf file) containing subroutines for dvi files.

Parameters

filename

[str or path-like]

Notes

The virtual font format is a derivative of dvi: http://mirrors.ctan.org/info/knuth/virtual-fonts This class reuses some of the machinery of *Dvi* but replaces the `_read` loop and dispatch mechanism.

Examples

```
vf = Vf(filename)
glyph = vf[code]
glyph.text, glyph.boxes, glyph.width
```

Read the data from the file named *filename* and convert TeX's internal units to units of *dpi* per inch. *dpi* only sets the units and does not limit the resolution. Use None to return TeX's internal units.

matplotlib.dviread.**find_tex_file**(*filename*, *format=None*)

Find a file in the texmf tree.

Calls `kpsewhich` which is an interface to the kpathsea library [1]. Most existing TeX distributions on Unix-like systems use kpathsea. It is also available as part of MikTeX, a popular distribution on Windows.

If the file is not found, an empty string is returned.

Parameters

filename

[str or path-like]

format

[str or bytes] Used as the value of the `--format` option to `kpsewhich`. Could be e.g. 'tfm' or 'vf' to limit the search to that type of files.

References

[1]

18.23 matplotlib.figure

matplotlib.figure implements the following classes:

Figure

Top level *Artist*, which holds all plot elements.

SubplotParams

Control the default spacing between subplots.

18.23.1 Classes

AxesStack	[*Deprecated*]
Figure	The top level container for all the plot elements.
SubplotParams	A class to hold the parameters for a subplot.

matplotlib.figure.AxesStack

class matplotlib.figure.**AxesStack**(***kwargs*)

Bases: matplotlib.figure._AxesStack

[*Deprecated*]

Notes

Deprecated since version 3.2:

__init__(*self*)

Initialize self. See help(type(self)) for accurate signature.

__module__ = 'matplotlib.figure'

Examples using matplotlib.figure.AxesStack

matplotlib.figure.Figure

class matplotlib.figure.**Figure**(*figsize=None*, *dpi=None*, *facecolor=None*, *edgecolor=None*, *linewidth=0.0*, *frameon=None*, *subplotpars=None*, *tight_layout=None*, *constrained_layout=None*)

Bases: *matplotlib.artist.Artist*

The top level container for all the plot elements.

The Figure instance supports callbacks through a *callbacks* attribute which is a *CallbackRegistry* instance. The events you can connect to are 'dpi_changed', and the callback will be called with `func(fig)` where fig is the *Figure* instance.

Attributes

patch

The *Rectangle* instance representing the figure background patch.

suppressComposite

For multiple figure images, the figure will make composite images depending on the renderer option_image_nocomposite function. If *suppressComposite* is a boolean, this will override the renderer.

Parameters

figsize

[2-tuple of floats, default: `rcParams["figure.figsize"]` (default: `[6.4, 4.8]`)] Figure dimension `(width, height)` in inches.

dpi

[float, default: `rcParams["figure.dpi"]` (default: `100.0`)] Dots per inch.

facecolor

[default: `rcParams["figure.facecolor"]` (default: `'white'`)] The figure patch facecolor.

edgecolor

[default: `rcParams["figure.edgecolor"]` (default: `'white'`)] The figure patch edge color.

linewidth

[float] The linewidth of the frame (i.e. the edge linewidth of the figure patch).

frameon

[bool, default: `rcParams["figure.frameon"]` (default: `True`)] If `False`, suppress drawing the figure background patch.

subplotpars

[*SubplotParams*] Subplot parameters. If not given, the default subplot parameters `rcParams["figure.subplot.*"]` are used.

tight_layout

[bool or dict, default: `rcParams["figure.autolayout"]` (default: `False`)] If `False` use *subplotpars*. If `True` adjust subplot parameters using *tight_layout* with default padding. When providing a dict containing the keys `pad`, `w_pad`, `h_pad`, and `rect`, the default *tight_layout* paddings will be overridden.

constrained_layout

[bool, default: `rcParams["figure.constrained_layout.use"]` (default: `False`)] If `True` use constrained layout to adjust positioning of plot elements. Like `tight_layout`, but designed to be more flexible. See *Constrained Layout Guide* for examples. (Note: does not work with *add_subplot* or *subplot2grid*.)

`__getstate__`(*self*)

`__init__`(*self*, *figsize=None*, *dpi=None*, *facecolor=None*, *edgecolor=None*, *linewidth=0.0*, *frameon=None*, *subplotpars=None*, *tight_layout=None*, *constrained_layout=None*)

Parameters

figsize

[2-tuple of floats, default: `rcParams["figure.figsize"]` (default: `[6.4, 4.8]`)] Figure dimension `(width, height)` in inches.

dpi

[float, default: `rcParams["figure.dpi"]` (default: `100.0`)] Dots per inch.

facecolor

[default: `rcParams["figure.facecolor"]` (default: `'white'`)] The figure patch facecolor.

edgecolor

[default: `rcParams["figure.edgecolor"]` (default: `'white'`)] The figure patch edge color.

linewidth

[float] The linewidth of the frame (i.e. the edge linewidth of the figure patch).

frameon

[bool, default: `rcParams["figure.frameon"]` (default: `True`)] If `False`, suppress drawing the figure background patch.

subplotpars

[*SubplotParams*] Subplot parameters. If not given, the default subplot parameters `rcParams["figure.subplot.*"]` are used.

tight_layout

[bool or dict, default: `rcParams["figure.autolayout"]` (default: `False`)] If `False` use *subplotpars*. If `True` adjust subplot parameters using *tight_layout* with default padding. When providing a dict containing the keys `pad`, `w_pad`, `h_pad`, and `rect`, the default *tight_layout* paddings will be overridden.

constrained_layout

[bool, default: `rcParams["figure.constrained_layout.use"]` (default: `False`)] If `True` use constrained layout to adjust positioning of plot elements. Like `tight_layout`, but designed to be more flexible. See *Constrained Layout Guide* for examples. (Note: does not work with *add_subplot* or *subplot2grid*.)

`__module__ = 'matplotlib.figure'`

`__repr__`(*self*)
Return repr(self).

`__setstate__`(*self*, *state*)

`__str__`(*self*)
Return str(self).

`add_artist`(*self*, *artist*, *clip=False*)
Add an *Artist* to the figure.

Usually artists are added to axes objects using *Axes.add_artist*; this method can be used in the rare cases where one needs to add artists directly to the figure instead.

Parameters

artist

[*Artist*] The artist to add to the figure. If the added artist has no transform previously set, its transform will be set to `figure.transFigure`.

clip

[bool, default: False] Whether the added artist should be clipped by the figure patch.

Returns

Artist

The added artist.

`add_axes`(*self*, **args*, ***kwargs*)
Add an axes to the figure.

Call signatures:

```
add_axes(rect, projection=None, polar=False, **kwargs)
add_axes(ax)
```

Parameters

rect

[sequence of float] The dimensions [left, bottom, width, height] of the new axes. All quantities are in fractions of figure width and height.

projection

[{None, 'aitoff', 'hammer', 'lambert', 'mollweide', 'polar', 'rectilinear', str}, optional] The projection type of the *Axes*. *str* is the name of a custom projection, see *projections*. The default None results in a 'rectilinear' projection.

polar

[bool, default: False] If True, equivalent to projection='polar'.

sharex, sharey

[*Axes*, optional] Share the x or y *axis* with sharex and/or sharey. The axis will have the same limits, ticks, and scale as the axis of the shared axes.

label

[str] A label for the returned axes.

Returns

Axes*, or a subclass of *Axes

The returned axes class depends on the projection used. It is *Axes* if rectilinear projection is used and *projections.polar.PolarAxes* if polar projection is used.

Other Parameters

****kwargs**

This method also takes the keyword arguments for the returned axes class. The keyword arguments for the rectilinear axes class *Axes* can be found in the following table but there might also be other keyword arguments if another projection is used, see the actual axes class.

Property	Description
adjustable	{'box', 'datalim'}
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and r
alpha	float or None
anchor	2-tuple of floats or {'C', 'SW', 'S', 'SE', ...}
animated	bool
aspect	{'auto'} or num
autoscale_on	bool
autoscalex_on	bool
autoscaley_on	bool
axes_locator	Callable[[Axes, Renderer], Bbox]
axisbelow	bool or 'line'
box_aspect	None, or a number
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
contains	unknown

Table 134 – continued from previous page

Property	Description
facecolor or fc	color
figure	*Figure*
frame_on	bool
gid	str
in_layout	bool
label	object
navigate	bool
navigate_mode	unknown
path_effects	*AbstractPathEffect*
picker	None or bool or callable
position	[left, bottom, width, height] or *Bbox*
prop_cycle	unknown
rasterization_zorder	float or None
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
title	str
transform	*Transform*
url	str
visible	bool
xbound	unknown
xlabel	str
xlim	(bottom: float, top: float)
xmargin	float greater than -0.5
xscale	{"linear", "log", "symlog", "logit", ...}
xticklabels	unknown
xticks	unknown
ybound	unknown
ylabel	str
ylim	(bottom: float, top: float)
ymargin	float greater than -0.5
yscale	{"linear", "log", "symlog", "logit", ...}
yticklabels	unknown
yticks	unknown
zorder	float

See also:

Figure.add_subplot

pyplot.subplot

pyplot.axes

Figure.subplots

pyplot.subplots

Notes

If the figure already has an axes with key (*args*, *kwargs*) then it will simply make that axes current and return it. This behavior is deprecated. Meanwhile, if you do not want this behavior (i.e., you want to force the creation of a new axes), you must use a unique set of args and kwargs. The axes *label* attribute has been exposed for this purpose: if you want two axes that are otherwise identical to be added to the figure, make sure you give them unique labels.

In rare circumstances, *add_axes* may be called with a single argument, a axes instance already created in the present figure but not in the figure's list of axes.

Examples

Some simple examples:

```
rect = l, b, w, h
fig = plt.figure()
fig.add_axes(rect, label=label1)
fig.add_axes(rect, label=label2)
fig.add_axes(rect, frameon=False, facecolor='g')
fig.add_axes(rect, polar=True)
ax = fig.add_axes(rect, projection='polar')
fig.delaxes(ax)
fig.add_axes(ax)
```

`add_axobserver`(*self*, *func*)
Whenever the axes state change, `func(self)` will be called.

`add_gridspec`(*self*, *nrows=1*, *ncols=1*, ***kwargs*)
Return a *GridSpec* that has this figure as a parent. This allows complex layout of axes in the figure.

Parameters

nrows

[int, default: 1] Number of rows in grid.

ncols

[int, default: 1] Number or columns in grid.

Returns

GridSpec

Other Parameters

****kwargs**

Keyword arguments are passed to *GridSpec*.

See also:

matplotlib.pyplot.subplots

Examples

Adding a subplot that spans two rows:

```
fig = plt.figure()
gs = fig.add_gridspec(2, 2)
ax1 = fig.add_subplot(gs[0, 0])
ax2 = fig.add_subplot(gs[1, 0])
# spans two rows:
ax3 = fig.add_subplot(gs[:, 1])
```

`add_subplot`(*self*, **args*, ***kwargs*)

Add an *Axes* to the figure as part of a subplot arrangement.

Call signatures:

```
add_subplot(nrows, ncols, index, **kwargs)
add_subplot(pos, **kwargs)
add_subplot(ax)
add_subplot()
```

Parameters

***args**

[int, (int, int, *index*), or *SubplotSpec*, default: (1, 1, 1)] The position of the subplot described by one of

- Three integers (*nrows*, *ncols*, *index*). The subplot will take the *index* position on a grid with *nrows* rows and *ncols* columns. *index* starts at 1 in the upper left corner and increases to the right. *index* can also be a two-tuple specifying the (*first*, *last*) indices (1-based, and including *last*) of the subplot, e.g., `fig.add_subplot(3, 1, (1, 2))` makes a subplot that spans the upper 2/3 of the figure.
- A 3-digit integer. The digits are interpreted as if given separately as three single-digit integers, i.e. `fig.add_subplot(235)` is the same as `fig.add_subplot(2, 3, 5)`. Note that this can only be used if there are no more than 9 subplots.

- A *SubplotSpec*.

In rare circumstances, *add_subplot* may be called with a single argument, a subplot axes instance already created in the present figure but not in the figure's list of axes.

projection

[{None, 'aitoff', 'hammer', 'lambert', 'mollweide', 'polar', 'rectilinear', str}, optional] The projection type of the subplot (*Axes*). *str* is the name of a custom projection, see *projections*. The default None results in a 'rectilinear' projection.

polar

[bool, default: False] If True, equivalent to projection='polar'.

sharex, sharey

[*Axes*, optional] Share the x or y *axis* with sharex and/or sharey. The axis will have the same limits, ticks, and scale as the axis of the shared axes.

label

[str] A label for the returned axes.

Returns

axes.SubplotBase, **or another subclass of** *Axes*

The axes of the subplot. The returned axes base class depends on the projection used. It is *Axes* if rectilinear projection is used and *projections.polar.PolarAxes* if polar projection is used. The returned axes is then a subplot subclass of the base class.

Other Parameters

****kwargs**

This method also takes the keyword arguments for the returned axes base class; except for the *figure* argument. The keyword arguments for the rectilinear base class *Axes* can be found in the following table but there might also be other keyword arguments if another projection is used.

Property	Description
adjustable	{'box', 'datalim'}
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and r
alpha	float or None
anchor	2-tuple of floats or {'C', 'SW', 'S', 'SE', ...}
animated	bool

Table 135 – continued from previous page

Property	Description
aspect	{'auto'} or num
autoscale_on	bool
autoscalex_on	bool
autoscaley_on	bool
axes_locator	Callable[[Axes, Renderer], Bbox]
axisbelow	bool or 'line'
box_aspect	None, or a number
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
contains	unknown
facecolor or fc	color
figure	*Figure*
frame_on	bool
gid	str
in_layout	bool
label	object
navigate	bool
navigate_mode	unknown
path_effects	*AbstractPathEffect*
picker	None or bool or callable
position	[left, bottom, width, height] or *Bbox*
prop_cycle	unknown
rasterization_zorder	float or None
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
title	str
transform	*Transform*
url	str
visible	bool
xbound	unknown
xlabel	str
xlim	(bottom: float, top: float)
xmargin	float greater than -0.5
xscale	{"linear", "log", "symlog", "logit", ...}
xticklabels	unknown
xticks	unknown
ybound	unknown
ylabel	str
ylim	(bottom: float, top: float)
ymargin	float greater than -0.5
yscale	{"linear", "log", "symlog", "logit", ...}

Table 135 – continued from previous page

Property	Description
yticklabels	unknown
yticks	unknown
zorder	float

See also:

Figure.add_axes

pyplot.subplot

pyplot.axes

Figure.subplots

pyplot.subplots

Notes

If the figure already has a subplot with key (*args*, *kwargs*) then it will simply make that subplot current and return it. This behavior is deprecated. Meanwhile, if you do not want this behavior (i.e., you want to force the creation of a new subplot), you must use a unique set of args and kwargs. The axes *label* attribute has been exposed for this purpose: if you want two subplots that are otherwise identical to be added to the figure, make sure you give them unique labels.

Examples

```
fig = plt.figure()

fig.add_subplot(231)
ax1 = fig.add_subplot(2, 3, 1)  # equivalent but more general

fig.add_subplot(232, frameon=False)  # subplot with no frame
fig.add_subplot(233, projection='polar')  # polar subplot
fig.add_subplot(234, sharex=ax1)  # subplot sharing x-axis with ax1
fig.add_subplot(235, facecolor="red")  # red subplot

ax1.remove()  # delete ax1 from the figure
fig.add_subplot(ax1)  # add ax1 back to the figure
```

`align_labels`(*self*, *axs=None*)

Align the xlabels and ylabels of subplots with the same subplots row or column (respectively) if label alignment is being done automatically (i.e. the label position is not manually set).

Alignment persists for draw events after this is called.

Parameters

axs

[list of *Axes*] Optional list (or ndarray) of *Axes* to align the labels. Default is to align all axes on the figure.

See also:

matplotlib.figure.Figure.align_xlabels

matplotlib.figure.Figure.align_ylabels

`align_xlabels`(*self*, *axs=None*)

Align the xlabels of subplots in the same subplot column if label alignment is being done automatically (i.e. the label position is not manually set).

Alignment persists for draw events after this is called.

If a label is on the bottom, it is aligned with labels on axes that also have their label on the bottom and that have the same bottom-most subplot row. If the label is on the top, it is aligned with labels on axes with the same top-most row.

Parameters

axs

[list of *Axes*] Optional list of (or ndarray) *Axes* to align the xlabels. Default is to align all axes on the figure.

See also:

matplotlib.figure.Figure.align_ylabels

matplotlib.figure.Figure.align_labels

Notes

This assumes that `axs` are from the same *GridSpec*, so that their *SubplotSpec* positions correspond to figure positions.

Examples

Example with rotated xtick labels:

```
fig, axs = plt.subplots(1, 2)
for tick in axs[0].get_xticklabels():
    tick.set_rotation(55)
axs[0].set_xlabel('XLabel 0')
axs[1].set_xlabel('XLabel 1')
fig.align_xlabels()
```

`align_ylabels`(*self*, *axs=None*)

Align the ylabels of subplots in the same subplot column if label alignment is being done automatically (i.e. the label position is not manually set).

Alignment persists for draw events after this is called.

If a label is on the left, it is aligned with labels on axes that also have their label on the left and that have the same left-most subplot column. If the label is on the right, it is aligned with labels on axes with the same right-most column.

Parameters

axs

[list of *Axes*] Optional list (or ndarray) of *Axes* to align the ylabels. Default is to align all axes on the figure.

See also:

matplotlib.figure.Figure.align_xlabels

matplotlib.figure.Figure.align_labels

Notes

This assumes that `axs` are from the same *GridSpec*, so that their *SubplotSpec* positions correspond to figure positions.

Examples

Example with large yticks labels:

```
fig, axs = plt.subplots(2, 1)
axs[0].plot(np.arange(0, 1000, 50))
axs[0].set_ylabel('YLabel 0')
axs[1].set_ylabel('YLabel 1')
fig.align_ylabels()
```

`autofmt_xdate`(*self*, *bottom=0.2*, *rotation=30*, *ha='right'*, *which='major'*)

Date ticklabels often overlap, so it is useful to rotate them and right align them. Also, a common use case is a number of subplots with shared xaxes where the x-axis is date data. The ticklabels are often long, and it helps to rotate them on the bottom subplot and turn them off on other subplots, as well as turn off xlabels.

Parameters

bottom

[float, default: 0.2] The bottom of the subplots for *subplots_adjust*.

rotation

[float, default: 30 degrees] The rotation angle of the xtick labels in degrees.

ha

[{'left', 'center', 'right'}, default: 'right'] The horizontal alignment of the xticklabels.

which

[{'major', 'minor', 'both'}, default: 'major'] Selects which ticklabels to rotate.

`property axes`

List of axes in the Figure. You can access and modify the axes in the Figure through this list.

Do not modify the list itself. Instead, use "*add_axes*, *add_subplot* or *delaxes* to add or remove an axes.

`clear`(*self*, *keep_observers=False*)

Clear the figure -- synonym for *clf*.

`clf`(*self*, *keep_observers=False*)

Clear the figure.

Set *keep_observers* to True if, for example, a gui widget is tracking the axes in the figure.

`colorbar`(*self*, *mappable*, *cax=None*, *ax=None*, *use_gridspec=True*, ***kw*)

Create a colorbar for a ScalarMappable instance, *mappable*.

Documentation for the pyplot thin wrapper:

Add a colorbar to a plot.

Function signatures for the *pyplot* interface; all but the first are also method signatures for the *colorbar* method:

```
colorbar(**kwargs)
colorbar(mappable, **kwargs)
colorbar(mappable, cax=cax, **kwargs)
colorbar(mappable, ax=ax, **kwargs)
```

Parameters

mappable

The *matplotlib.cm.ScalarMappable* (i.e., *AxesImage*, *ContourSet*, etc.) described by this colorbar. This argument is mandatory for the *Figure.colorbar* method but optional for the *pyplot.colorbar* function, which sets the default to the current image.

Note that one can create a *ScalarMappable* "on-the-fly" to generate colorbars not attached to a previously drawn artist, e.g.

```
fig.colorbar(cm.ScalarMappable(norm=norm, cmap=cmap), ax=ax)
```

cax

[*Axes*, optional] Axes into which the colorbar will be drawn.

ax

[*Axes*, list of Axes, optional] Parent axes from which space for a new colorbar axes will be stolen. If a list of axes is given they will all be resized to make room for the colorbar axes.

use_gridspec

[bool, optional] If *cax* is `None`, a new *cax* is created as an instance of Axes. If *ax* is an instance of Subplot and *use_gridspec* is `True`, *cax* is created as an instance of Subplot using the *gridspec* module.

Returns

colorbar

[*Colorbar*] See also its base class, *ColorbarBase*.

Notes

Additional keyword arguments are of two kinds:

axes properties:

fraction

[float, default: 0.15] Fraction of original axes to use for colorbar.

shrink

[float, default: 1.0] Fraction by which to multiply the size of the colorbar.

aspect

[float, default: 20] Ratio of long to short dimensions.

pad

[float, default: 0.05 if vertical, 0.15 if horizontal] Fraction of original axes between colorbar and new image axes.

anchor

[(float, float), optional] The anchor point of the colorbar axes. Defaults to (0.0, 0.5) if vertical; (0.5, 1.0) if horizontal.

panchor

[(float, float), or *False*, optional] The anchor point of the colorbar parent axes. If *False*, the parent axes' anchor will be unchanged. Defaults to (1.0, 0.5) if vertical; (0.5, 0.0) if horizontal.

colorbar properties:

Property	Description
extend	{'neither', 'both', 'min', 'max'} If not 'neither', make pointed end(s) for out-of- range values. These are set for a given colormap using the colormap set_under and set_over methods.
extendfrac	{*None*, 'auto', length, lengths} If set to *None*, both the minimum and maximum triangular colorbar extensions with have a length of 5% of the interior colorbar length (this is the default setting). If set to 'auto', makes the triangular colorbar extensions the same lengths as the interior boxes (when *spacing* is set to 'uniform') or the same lengths as the respective adjacent interior boxes (when *spacing* is set to 'proportional'). If a scalar, indicates the length of both the minimum and maximum triangular colorbar extensions as a fraction of the interior colorbar length. A two-element sequence of fractions may also be given, indicating the lengths of the minimum and maximum colorbar extensions respectively as a fraction of the interior colorbar length.
extendrect	bool If *False* the minimum and maximum colorbar extensions will be triangular (the default). If *True* the extensions will be rectangular.
spacing	{'uniform', 'proportional'} Uniform spacing gives each discrete color the same space; proportional makes the space proportional to the data interval.
ticks	*None* or list of ticks or Locator If None, ticks are determined automatically from the input.
format	None or str or Formatter If None, *ScalarFormatter* is used. If a format string is given, e.g., '%.3f', that is used. An alternative *Formatter* may be given instead.
drawedges	bool Whether to draw lines at color boundaries.
label	str The label on the colorbar's long axis.

The following will probably be useful only in the context of indexed colors (that is, when the mappable has norm=NoNorm()), or other unusual circumstances.

Property	Description
boundaries	None or a sequence
values	None or a sequence which must be of length 1 less than the sequence of *boundaries*. For each region delimited by adjacent entries in *boundaries*, the color mapped to the corresponding value in values will be used.

If *mappable* is a *ContourSet*, its *extend* kwarg is included automatically.

The *shrink* kwarg provides a simple way to scale the colorbar with respect to the axes. Note that if *cax* is specified, it determines the size of the colorbar and *shrink* and *aspect* kwargs are ignored.

For more precise control, you can manually specify the positions of the axes objects in which the mappable and the colorbar are drawn. In this case, do not use any of the axes properties kwargs.

It is known that some vector graphics viewers (svg and pdf) renders white gaps between segments of the colorbar. This is due to bugs in the viewers, not Matplotlib. As a workaround, the colorbar can be rendered with overlapping segments:

```
cbar = colorbar()
cbar.solids.set_edgecolor("face")
draw()
```

However this has negative consequences in other circumstances, e.g. with semi-transparent images (alpha < 1) and colorbar extensions; therefore, this workaround is not used by default (see issue #1188).

`contains`(*self*, *mouseevent*)
Test whether the mouse event occurred on the figure.

Returns

bool, {}

`delaxes`(*self*, *ax*)
Remove the *Axes* *ax* from the figure; update the current axes.

`property dpi`
The resolution in dots per inch.

`draw`(*self*, *renderer*)
Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

Parameters

renderer

[*RendererBase* subclass.]

Notes

This method is overridden in the Artist subclasses.

`draw_artist`(*self*, *a*)
Draw *Artist* instance *a* only.

This can only be called after the figure has been drawn.

`execute_constrained_layout`(*self*, *renderer=None*)
Use `layoutbox` to determine pos positions within axes.

See also *set_constrained_layout_pads*.

`figimage`(*self*, *X*, *xo=0*, *yo=0*, *alpha=None*, *norm=None*, *cmap=None*, *vmin=None*, *vmax=None*, *origin=None*, *resize=False*, ***kwargs*)
Add a non-resampled image to the figure.

The image is attached to the lower or upper left corner depending on *origin*.

Parameters

X

The image data. This is an array of one of the following shapes:

- MxN: luminance (grayscale) values
- MxNx3: RGB values
- MxNx4: RGBA values

xo, yo

[int] The *x*/*y* image offset in pixels.

alpha

[None or float] The alpha blending value.

norm

[*matplotlib.colors.Normalize*] A *Normalize* instance to map the luminance to the interval [0, 1].

cmap

[str or *matplotlib.colors.Colormap*, default: `rcParams["image.cmap"]` (default: `'viridis'`)] The colormap to use.

vmin, vmax

[float] If *norm* is not given, these values set the data limits for the colormap.

origin

[{'upper', 'lower'}, default: `rcParams["image.origin"]` (default: `'upper'`)] Indicates where the [0, 0] index of the array is in the upper left or lower left corner of the axes.

resize

[bool] If *True*, resize the figure to match the given image size.

Returns

matplotlib.image.FigureImage

Other Parameters

****kwargs**

Additional kwargs are *Artist* kwargs passed on to *FigureImage*.

Notes

figimage complements the axes image (*imshow*) which will be resampled to fit the current axes. If you want a resampled image to fill the entire figure, you can define an *Axes* with extent [0, 0, 1, 1].

Examples

```
f = plt.figure()
nx = int(f.get_figwidth() * f.dpi)
ny = int(f.get_figheight() * f.dpi)
data = np.random.random((ny, nx))
f.figimage(data)
plt.show()
```

`property frameon`
Return the figure's background patch visibility, i.e. whether the figure background will be drawn. Equivalent to `Figure.patch.get_visible()`.

`gca`(*self*, ***kwargs*)
Get the current axes, creating one if necessary.

The following kwargs are supported for ensuring the returned axes adheres to the given projection etc., and for axes creation if the active axes does not exist:

Property	Description
adjustable	{'box', 'datalim'}
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and r
alpha	float or None
anchor	2-tuple of floats or {'C', 'SW', 'S', 'SE', ...}
animated	bool
aspect	{'auto'} or num
autoscale_on	bool
autoscalex_on	bool
autoscaley_on	bool
axes_locator	Callable[[Axes, Renderer], Bbox]
axisbelow	bool or 'line'
box_aspect	None, or a number
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
contains	unknown
facecolor or fc	color
figure	*Figure*
frame_on	bool
gid	str
in_layout	bool
label	object
navigate	bool
navigate_mode	unknown
path_effects	*AbstractPathEffect*
picker	None or bool or callable
position	[left, bottom, width, height] or *Bbox*
prop_cycle	unknown
rasterization_zorder	float or None
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
title	str
transform	*Transform*
url	str
visible	bool
xbound	unknown
xlabel	str
xlim	(bottom: float, top: float)
xmargin	float greater than -0.5
xscale	{"linear", "log", "symlog", "logit", ...}
xticklabels	unknown
xticks	unknown

Table 136 – continued from previous page

Property	Description
ybound	unknown
ylabel	str
ylim	(bottom: float, top: float)
ymargin	float greater than -0.5
yscale	{"linear", "log", "symlog", "logit", ...}
yticklabels	unknown
yticks	unknown
zorder	float

`get_axes`(*self*)
Return a list of axes in the Figure. You can access and modify the axes in the Figure through this list.

Do not modify the list itself. Instead, use *add_axes*, *add_subplot* or *delaxes* to add or remove an axes.

Note: This is equivalent to the property *axes*.

`get_children`(*self*)
Get a list of artists contained in the figure.

`get_constrained_layout`(*self*)
Return whether constrained layout is being used.

See *Constrained Layout Guide*.

`get_constrained_layout_pads`(*self*, *relative=False*)
Get padding for `constrained_layout`.

Returns a list of `w_pad, h_pad` in inches and `wspace` and `hspace` as fractions of the subplot.

See *Constrained Layout Guide*.

Parameters

relative

[bool] If `True`, then convert from inches to figure relative.

`get_default_bbox_extra_artists`(*self*)

`get_dpi`(*self*)
Return the resolution in dots per inch as a float.

`get_edgecolor`(*self*)
Get the edge color of the Figure rectangle.

`get_facecolor`(*self*)
Get the face color of the Figure rectangle.

get_figheight(*self*)
Return the figure height in inches.

get_figwidth(*self*)
Return the figure width in inches.

get_frameon(*self*)
Return the figure's background patch visibility, i.e. whether the figure background will be drawn. Equivalent to `Figure.patch.get_visible()`.

get_size_inches(*self*)
Return the current size of the figure in inches.

Returns

ndarray

The size (width, height) of the figure in inches.

See also:

matplotlib.figure.Figure.set_size_inches

matplotlib.figure.Figure.get_figwidth

matplotlib.figure.Figure.get_figheight

Notes

The size in pixels can be obtained by multiplying with *Figure.dpi*.

get_tight_layout(*self*)
Return whether *tight_layout* is called when drawing.

get_tightbbox(*self*, *renderer*, *bbox_extra_artists=None*)
Return a (tight) bounding box of the figure in inches.

Artists that have `artist.set_in_layout(False)` are not included in the bbox.

Parameters

renderer

[*RendererBase* subclass] renderer that will be used to draw the figures (i.e. `fig.canvas.get_renderer()`)

bbox_extra_artists

[list of *Artist* or `None`] List of artists to include in the tight bounding box. If `None` (default), then all artist children of each axes are included in the tight bounding box.

Returns

BboxBase

containing the bounding box (in figure inches).

`get_window_extent`(*self*, **args*, ***kwargs*)

Return the figure bounding box in display space. Arguments are ignored.

`ginput`(*self*, *n=1*, *timeout=30*, *show_clicks=True*, *mouse_add=<MouseButton.LEFT: 1>*, *mouse_pop=<MouseButton.RIGHT: 3>*, *mouse_stop=<MouseButton.MIDDLE: 2>*)

Blocking call to interact with a figure.

Wait until the user clicks *n* times on the figure, and return the coordinates of each click in a list.

There are three possible interactions:

- Add a point.
- Remove the most recently added point.
- Stop the interaction and return the points added so far.

The actions are assigned to mouse buttons via the arguments *mouse_add*, *mouse_pop* and *mouse_stop*.

Parameters

n

[int, default: 1] Number of mouse clicks to accumulate. If negative, accumulate clicks until the input is terminated manually.

timeout

[float, default: 30 seconds] Number of seconds to wait before timing out. If zero or negative will never timeout.

show_clicks

[bool, default: True] If True, show a red cross at the location of each click.

mouse_add

[*MouseButton* or None, default: *MouseButton.LEFT*] Mouse button used to add points.

mouse_pop

[*MouseButton* or None, default: *MouseButton.RIGHT*] Mouse button used to remove the most recently added point.

mouse_stop

[*MouseButton* or None, default: *MouseButton.MIDDLE*] Mouse button used to stop input.

Returns

list of tuples

A list of the clicked (x, y) coordinates.

Notes

The keyboard can also be used to select points in case your mouse does not have one or more of the buttons. The delete and backspace keys act like right clicking (i.e., remove last point), the enter key terminates input and any other key (not already used by the window manager) selects a point.

`init_layoutbox`(*self*)
Initialize the layoutbox for use in constrained_layout.

`legend`(*self*, **args*, ***kwargs*)
Place a legend on the figure.

To make a legend from existing artists on every axes:

```
legend()
```

To make a legend for a list of lines and labels:

```
legend(
    (line1, line2, line3),
    ('label1', 'label2', 'label3'),
    loc='upper right')
```

These can also be specified by keyword:

```
legend(
    handles=(line1, line2, line3),
    labels=('label1', 'label2', 'label3'),
    loc='upper right')
```

Parameters

handles

[list of *Artist*, optional] A list of Artists (lines, patches) to be added to the legend. Use this together with *labels*, if you need full control on what is shown in the legend and the automatic mechanism described above is not sufficient.

The length of handles and labels should be the same in this case. If they are not, they are truncated to the smaller length.

labels

[list of str, optional] A list of labels to show next to the artists. Use this together with *handles*, if you need full control on what is shown in the legend and the automatic mechanism described above is not sufficient.

Returns

Legend

Other Parameters

loc

[str or pair of floats, default: `rcParams["legend.loc"]` (default: `'best'`) ('best' for axes, 'upper right' for figures)] The location of the legend.

The strings `'upper left'`, `'upper right'`, `'lower left'`, `'lower right'` place the legend at the corresponding corner of the axes/figure.

The strings `'upper center'`, `'lower center'`, `'center left'`, `'center right'` place the legend at the center of the corresponding edge of the axes/figure.

The string `'center'` places the legend at the center of the axes/figure.

The string `'best'` places the legend at the location, among the nine locations defined so far, with the minimum overlap with other drawn artists. This option can be quite slow for plots with large amounts of data; your plotting speed may benefit from providing a specific location.

The location can also be a 2-tuple giving the coordinates of the lower-left corner of the legend in axes coordinates (in which case *bbox_to_anchor* will be ignored).

For back-compatibility, `'center right'` (but no other location) can also be spelled `'right'`, and each "string" locations can also be given as a numeric value:

Location String	Location Code
'best'	0
'upper right'	1
'upper left'	2
'lower left'	3
'lower right'	4
'right'	5
'center left'	6
'center right'	7
'lower center'	8
'upper center'	9
'center'	10

bbox_to_anchor

[*BboxBase*, 2-tuple, or 4-tuple of floats] Box that is used to position the legend in conjunction with *loc*. Defaults to `axes.bbox` (if called as a method to *Axes.legend*) or `figure.bbox` (if *Figure.legend*). This argument allows arbitrary placement of the legend.

Bbox coordinates are interpreted in the coordinate system given by *bbox_transform*, with the default transform Axes or Figure coordinates, depending on which `legend` is called.

If a 4-tuple or *BboxBase* is given, then it specifies the bbox `(x, y, width, height)` that the legend is placed in. To put the legend in the best location in the bottom right quadrant of the axes (or figure):

```
loc='best', bbox_to_anchor=(0.5, 0., 0.5, 0.5)
```

A 2-tuple `(x, y)` places the corner of the legend specified by *loc* at x, y. For example, to put the legend's upper right-hand corner in the center of the axes (or figure) the following keywords can be used:

```
loc='upper right', bbox_to_anchor=(0.5, 0.5)
```

ncol

[int, default: 1] The number of columns that the legend has.

prop

[None or *matplotlib.font_manager.FontProperties* or dict] The font properties of the legend. If None (default), the current *matplotlib.rcParams* will be used.

fontsize

[int or {'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large'}] The font size of the legend. If the value is numeric the size will be the absolute font size in points. String values are relative to the current default font size. This argument is only used if *prop* is not specified.

labelcolor

[str or list] Sets the color of the text in the legend. Can be a valid color string (for example, 'red'), or a list of color strings. The labelcolor can also be made to match the color of the line or marker using 'linecolor', 'markerfacecolor' (or 'mfc'), or 'markeredgecolor' (or 'mec').

numpoints

[int, default: `rcParams["legend.numpoints"]` (default: `1`)] The number of marker points in the legend when creating a legend entry for a *Line2D* (line).

scatterpoints

[int, default: `rcParams["legend.scatterpoints"]` (default: `1`)] The number of marker points in the legend when creating a legend entry for a *PathCollection* (scatter plot).

scatteryoffsets

[iterable of floats, default: `[0.375, 0.5, 0.3125]`] The vertical offset (relative to the font size) for the markers created for a scatter plot legend entry. 0.0 is at the base the legend text, and 1.0 is at the top. To draw all markers at the same height, set to `[0.5]`.

markerscale

[float, default: `rcParams["legend.markerscale"]` (default: `1.0`)] The relative size of legend markers compared with the originally drawn ones.

markerfirst

[bool, default: True] If *True*, legend marker is placed to the left of the legend label. If *False*, legend marker is placed to the right of the legend label.

frameon

[bool, default: `rcParams["legend.frameon"]` (default: `True`)] Whether the legend should be drawn on a patch (frame).

fancybox

[bool, default: `rcParams["legend.fancybox"]` (default: `True`)] Whether round edges should be enabled around the *FancyBboxPatch* which makes up the legend's background.

shadow

[bool, default: rcParams["legend.shadow"] (default: False)] Whether to draw a shadow behind the legend.

framealpha

[float, default: rcParams["legend.framealpha"] (default: 0.8)] The alpha transparency of the legend's background. If *shadow* is activated and *framealpha* is None, the default value is ignored.

facecolor

["inherit" or color, default: rcParams["legend.facecolor"] (default: 'inherit')] The legend's background color. If "inherit", use rcParams["axes.facecolor"] (default: 'white').

edgecolor

["inherit" or color, default: rcParams["legend.edgecolor"] (default: '0.8')] The legend's background patch edge color. If "inherit", use take rcParams["axes.edgecolor"] (default: 'black').

mode

[{"expand", None}] If *mode* is set to "expand" the legend will be horizontally expanded to fill the axes area (or *bbox_to_anchor* if defines the legend's size).

bbox_transform

[None or *matplotlib.transforms.Transform*] The transform for the bounding box (*bbox_to_anchor*). For a value of None (default) the Axes' transAxes transform will be used.

title

[str or None] The legend's title. Default is no title (None).

title_fontsize

[int or {'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large'}, default: rcParams["legend.title_fontsize"] (default: None)] The font size of the legend's title.

borderpad

[float, default: rcParams["legend.borderpad"] (default: 0.4)] The fractional whitespace inside the legend border, in font-size units.

labelspacing

[float, default: rcParams["legend.labelspacing"] (default: 0.5)] The vertical space between the legend entries, in font-size units.

handlelength

[float, default: `rcParams["legend.handlelength"]` (default: `2.0`)] The length of the legend handles, in font-size units.

handletextpad

[float, default: `rcParams["legend.handletextpad"]` (default: `0.8`)] The pad between the legend handle and text, in font-size units.

borderaxespad

[float, default: `rcParams["legend.borderaxespad"]` (default: `0.5`)] The pad between the axes and legend border, in font-size units.

columnspacing

[float, default: `rcParams["legend.columnspacing"]` (default: `2.0`)] The spacing between columns, in font-size units.

handler_map

[dict or None] The custom dictionary mapping instances or types to a legend handler. This *handler_map* updates the default handler map found at *matplotlib.legend.Legend.get_legend_handler_map*.

Notes

Some artists are not supported by this function. See *Legend guide* for details.

`savefig`(*self*, *fname*, *, *transparent=None*, **kwargs*)

Save the current figure.

Call signature:

```
savefig(fname, dpi=None, facecolor='w', edgecolor='w',
        orientation='portrait', papertype=None, format=None,
        transparent=False, bbox_inches=None, pad_inches=0.1,
        frameon=None, metadata=None)
```

The available output formats depend on the backend being used.

Parameters

fname

[str or path-like or file-like] A path, or a Python file-like object, or possibly some backend-dependent object such as *matplotlib.backends.backend_pdf.PdfPages*.

If *format* is set, it determines the output format, and the file is saved as *fname*. Note that *fname* is used verbatim, and there

is no attempt to make the extension, if any, of *fname* match *format*, and no extension is appended.

If *format* is not set, then the format is inferred from the extension of *fname*, if there is one. If *format* is not set and *fname* has no extension, then the file is saved with `rcParams["savefig.format"]` (default: `'png'`) and the appropriate extension is appended to *fname*.

Other Parameters

dpi

[float or 'figure', default: `rcParams["savefig.dpi"]` (default: `'figure'`)] The resolution in dots per inch. If 'figure', use the figure's dpi value.

quality

[int, default: `rcParams["savefig.jpeg_quality"]` (default: `95`)] Applicable only if *format* is 'jpg' or 'jpeg', ignored otherwise.

The image quality, on a scale from 1 (worst) to 95 (best). Values above 95 should be avoided; 100 disables portions of the JPEG compression algorithm, and results in large files with hardly any gain in image quality.

This parameter is deprecated.

optimize

[bool, default: False] Applicable only if *format* is 'jpg' or 'jpeg', ignored otherwise.

Whether the encoder should make an extra pass over the image in order to select optimal encoder settings.

This parameter is deprecated.

progressive

[bool, default: False] Applicable only if *format* is 'jpg' or 'jpeg', ignored otherwise.

Whether the image should be stored as a progressive JPEG file.

This parameter is deprecated.

facecolor

[color or 'auto', default: `rcParams["savefig.facecolor"]` (default: `'auto'`)] The facecolor of the figure. If 'auto', use the current figure facecolor.

edgecolor

[color or 'auto', default: `rcParams["savefig.edgecolor"]` (default: `'auto'`)] The edgecolor of the figure. If 'auto', use the current figure edgecolor.

orientation

[{'landscape', 'portrait'}] Currently only supported by the postscript backend.

papertype

[str] One of 'letter', 'legal', 'executive', 'ledger', 'a0' through 'a10', 'b0' through 'b10'. Only supported for postscript output.

format

[str] The file format, e.g. 'png', 'pdf', 'svg', ... The behavior when this is unset is documented under *fname*.

transparent

[bool] If *True*, the axes patches will all be transparent; the figure patch will also be transparent unless facecolor and/or edgecolor are specified via kwargs. This is useful, for example, for displaying a plot on top of a colored background on a web page. The transparency of these patches will be restored to their original values upon exit of this function.

bbox_inches

[str or *Bbox*, default: `rcParams["savefig.bbox"]` (default: `None`)] Bounding box in inches: only the given portion of the figure is saved. If 'tight', try to figure out the tight bbox of the figure.

pad_inches

[float, default: `rcParams["savefig.pad_inches"]` (default: `0.1`)] Amount of padding around the figure when bbox_inches is 'tight'.

bbox_extra_artists

[list of *Artist*, optional] A list of extra artists that will be considered when the tight bbox is calculated.

backend

[str, optional] Use a non-default backend to render the file, e.g. to render a png file with the "cairo" backend rather than the default "agg", or a pdf file with the "pgf" backend rather than the default "pdf". Note that the default backend is normally sufficient. See *The builtin backends* for a list of valid backends for each file format. Custom backends can be referenced as "module://...".

metadata

[dict, optional] Key/value pairs to store in the image metadata. The supported keys and defaults depend on the image format and backend:

- 'png' with Agg backend: See the parameter `metadata` of *print_png*.
- 'pdf' with pdf backend: See the parameter `metadata` of *PdfPages*.
- 'svg' with svg backend: See the parameter `metadata` of *print_svg*.
- 'eps' and 'ps' with PS backend: Only 'Creator' is supported.

pil_kwargs

[dict, optional] Additional keyword arguments that are passed to `PIL.Image.Image.save` when saving the figure.

`sca`(*self*, *a*)
Set the current axes to be *a* and return *a*.

`set_canvas`(*self*, *canvas*)
Set the canvas that contains the figure

Parameters

canvas

[FigureCanvas]

`set_constrained_layout`(*self*, *constrained*)
Set whether `constrained_layout` is used upon drawing. If None, `rcParams["figure.constrained_layout.use"]` (default: `False`) value will be used.

When providing a dict containing the keys `w_pad`, `h_pad` the default `constrained_layout` paddings will be overridden. These pads are in inches and default to 3.0/72.0. `w_pad` is the width padding and `h_pad` is the height padding.

See *Constrained Layout Guide*.

Parameters

constrained

[bool or dict or None]

`set_constrained_layout_pads`(*self*, ***kwargs*)
Set padding for `constrained_layout`. Note the kwargs can be passed as a dictionary `fig.set_constrained_layout(**paddict)`.

See *Constrained Layout Guide*.

Parameters

w_pad

[float] Width padding in inches. This is the pad around axes and is meant to make sure there is enough room for fonts to look good. Defaults to 3 pts = 0.04167 inches

h_pad

[float] Height padding in inches. Defaults to 3 pts.

wspace

[float] Width padding between subplots, expressed as a fraction of the subplot width. The total padding ends up being w_pad + wspace.

hspace

[float] Height padding between subplots, expressed as a fraction of the subplot width. The total padding ends up being h_pad + hspace.

`set_dpi`(*self*, *val*)
Set the resolution of the figure in dots-per-inch.

Parameters

val

[float]

`set_edgecolor`(*self*, *color*)
Set the edge color of the Figure rectangle.

Parameters

color

[color]

`set_facecolor`(*self*, *color*)
Set the face color of the Figure rectangle.

Parameters

color

[color]

`set_figheight`(*self*, *val*, *forward=True*)
Set the height of the figure in inches.

Parameters

val

[float]

forward

[bool] See *set_size_inches*.

See also:

matplotlib.figure.Figure.set_figwidth

matplotlib.figure.Figure.set_size_inches

set_figwidth(*self*, *val*, *forward=True*)

Set the width of the figure in inches.

Parameters

val

[float]

forward

[bool] See *set_size_inches*.

See also:

matplotlib.figure.Figure.set_figheight

matplotlib.figure.Figure.set_size_inches

set_frameon(*self*, *b*)

Set the figure's background patch visibility, i.e. whether the figure background will be drawn. Equivalent to `Figure.patch.set_visible()`.

Parameters

b

[bool]

set_size_inches(*self*, *w*, *h=None*, *forward=True*)

Set the figure size in inches.

Call signatures:

```
fig.set_size_inches(w, h)  # OR
fig.set_size_inches((w, h))
```

Parameters

w

[(float, float) or float] Width and height in inches (if height not specified as a separate argument) or width.

h

[float] Height in inches.

forward

[bool, default: True] If `True`, the canvas size is automatically updated, e.g., you can resize the figure window from the shell.

See also:

matplotlib.figure.Figure.get_size_inches

matplotlib.figure.Figure.set_figwidth

matplotlib.figure.Figure.set_figheight

Notes

To transform from pixels to inches divide by *Figure.dpi*.

`set_tight_layout`(*self*, *tight*)

Set whether and how *tight_layout* is called when drawing.

Parameters

tight

[bool or dict with keys "pad", "w_pad", "h_pad", "rect" or None] If a bool, sets whether to call *tight_layout* upon drawing. If `None`, use the `figure.autolayout` rcparam instead. If a dict, pass it as kwargs to *tight_layout*, overriding the default paddings.

`show`(*self*, *warn=True*)

If using a GUI backend with pyplot, display the figure window.

If the figure was not created using *figure*, it will lack a *FigureManagerBase*, and this method will raise an AttributeError.

Warning: This does not manage an GUI event loop. Consequently, the figure may only be shown briefly or not shown at all if you or your environment are not managing an event loop.

Proper use cases for *Figure.show* include running this from a GUI application or an IPython shell.

If you're running a pure python shell or executing a non-GUI python script, you should use *matplotlib.pyplot.show* instead, which takes care of managing the event loop for you.

Parameters

warn

[bool, default: True] If `True` and we are not running headless (i.e. on Linux with an unset DISPLAY), issue warning when called on a non-GUI backend.

`subplot_mosaic`(*self*, *layout*, *, *subplot_kw=None*, *gridspec_kw=None*, *empty_sentinel='.'*)

Build a layout of Axes based on ASCII art or nested lists.

This is a helper function to build complex GridSpec layouts visually.

Note: This API is provisional and may be revised in the future based on early user feedback.

Parameters

layout

[list of list of {hashable or nested} or str] A visual layout of how you want your Axes to be arranged labeled as strings. For example

```
x = [['A panel', 'A panel', 'edge'],
     ['C panel', '.',       'edge']]
```

Produces 4 axes:

- 'A panel' which is 1 row high and spans the first two columns
- 'edge' which is 2 rows high and is on the right edge
- 'C panel' which in 1 row and 1 column wide in the bottom left
- a blank space 1 row and 1 column wide in the bottom center

Any of the entries in the layout can be a list of lists of the same form to create nested layouts.

If input is a str, then it must be of the form

```
'''
AAE
C.E
'''
```

where each character is a column and each line is a row. This only allows only single character Axes labels and does not allow nesting but is very terse.

subplot_kw

[dict, optional] Dictionary with keywords passed to the *Figure.add_subplot* call used to create each subplot.

gridspec_kw

[dict, optional] Dictionary with keywords passed to the *GridSpec* constructor used to create the grid the subplots are placed on.

empty_sentinel

[object, optional] Entry in the layout to mean "leave this space empty". Defaults to `'.'`. Note, if *layout* is a string, it is processed via `inspect.cleandoc` to remove leading white space, which may interfere with using white-space as the empty sentinel.

Returns

dict[label, Axes]

A dictionary mapping the labels to the Axes objects.

`subplots`(*self*, *nrows=1*, *ncols=1*, ***, *sharex=False*, *sharey=False*, *squeeze=True*, *subplot_kw=None*, *gridspec_kw=None*)

Add a set of subplots to this figure.

This utility wrapper makes it convenient to create common layouts of subplots in a single call.

Parameters

nrows, ncols

[int, default: 1] Number of rows/columns of the subplot grid.

sharex, sharey

[bool or {'none', 'all', 'row', 'col'}, default: False] Controls sharing of properties among x (*sharex*) or y (*sharey*) axes:

- True or 'all': x- or y-axis will be shared among all subplots.
- False or 'none': each subplot x- or y-axis will be independent.

- 'row': each subplot row will share an x- or y-axis.
- 'col': each subplot column will share an x- or y-axis.

When subplots have a shared x-axis along a column, only the x tick labels of the bottom subplot are created. Similarly, when subplots have a shared y-axis along a row, only the y tick labels of the first column subplot are created. To later turn other subplots' ticklabels on, use *tick_params*.

squeeze

[bool, default: True]

- If True, extra dimensions are squeezed out from the returned array of Axes:
  - if only one subplot is constructed (nrows=ncols=1), the resulting single Axes object is returned as a scalar.
  - for Nx1 or 1xM subplots, the returned object is a 1D numpy object array of Axes objects.
  - for NxM, subplots with N>1 and M>1 are returned as a 2D array.
- If False, no squeezing at all is done: the returned Axes object is always a 2D array containing Axes instances, even if it ends up being 1x1.

subplot_kw

[dict, optional] Dict with keywords passed to the *Figure.add_subplot* call used to create each subplot.

gridspec_kw

[dict, optional] Dict with keywords passed to the *GridSpec* constructor used to create the grid the subplots are placed on.

Returns

Axes **or array of Axes**

Either a single *Axes* object or an array of Axes objects if more than one subplot was created. The dimensions of the resulting array can be controlled with the *squeeze* keyword, see above.

See also:

pyplot.subplots

Figure.add_subplot

pyplot.subplot

Examples

```
# First create some toy data:
x = np.linspace(0, 2*np.pi, 400)
y = np.sin(x**2)

# Create a figure
plt.figure()

# Create a subplot
ax = fig.subplots()
ax.plot(x, y)
ax.set_title('Simple plot')

# Create two subplots and unpack the output array immediately
ax1, ax2 = fig.subplots(1, 2, sharey=True)
ax1.plot(x, y)
ax1.set_title('Sharing Y axis')
ax2.scatter(x, y)

# Create four polar axes and access them through the returned array
axes = fig.subplots(2, 2, subplot_kw=dict(polar=True))
axes[0, 0].plot(x, y)
axes[1, 1].scatter(x, y)

# Share a X axis with each column of subplots
fig.subplots(2, 2, sharex='col')

# Share a Y axis with each row of subplots
fig.subplots(2, 2, sharey='row')

# Share both X and Y axes with all subplots
fig.subplots(2, 2, sharex='all', sharey='all')

# Note that this is the same as
fig.subplots(2, 2, sharex=True, sharey=True)
```

subplots_adjust(*self*, *left=None*, *bottom=None*, *right=None*, *top=None*, *wspace=None*, *hspace=None*)

Adjust the subplot layout parameters.

Unset parameters are left unmodified; initial values are given by `rcParams["figure.subplot.[name]"]`.

Parameters

left

[float, optional] The position of the left edge of the subplots, as a fraction of the figure width.

right

[float, optional] The position of the right edge of the subplots,

as a fraction of the figure width.

bottom

[float, optional] The position of the bottom edge of the subplots, as a fraction of the figure height.

top

[float, optional] The position of the top edge of the subplots, as a fraction of the figure height.

wspace

[float, optional] The width of the padding between subplots, as a fraction of the average axes width.

hspace

[float, optional] The height of the padding between subplots, as a fraction of the average axes height.

`suptitle`(*self*, *t*, ***kwargs*)

Add a centered title to the figure.

Parameters

t

[str] The title text.

x

[float, default 0.5] The x location of the text in figure coordinates.

y

[float, default 0.98] The y location of the text in figure coordinates.

horizontalalignment, ha

[{'center', 'left', right'}, default: 'center'] The horizontal alignment of the text relative to (*x*, *y*).

verticalalignment, va

[{'top', 'center', 'bottom', 'baseline'}, default: 'top'] The vertical alignment of the text relative to (*x*, *y*).

fontsize, size

[default: `rcParams["figure.titlesize"]` (default: `'large'`)] The font size of the text. See *Text.set_size* for possible values.

fontweight, weight

[default: rcParams["figure.titleweight"] (default: 'normal')] The font weight of the text. See *Text.set_weight* for possible values.

Returns

text

The *Text* instance of the title.

Other Parameters

fontproperties

[None or dict, optional] A dict of font properties. If *fontproperties* is given the default values for font size and weight are taken from the *FontProperties* defaults. rcParams["figure.titlesize"] (default: 'large') and rcParams["figure.titleweight"] (default: 'normal') are ignored in this case.

****kwargs**

Additional kwargs are *matplotlib.text.Text* properties.

Examples

```
>>> fig.suptitle('This is the figure title', fontsize=12)
```

text(*self*, *x*, *y*, *s*, *fontdict=None*, ***kwargs*)

Add text to figure.

Parameters

x, y

[float] The position to place the text. By default, this is in figure coordinates, floats in [0, 1]. The coordinate system can be changed using the *transform* keyword.

s

[str] The text string.

fontdict

[dict, optional] A dictionary to override the default text properties. If not given, the defaults are determined by rcParams["font.*"]. Properties passed as *kwargs* override the corresponding ones given in *fontdict*.

Returns

Text

Other Parameters

**kwargs

[*Text* properties] Other miscellaneous text parameters.

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array an
alpha	float or None
animated	bool
backgroundcolor	color
bbox	dict with properties for *patches.FancyBboxPatch*
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color or c	color
contains	unknown
figure	*Figure*
fontfamily or family	{FONTNAME, 'serif', 'sans-serif', 'cursive', 'fantasy', '
fontproperties or font or font_properties	*font_manager.FontProperties* or `str` or `pathlib.Path`
fontsize or size	float or {'xx-small', 'x-small', 'small', 'medium', 'large',
fontstretch or stretch	{a numeric value in range 0-1000, 'ultra-condensed',
fontstyle or style	{'normal', 'italic', 'oblique'}
fontvariant or variant	{'normal', 'small-caps'}
fontweight or weight	{a numeric value in range 0-1000, 'ultralight', 'light',
gid	str
horizontalalignment or ha	{'center', 'right', 'left'}
in_layout	bool
label	object
linespacing	float (multiple of font size)
multialignment or ma	{'left', 'right', 'center'}
path_effects	*AbstractPathEffect*
picker	None or bool or callable
position	(float, float)
rasterized	bool or None
rotation	float or {'vertical', 'horizontal'}
rotation_mode	{None, 'default', 'anchor'}
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
text	object
transform	*Transform*
url	str
usetex	bool or None
verticalalignment or va	{'center', 'top', 'bottom', 'baseline', 'center_baseline'}

Tabl

Property	Description
visible	bool
wrap	bool
x	float
y	float
zorder	float

See also:

Axes.text

pyplot.text

`tight_layout`(*self*, *renderer=<deprecated parameter>*, *pad=1.08*, *h_pad=None*, *w_pad=None*, *rect=None*)

Adjust the padding between and around subplots.

To exclude an artist on the axes from the bounding box calculation that determines the subplot parameters (i.e. legend, or annotation), set `a.set_in_layout(False)` for that artist.

Parameters

renderer

[subclass of *RendererBase*, optional] Defaults to the renderer for the figure. Deprecated.

pad

[float, default: 1.08] Padding between the figure edge and the edges of subplots, as a fraction of the font size.

h_pad, w_pad

[float, default: *pad*] Padding (height/width) between edges of adjacent subplots, as a fraction of the font size.

rect

[tuple (left, bottom, right, top), default: (0, 0, 1, 1)] A rectangle in normalized figure coordinates into which the whole subplots area (including labels) will fit.

See also:

Figure.set_tight_layout

pyplot.tight_layout

waitforbuttonpress(*self*, *timeout=- 1*)

Blocking call to interact with the figure.

Wait for user input and return True if a key was pressed, False if a mouse button was pressed and None if no input was given within *timeout* seconds. Negative values deactivate *timeout*.

Examples using matplotlib.figure.Figure

- sphx_glr_gallery_user_interfaces_embedding_in_gtk3_panzoom_sgskip.py
- sphx_glr_gallery_user_interfaces_embedding_in_gtk3_sgskip.py
- sphx_glr_gallery_user_interfaces_embedding_in_qt_sgskip.py
- sphx_glr_gallery_user_interfaces_embedding_in_tk_sgskip.py
- sphx_glr_gallery_user_interfaces_embedding_in_wx2_sgskip.py
- sphx_glr_gallery_user_interfaces_embedding_in_wx3_sgskip.py
- sphx_glr_gallery_user_interfaces_embedding_in_wx4_sgskip.py
- sphx_glr_gallery_user_interfaces_embedding_in_wx5_sgskip.py
- sphx_glr_gallery_user_interfaces_embedding_webagg_sgskip.py
- sphx_glr_gallery_user_interfaces_fourier_demo_wx_sgskip.py
- sphx_glr_gallery_user_interfaces_gtk_spreadsheet_sgskip.py
- sphx_glr_gallery_user_interfaces_mathtext_wx_sgskip.py
- sphx_glr_gallery_user_interfaces_mpl_with_glade3_sgskip.py
- sphx_glr_gallery_user_interfaces_wxcursor_demo_sgskip.py

matplotlib.figure.SubplotParams

class matplotlib.figure.**SubplotParams**(*left=None*, *bottom=None*, *right=None*, *top=None*, *wspace=None*, *hspace=None*)

Bases: object

A class to hold the parameters for a subplot.

Defaults are given by rcParams["figure.subplot.[name]"].

Parameters

left

[float] The position of the left edge of the subplots, as a fraction of the figure width.

right

[float] The position of the right edge of the subplots, as a fraction of the figure width.

bottom

[float] The position of the bottom edge of the subplots, as a fraction of the figure height.

top

[float] The position of the top edge of the subplots, as a fraction of the figure height.

wspace

[float] The width of the padding between subplots, as a fraction of the average axes width.

hspace

[float] The height of the padding between subplots, as a fraction of the average axes height.

`__dict__ = mappingproxy({'__module__': 'matplotlib.figure', '__doc__': '\n A class to hol`

`__init__`(*self*, *left=None*, *bottom=None*, *right=None*, *top=None*, *wspace=None*, *hspace=None*)

Defaults are given by `rcParams["figure.subplot.[name]"]`.

Parameters

left

[float] The position of the left edge of the subplots, as a fraction of the figure width.

right

[float] The position of the right edge of the subplots, as a fraction of the figure width.

bottom

[float] The position of the bottom edge of the subplots, as a fraction of the figure height.

top

[float] The position of the top edge of the subplots, as a fraction of the figure height.

wspace

[float] The width of the padding between subplots, as a fraction of the average axes width.

hspace

[float] The height of the padding between subplots, as a fraction of the average axes height.

`__module__ = 'matplotlib.figure'`

`__slotnames__ = []`

`__weakref__`
list of weak references to the object (if defined)

`update`(*self*, *left=None*, *bottom=None*, *right=None*, *top=None*, *wspace=None*, *hspace=None*)
Update the dimensions of the passed parameters. *None* means unchanged.

Examples using `matplotlib.figure.SubplotParams`

- sphx_glr_gallery_pyplots_auto_subplots_adjust.py

18.23.2 Functions

figaspect	Calculate the width and height for a figure with a specified aspect ratio.

matplotlib.figure.figaspect

`matplotlib.figure.figaspect`(*arg*)
Calculate the width and height for a figure with a specified aspect ratio.

While the height is taken from `rcParams["figure.figsize"]` (default: `[6.4, 4.8]`), the width is adjusted to match the desired aspect ratio. Additionally, it is ensured that the width is in the range [4., 16.] and the height is in the range [2., 16.]. If necessary, the default height is adjusted to ensure this.

Parameters

arg

[float or 2d array] If a float, this defines the aspect ratio (i.e. the ratio height / width). In case of an array the aspect ratio is number of rows / number of columns, so that the array could be fitted in the figure undistorted.

Returns

width, height

The figure size in inches.

Notes

If you want to create an axes within the figure, that still preserves the aspect ratio, be sure to create it with equal width and height. See examples below.

Thanks to Fernando Perez for this function.

Examples

Make a figure twice as tall as it is wide:

```
w, h = figaspect(2.)
fig = Figure(figsize=(w, h))
ax = fig.add_axes([0.1, 0.1, 0.8, 0.8])
ax.imshow(A, **kwargs)
```

Make a figure with the proper aspect for an array:

```
A = rand(5, 3)
w, h = figaspect(A)
fig = Figure(figsize=(w, h))
ax = fig.add_axes([0.1, 0.1, 0.8, 0.8])
ax.imshow(A, **kwargs)
```

Examples using `matplotlib.figure.figaspect`

18.24 matplotlib.font_manager

A module for finding, managing, and using fonts across platforms.

This module provides a single *FontManager* instance that can be shared across backends and platforms. The *findfont* function returns the best TrueType (TTF) font file in the local or system font path that matches the specified *FontProperties* instance. The *FontManager* also handles Adobe Font Metrics (AFM) font files for use by the PostScript backend.

The design is based on the W3C Cascading Style Sheet, Level 1 (CSS1) font specification. Future versions may implement the Level 2 or 2.1 specifications.

class matplotlib.font_manager.**FontEntry**(*fname*='', *name*='', *style*='normal', *variant*='normal', *weight*='normal', *stretch*='normal', *size*='medium')

Bases: `object`

A class for storing Font properties. It is used when populating the font lookup dictionary.

class matplotlib.font_manager.**FontManager**(*size=None*, *weight='normal'*)

Bases: object

On import, the *FontManager* singleton instance creates a list of ttf and afm fonts and caches their *FontProperties*. The *FontManager.findfont* method does a nearest neighbor search to find the font that most closely matches the specification. If no good enough match is found, the default font is returned.

addfont(*self*, *path*)

Cache the properties of the font at *path* to make it available to the *FontManager*. The type of font is inferred from the path suffix.

Parameters

path

[str or path-like]

property **defaultFont**

findfont(*self*, *prop*, *fontext='ttf'*, *directory=None*, *fallback_to_default=True*, *rebuild_if_missing=True*)

Find a font that most closely matches the given font properties.

Parameters

prop

[str or *FontProperties*] The font properties to search for. This can be either a *FontProperties* object or a string defining a fontconfig patterns.

fontext

[{'ttf', 'afm'}, default: 'ttf'] The extension of the font file:

- 'ttf': TrueType and OpenType fonts (.ttf, .ttc, .otf)
- 'afm': Adobe Font Metrics (.afm)

directory

[str, optional] If given, only search this directory and its subdirectories.

fallback_to_default

[bool] If True, will fallback to the default font family (usually "DejaVu Sans" or "Helvetica") if the first lookup hard-fails.

rebuild_if_missing

[bool] Whether to rebuild the font cache and search again if the first match appears to point to a nonexisting font (i.e., the font cache contains outdated entries).

Returns

str

The filename of the best matching font.

Notes

This performs a nearest neighbor search. Each font is given a similarity score to the target font properties. The first font with the highest score is returned. If no matches below a certain threshold are found, the default font (usually DejaVu Sans) is returned.

The result is cached, so subsequent lookups don't have to perform the O(n) nearest neighbor search.

See the W3C Cascading Style Sheet, Level 1 documentation for a description of the font finding algorithm.

`static get_default_size()`
Return the default font size.

`get_default_weight(self)`
Return the default font weight.

`score_family(self, families, family2)`
Return a match score between the list of font families in *families* and the font family name *family2*.

An exact match at the head of the list returns 0.0.

A match further down the list will return between 0 and 1.

No match will return 1.0.

`score_size(self, size1, size2)`
Return a match score between *size1* and *size2*.

If *size2* (the size specified in the font file) is 'scalable', this function always returns 0.0, since any font size can be generated.

Otherwise, the result is the absolute distance between *size1* and *size2*, normalized so that the usual range of font sizes (6pt - 72pt) will lie between 0.0 and 1.0.

`score_stretch(self, stretch1, stretch2)`
Return a match score between *stretch1* and *stretch2*.

The result is the absolute value of the difference between the CSS numeric values of *stretch1* and *stretch2*, normalized between 0.0 and 1.0.

`score_style(self, style1, style2)`
Return a match score between *style1* and *style2*.

An exact match returns 0.0.

A match between 'italic' and 'oblique' returns 0.1.

No match returns 1.0.

`score_variant`(*self*, *variant1*, *variant2*)
Return a match score between *variant1* and *variant2*.

An exact match returns 0.0, otherwise 1.0.

`score_weight`(*self*, *weight1*, *weight2*)
Return a match score between *weight1* and *weight2*.

The result is 0.0 if both weight1 and weight 2 are given as strings and have the same value.

Otherwise, the result is the absolute value of the difference between the CSS numeric values of *weight1* and *weight2*, normalized between 0.05 and 1.0.

`set_default_weight`(*self*, *weight*)
Set the default font weight. The initial value is 'normal'.

`class matplotlib.font_manager.FontProperties`(*family=None*, *style=None*, *variant=None*, *weight=None*, *stretch=None*, *size=None*, *fname=None*)

Bases: `object`

A class for storing and manipulating font properties.

The font properties are those described in the W3C Cascading Style Sheet, Level 1 font specification. The six properties are:

- family: A list of font names in decreasing order of priority. The items may include a generic font family name, either 'serif', 'sans-serif', 'cursive', 'fantasy', or 'monospace'. In that case, the actual font to be used will be looked up from the associated rcParam.
- style: Either 'normal', 'italic' or 'oblique'.
- variant: Either 'normal' or 'small-caps'.
- stretch: A numeric value in the range 0-1000 or one of 'ultra-condensed', 'extra-condensed', 'condensed', 'semi-condensed', 'normal', 'semi-expanded', 'expanded', 'extra-expanded' or 'ultra-expanded'.
- weight: A numeric value in the range 0-1000 or one of 'ultralight', 'light', 'normal', 'regular', 'book', 'medium', 'roman', 'semibold', 'demibold', 'demi', 'bold', 'heavy', 'extra bold', 'black'.
- size: Either an relative value of 'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large' or an absolute font size, e.g., 12.

The default font property for TrueType fonts (as specified in the default rcParams) is

```
sans-serif, normal, normal, normal, normal, scalable.
```

Alternatively, a font may be specified using the absolute path to a font file, by using the *fname* kwarg. However, in this case, it is typically simpler to just pass the path (as a `pathlib.Path`, not a `str`) to the *font* kwarg of the *`Text`* object.

The preferred usage of font sizes is to use the relative values, e.g., 'large', instead of absolute font sizes, e.g., 12. This approach allows all text sizes to be made larger or smaller based on the font manager's default font size.

This class will also accept a fontconfig pattern, if it is the only argument provided. This support does not depend on fontconfig; we are merely borrowing its pattern syntax for use here.

Note that Matplotlib's internal font manager and fontconfig use a different algorithm to lookup fonts, so the results of the same pattern may be different in Matplotlib than in other applications that use fontconfig.

`copy`(*self*)
: Return a copy of self.

`get_family`(*self*)
: Return a list of font names that comprise the font family.

`get_file`(*self*)
: Return the filename of the associated font.

`get_fontconfig_pattern`(*self*)
: Get a fontconfig pattern suitable for looking up the font as specified with fontconfig's `fc-match` utility.

  This support does not depend on fontconfig; we are merely borrowing its pattern syntax for use here.

`get_name`(*self*)
: Return the name of the font that best matches the font properties.

`get_size`(*self*)
: Return the font size.

`get_size_in_points`(*self*)

`get_slant`(*self*)
: Return the font style. Values are: 'normal', 'italic' or 'oblique'.

`get_stretch`(*self*)
: Return the font stretch or width. Options are: 'ultra-condensed', 'extra-condensed', 'condensed', 'semi-condensed', 'normal', 'semi-expanded', 'expanded', 'extra-expanded', 'ultra-expanded'.

`get_style`(*self*)
: Return the font style. Values are: 'normal', 'italic' or 'oblique'.

`get_variant`(*self*)
: Return the font variant. Values are: 'normal' or 'small-caps'.

get_weight(*self*)
Set the font weight. Options are: A numeric value in the range 0-1000 or one of 'light', 'normal', 'regular', 'book', 'medium', 'roman', 'semibold', 'demibold', 'demi', 'bold', 'heavy', 'extra bold', 'black'

set_family(*self*, *family*)
Change the font family. May be either an alias (generic name is CSS parlance), such as: 'serif', 'sans-serif', 'cursive', 'fantasy', or 'monospace', a real font name or a list of real font names. Real font names are not supported when `rcParams["text.usetex"]` (default: `False`) is `True`.

set_file(*self*, *file*)
Set the filename of the fontfile to use. In this case, all other properties will be ignored.

set_fontconfig_pattern(*self*, *pattern*)
Set the properties by parsing a fontconfig *pattern*.

This support does not depend on fontconfig; we are merely borrowing its pattern syntax for use here.

set_name(*self*, *family*)
Change the font family. May be either an alias (generic name is CSS parlance), such as: 'serif', 'sans-serif', 'cursive', 'fantasy', or 'monospace', a real font name or a list of real font names. Real font names are not supported when `rcParams["text.usetex"]` (default: `False`) is `True`.

set_size(*self*, *size*)
Set the font size. Either an relative value of 'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large' or an absolute font size, e.g., 12.

set_slant(*self*, *style*)
Set the font style. Values are: 'normal', 'italic' or 'oblique'.

set_stretch(*self*, *stretch*)
Set the font stretch or width. Options are: 'ultra-condensed', 'extra-condensed', 'condensed', 'semi-condensed', 'normal', 'semi-expanded', 'expanded', 'extra-expanded' or 'ultra-expanded', or a numeric value in the range 0-1000.

set_style(*self*, *style*)
Set the font style. Values are: 'normal', 'italic' or 'oblique'.

set_variant(*self*, *variant*)
Set the font variant. Values are: 'normal' or 'small-caps'.

set_weight(*self*, *weight*)
Set the font weight. May be either a numeric value in the range 0-1000 or one of 'ultralight', 'light', 'normal', 'regular', 'book', 'medium', 'roman', 'semibold', 'demibold', 'demi', 'bold', 'heavy', 'extra bold', 'black'

class matplotlib.font_manager.JSONEncoder(**kwargs)
Bases: matplotlib.font_manager._JSONEncoder

[*Deprecated*]

Notes

Deprecated since version 3.2:

Constructor for JSONEncoder, with sensible defaults.

If skipkeys is false, then it is a TypeError to attempt encoding of keys that are not str, int, float or None. If skipkeys is True, such items are simply skipped.

If ensure_ascii is true, the output is guaranteed to be str objects with all incoming non-ASCII characters escaped. If ensure_ascii is false, the output can contain non-ASCII characters.

If check_circular is true, then lists, dicts, and custom encoded objects will be checked for circular references during encoding to prevent an infinite recursion (which would cause an OverflowError). Otherwise, no such check takes place.

If allow_nan is true, then NaN, Infinity, and -Infinity will be encoded as such. This behavior is not JSON specification compliant, but is consistent with most JavaScript based encoders and decoders. Otherwise, it will be a ValueError to encode such floats.

If sort_keys is true, then the output of dictionaries will be sorted by key; this is useful for regression tests to ensure that JSON serializations can be compared on a day-to-day basis.

If indent is a non-negative integer, then JSON array elements and object members will be pretty-printed with that indent level. An indent level of 0 will only insert newlines. None is the most compact representation.

If specified, separators should be an (item_separator, key_separator) tuple. The default is (', ', ': ') if *indent* is `None` and (',', ': ') otherwise. To get the most compact JSON representation, you should specify (',', ':') to eliminate whitespace.

If specified, default is a function that gets called for objects that can't otherwise be serialized. It should return a JSON encodable version of the object or raise a `TypeError`.

`matplotlib.font_manager.`**`afmFontProperty`**(*fontpath*, *font*)

Extract information from an AFM font file.

Parameters

font

[*AFM*] The AFM font file from which information will be extracted.

Returns

FontEntry

The extracted font properties.

matplotlib.font_manager.**createFontList**(*fontfiles*, *fontext='ttf'*)

[*Deprecated*] Create a font lookup list. The default is to create a list of TrueType fonts. An AFM font list can optionally be created.

Notes

Deprecated since version 3.2.

matplotlib.font_manager.**findSystemFonts**(*fontpaths=None*, *fontext='ttf'*)

Search for fonts in the specified font paths. If no paths are given, will use a standard set of system paths, as well as the list of fonts tracked by fontconfig if fontconfig is installed and available. A list of TrueType fonts are returned by default with AFM fonts as an option.

matplotlib.font_manager.**findfont**(*prop*, *fontext='ttf'*, *directory=None*, *fallback_to_default=True*, *rebuild_if_missing=True*)

Find a font that most closely matches the given font properties.

Parameters

prop

[str or *FontProperties*] The font properties to search for. This can be either a *FontProperties* object or a string defining a fontconfig patterns.

fontext

[{'ttf', 'afm'}, default: 'ttf'] The extension of the font file:

- 'ttf': TrueType and OpenType fonts (.ttf, .ttc, .otf)
- 'afm': Adobe Font Metrics (.afm)

directory

[str, optional] If given, only search this directory and its subdirectories.

fallback_to_default

[bool] If True, will fallback to the default font family (usually "DejaVu Sans" or "Helvetica") if the first lookup hard-fails.

rebuild_if_missing

[bool] Whether to rebuild the font cache and search again if the first match appears to point to a nonexisting font (i.e., the font cache contains outdated entries).

Returns

str

The filename of the best matching font.

Notes

This performs a nearest neighbor search. Each font is given a similarity score to the target font properties. The first font with the highest score is returned. If no matches below a certain threshold are found, the default font (usually DejaVu Sans) is returned.

The result is cached, so subsequent lookups don't have to perform the O(n) nearest neighbor search.

See the W3C Cascading Style Sheet, Level 1 documentation for a description of the font finding algorithm.

`matplotlib.font_manager.get_font(`*filename*, *hinting_factor=None*`)`

`matplotlib.font_manager.get_fontconfig_fonts(`*fontext='ttf'*`)`
List font filenames known to `fc-list` having the given extension.

`matplotlib.font_manager.get_fontext_synonyms(`*fontext*`)`
Return a list of file extensions extensions that are synonyms for the given file extension *fileext*.

`matplotlib.font_manager.is_opentype_cff_font(`*filename*`)`
Return whether the given font is a Postscript Compact Font Format Font embedded in an OpenType wrapper. Used by the PostScript and PDF backends that can not subset these fonts.

`matplotlib.font_manager.json_dump(`*data*, *filename*`)`
Dump *FontManager* *data* as JSON to the file named *filename*.

See also:

json_load

Notes

File paths that are children of the Matplotlib data path (typically, fonts shipped with Matplotlib) are stored relative to that data path (to remain valid across virtualenvs).

This function temporarily locks the output file to prevent multiple processes from overwriting one another's output.

`matplotlib.font_manager.json_load(`*filename*`)`
Load a *FontManager* from the JSON file named *filename*.

See also:

json_dump

matplotlib.font_manager.**list_fonts**(*directory*, *extensions*)
Return a list of all fonts matching any of the extensions, found recursively under the directory.

matplotlib.font_manager.**ttfFontProperty**(*font*)
Extract information from a TrueType font file.

Parameters

font

[FT2Font] The TrueType font file from which information will be extracted.

Returns

FontEntry

The extracted font properties.

matplotlib.font_manager.**win32FontDirectory**()
Return the user-specified font directory for Win32. This is looked up from the registry key

```
\\HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Explorer\Shell␣
↪Folders\Fonts
```

If the key is not found, %WINDIR%\Fonts will be returned.

matplotlib.font_manager.**win32InstalledFonts**(*directory=None*, *fontext='ttf'*)
Search for fonts in the specified font directory, or use the system directories if none given. Additionally, it is searched for user fonts installed. A list of TrueType font filenames are returned by default, or AFM fonts if *fontext* == 'afm'.

18.25 matplotlib.fontconfig_pattern

A module for parsing and generating fontconfig patterns.

class matplotlib.fontconfig_pattern.**FontconfigPatternParser**
Bases: object

A simple pyparsing-based parser for fontconfig patterns.

parse(*self*, *pattern*)
Parse the given fontconfig *pattern* and return a dictionary of key/value pairs useful for initializing a *font_manager.FontProperties* object.

matplotlib.fontconfig_pattern.**family_escape**(*repl*, *string*, *count=0*)
Return the string obtained by replacing the leftmost non-overlapping occurrences of pattern in string by the replacement repl.

matplotlib.fontconfig_pattern.**family_unescape**(*repl*, *string*, *count=0*)
Return the string obtained by replacing the leftmost non-overlapping occurrences of pattern in string by the replacement repl.

matplotlib.fontconfig_pattern.**generate_fontconfig_pattern**(*d*)
Given a dictionary of key/value pairs, generates a fontconfig pattern string.

matplotlib.fontconfig_pattern.**parse_fontconfig_pattern**(*pattern*)
Parse the given fontconfig *pattern* and return a dictionary of key/value pairs useful for initializing a *font_manager.FontProperties* object.

matplotlib.fontconfig_pattern.**value_escape**(*repl*, *string*, *count=0*)
Return the string obtained by replacing the leftmost non-overlapping occurrences of pattern in string by the replacement repl.

matplotlib.fontconfig_pattern.**value_unescape**(*repl*, *string*, *count=0*)
Return the string obtained by replacing the leftmost non-overlapping occurrences of pattern in string by the replacement repl.

18.26 matplotlib.gridspec

gridspec contains classes that help to layout multiple *Axes* in a grid-like pattern within a figure.

The *GridSpec* specifies the overall grid structure. Individual cells within the grid are referenced by *SubplotSpec*s.

See the tutorial *Customizing Figure Layouts Using GridSpec and Other Functions* for a comprehensive usage guide.

18.26.1 Classes

GridSpec(nrows, ncols[, figure, left, ...])	A grid layout to place subplots within a figure.
SubplotSpec(gridspec, num1[, num2])	Specifies the location of a subplot in a *GridSpec*.
GridSpecBase(nrows, ncols[, height_ratios, ...])	A base class of GridSpec that specifies the geometry of the grid that a subplot will be placed.
GridSpecFromSubplotSpec(nrows, ncols, ...[, ...])	GridSpec whose subplot layout parameters are inherited from the location specified by a given SubplotSpec.

matplotlib.gridspec.GridSpec

class matplotlib.gridspec.**GridSpec**(*nrows*, *ncols*, *figure=None*, *left=None*, *bottom=None*, *right=None*, *top=None*, *wspace=None*, *hspace=None*, *width_ratios=None*, *height_ratios=None*)

Bases: *matplotlib.gridspec.GridSpecBase*

A grid layout to place subplots within a figure.

The location of the grid cells is determined in a similar way to *SubplotParams* using *left*, *right*, *top*, *bottom*, *wspace* and *hspace*.

Parameters

nrows, ncols

[int] The number of rows and columns of the grid.

figure

[*Figure*, optional] Only used for constrained layout to create a proper layoutbox.

left, right, top, bottom

[float, optional] Extent of the subplots as a fraction of figure width or height. Left cannot be larger than right, and bottom cannot be larger than top. If not given, the values will be inferred from a figure or rcParams at draw time. See also *GridSpec.get_subplot_params*.

wspace

[float, optional] The amount of width reserved for space between subplots, expressed as a fraction of the average axis width. If not given, the values will be inferred from a figure or rcParams when necessary. See also *GridSpec.get_subplot_params*.

hspace

[float, optional] The amount of height reserved for space between subplots, expressed as a fraction of the average axis height. If not given, the values will be inferred from a figure or rcParams when necessary. See also *GridSpec.get_subplot_params*.

width_ratios

[array-like of length *ncols*, optional] Defines the relative widths of the columns. Each column gets a relative width of `width_ratios[i] / sum(width_ratios)`. If not given, all columns will have the same width.

height_ratios

[array-like of length *nrows*, optional] Defines the relative heights of the rows. Each column gets a relative height of `height_ratios[i] / sum(height_ratios)`. If not given, all rows will have the same height.

`__getstate__`(*self*)

`__init__`(*self*, *nrows*, *ncols*, *figure=None*, *left=None*, *bottom=None*, *right=None*, *top=None*, *wspace=None*, *hspace=None*, *width_ratios=None*, *height_ratios=None*)

Parameters

nrows, ncols

[int] The number of rows and columns of the grid.

figure

[*Figure*, optional] Only used for constrained layout to create a proper layoutbox.

left, right, top, bottom

[float, optional] Extent of the subplots as a fraction of figure width or height. Left cannot be larger than right, and bottom cannot be larger than top. If not given, the values will be inferred from a figure or rcParams at draw time. See also *GridSpec.get_subplot_params*.

wspace

[float, optional] The amount of width reserved for space between subplots, expressed as a fraction of the average axis width. If not given, the values will be inferred from a figure or rcParams when necessary. See also *GridSpec.get_subplot_params*.

hspace

[float, optional] The amount of height reserved for space between subplots, expressed as a fraction of the average axis height. If not given, the values will be inferred from a figure or rcParams when necessary. See also *GridSpec.get_subplot_params*.

width_ratios

[array-like of length *ncols*, optional] Defines the relative widths of the columns. Each column gets a relative width of `width_ratios[i] / sum(width_ratios)`. If not given, all columns will have the same width.

height_ratios

[array-like of length *nrows*, optional] Defines the relative heights of the rows. Each column gets a relative height of `height_ratios[i] / sum(height_ratios)`. If not given, all rows will have the same height.

`__module__ = 'matplotlib.gridspec'`

`get_subplot_params`(*self*, *figure=None*)
Return the *SubplotParams* for the GridSpec.

In order of precedence the values are taken from

- non-*None* attributes of the GridSpec
- the provided *figure*
- `rcParams["figure.subplot.*"]`

`locally_modified_subplot_params`(*self*)
Return a list of the names of the subplot parameters explicitly set in the GridSpec.

This is a subset of the attributes of *SubplotParams*.

`tight_layout`(*self*, *figure*, *renderer=None*, *pad=1.08*, *h_pad=None*, *w_pad=None*, *rect=None*)
Adjust subplot parameters to give specified padding.

Parameters

pad

[float] Padding between the figure edge and the edges of subplots, as a fraction of the font-size.

h_pad, w_pad

[float, optional] Padding (height/width) between edges of adjacent subplots. Defaults to *pad*.

rect

[tuple of 4 floats, default: (0, 0, 1, 1), i.e. the whole figure] (left, bottom, right, top) rectangle in normalized figure coordinates that the whole subplots area (including labels) will fit into.

`update`(*self*, ***kwargs*)
Update the subplot parameters of the grid.

Parameters that are not explicitly given are not changed. Setting a parameter to *None* resets it to `rcParams["figure.subplot.*"]`.

Parameters

left, right, top, bottom

[float or None, optional] Extent of the subplots as a fraction of figure width or height.

wspace, hspace

[float, optional] Spacing between the subplots as a fraction of the average subplot width / height.

Examples using `matplotlib.gridspec.GridSpec`

- sphx_glr_gallery_lines_bars_and_markers_psd_demo.py
- sphx_glr_gallery_lines_bars_and_markers_scatter_hist.py
- sphx_glr_gallery_images_contours_and_fields_plot_streamplot.py
- sphx_glr_gallery_subplots_axes_and_figures_align_labels_demo.py
- sphx_glr_gallery_subplots_axes_and_figures_demo_constrained_layout.py
- sphx_glr_gallery_subplots_axes_and_figures_demo_tight_layout.py
- sphx_glr_gallery_subplots_axes_and_figures_gridspec_and_subplots.py
- sphx_glr_gallery_subplots_axes_and_figures_gridspec_multicolumn.py
- sphx_glr_gallery_subplots_axes_and_figures_gridspec_nested.py
- sphx_glr_gallery_subplots_axes_and_figures_subplots_demo.py
- sphx_glr_gallery_userdemo_demo_gridspec03.py
- sphx_glr_gallery_userdemo_demo_gridspec06.py
- *Customizing Figure Layouts Using GridSpec and Other Functions*
- *Constrained Layout Guide*
- *Tight Layout guide*
- *origin and extent in imshow*

matplotlib.gridspec.SubplotSpec

`class matplotlib.gridspec.SubplotSpec(`*gridspec*, *num1*, *num2=None*`)`

Bases: `object`

Specifies the location of a subplot in a *`GridSpec`*.

Note: Likely, you'll never instantiate a *`SubplotSpec`* yourself. Instead you will typically obtain one from a *`GridSpec`* using item-access.

Parameters

gridspec

[*GridSpec*] The GridSpec, which the subplot is referencing.

num1, num2

[int] The subplot will occupy the num1-th cell of the given gridspec. If num2 is provided, the subplot will span between num1-th cell and num2-th cell *inclusive*.

The index starts from 0.

`__dict__ = mappingproxy({'__module__': 'matplotlib.gridspec', '__doc__': "\n Specifies th`

`__eq__`(*self*, *other*)
Two SubplotSpecs are considered equal if they refer to the same position(s) in the same *GridSpec*.

`__getstate__`(*self*)

`__hash__`(*self*)
Return hash(self).

`__init__`(*self*, *gridspec*, *num1*, *num2=None*)
Initialize self. See help(type(self)) for accurate signature.

`__module__ = 'matplotlib.gridspec'`

`__repr__`(*self*)
Return repr(self).

`__weakref__`
list of weak references to the object (if defined)

`property colspan`
The columns spanned by this subplot, as a `range` object.

`get_geometry`(*self*)
Return the subplot geometry as tuple `(n_rows, n_cols, start, stop)`.

The indices *start* and *stop* define the range of the subplot within the *GridSpec*. *stop* is inclusive (i.e. for a single cell `start == stop`).

`get_gridspec`(*self*)

`get_position`(*self*, *figure*, *return_all=False*)
Update the subplot position from `figure.subplotpars`.

`get_rows_columns`(*self*)
[*Deprecated*] Return the subplot row and column numbers as a tuple `(n_rows, n_cols, row_start, row_stop, col_start, col_stop)`.

Notes

Deprecated since version 3.3.

`get_topmost_subplotspec`(*self*)
Return the topmost *SubplotSpec* instance associated with the subplot.

`property num2`

`property rowspan`
The rows spanned by this subplot, as a `range` object.

`subgridspec`(*self*, *nrows*, *ncols*, ***kwargs*)
Create a GridSpec within this subplot.

The created *GridSpecFromSubplotSpec* will have this *SubplotSpec* as a parent.

Parameters

nrows

[int] Number of rows in grid.

ncols

[int] Number or columns in grid.

Returns

GridSpecFromSubplotSpec

Other Parameters

****kwargs**

All other parameters are passed to *GridSpecFromSubplotSpec*.

See also:

matplotlib.pyplot.subplots

Examples

Adding three subplots in the space occupied by a single subplot:

```
fig = plt.figure()
gs0 = fig.add_gridspec(3, 1)
ax1 = fig.add_subplot(gs0[0])
ax2 = fig.add_subplot(gs0[1])
gssub = gs0[2].subgridspec(1, 3)
for i in range(3):
    fig.add_subplot(gssub[0, i])
```

Examples using `matplotlib.gridspec.SubplotSpec`

- *Customizing Figure Layouts Using GridSpec and Other Functions*
- *Constrained Layout Guide*
- *Tight Layout guide*

matplotlib.gridspec.GridSpecBase

class matplotlib.gridspec.**GridSpecBase**(*nrows*, *ncols*, *height_ratios=None*, *width_ratios=None*)

Bases: `object`

A base class of GridSpec that specifies the geometry of the grid that a subplot will be placed.

Parameters

nrows, ncols

[int] The number of rows and columns of the grid.

width_ratios

[array-like of length *ncols*, optional] Defines the relative widths of the columns. Each column gets a relative width of `width_ratios[i] / sum(width_ratios)`. If not given, all columns will have the same width.

height_ratios

[array-like of length *nrows*, optional] Defines the relative heights of the rows. Each column gets a relative height of `height_ratios[i] / sum(height_ratios)`. If not given, all rows will have the same height.

`__dict__ = mappingproxy({'__module__': 'matplotlib.gridspec', '__doc__': '\n A base class`

`__getitem__`(*self*, *key*)

Create and return a *SubplotSpec* instance.

`__init__`(*self*, *nrows*, *ncols*, *height_ratios=None*, *width_ratios=None*)

Parameters

nrows, ncols

[int] The number of rows and columns of the grid.

width_ratios

[array-like of length *ncols*, optional] Defines the relative widths of the columns. Each column gets a relative width of

`width_ratios[i] / sum(width_ratios)`. If not given, all columns will have the same width.

height_ratios

[array-like of length *nrows*, optional] Defines the relative heights of the rows. Each column gets a relative height of `height_ratios[i] / sum(height_ratios)`. If not given, all rows will have the same height.

`__module__ = 'matplotlib.gridspec'`

`__repr__`(*self*)
Return repr(self).

`__weakref__`
list of weak references to the object (if defined)

`get_geometry`(*self*)
Return a tuple containing the number of rows and columns in the grid.

`get_grid_positions`(*self*, *fig*, *raw=False*)
Return the positions of the grid cells in figure coordinates.

Parameters

fig

[*Figure*] The figure the grid should be applied to. The subplot parameters (margins and spacing between subplots) are taken from *fig*.

raw

[bool, default: False] If *True*, the subplot parameters of the figure are not taken into account. The grid spans the range [0, 1] in both directions without margins and there is no space between grid cells. This is used for constrained_layout.

Returns

bottoms, tops, lefts, rights

[array] The bottom, top, left, right positions of the grid cells in figure coordinates.

`get_height_ratios`(*self*)
Return the height ratios.

This is *None* if no height ratios have been set explicitly.

`get_subplot_params`(*self*, *figure=None*)

`get_width_ratios`(*self*)
Return the width ratios.

This is *None* if no width ratios have been set explicitly.

`property ncols`
The number of columns in the grid.

`new_subplotspec`(*self*, *loc*, *rowspan=1*, *colspan=1*)
Create and return a *SubplotSpec* instance.

Parameters

loc

[(int, int)] The position of the subplot in the grid as `(row_index, column_index)`.

rowspan, colspan

[int, default: 1] The number of rows and columns the subplot should span in the grid.

`property nrows`
The number of rows in the grid.

`set_height_ratios`(*self*, *height_ratios*)
Set the relative heights of the rows.

height_ratios must be of length *nrows*. Each row gets a relative height of `height_ratios[i] / sum(height_ratios)`.

`set_width_ratios`(*self*, *width_ratios*)
Set the relative widths of the columns.

width_ratios must be of length *ncols*. Each column gets a relative width of `width_ratios[i] / sum(width_ratios)`.

`subplots`(*self*, ***, *sharex=False*, *sharey=False*, *squeeze=True*, *subplot_kw=None*)
Add all subplots specified by this *GridSpec* to its parent figure.

This utility wrapper makes it convenient to create common layouts of subplots in a single call.

Parameters

sharex, sharey

[bool or {'none', 'all', 'row', 'col'}, default: False] Controls sharing of properties among x (*sharex*) or y (*sharey*) axes:

- True or 'all': x- or y-axis will be shared among all subplots.
- False or 'none': each subplot x- or y-axis will be independent.
- 'row': each subplot row will share an x- or y-axis.

- 'col': each subplot column will share an x- or y-axis.

When subplots have a shared x-axis along a column, only the x tick labels of the bottom subplot are created. Similarly, when subplots have a shared y-axis along a row, only the y tick labels of the first column subplot are created. To later turn other subplots' ticklabels on, use *tick_params*.

squeeze

[bool, optional, default: True]

- If True, extra dimensions are squeezed out from the returned array of Axes:
  - if only one subplot is constructed (nrows=ncols=1), the resulting single Axes object is returned as a scalar.
  - for Nx1 or 1xM subplots, the returned object is a 1D numpy object array of Axes objects.
  - for NxM, subplots with N>1 and M>1 are returned as a 2D array.
- If False, no squeezing at all is done: the returned Axes object is always a 2D array containing Axes instances, even if it ends up being 1x1.

subplot_kw

[dict, optional] Dict with keywords passed to the *add_subplot* call used to create each subplot.

Returns

ax

[*Axes* object or array of Axes objects.] *ax* can be either a single *Axes* object or an array of Axes objects if more than one subplot was created. The dimensions of the resulting array can be controlled with the squeeze keyword, see above.

See also:

pyplot.subplots

Figure.add_subplot

pyplot.subplot

Examples using `matplotlib.gridspec.GridSpecBase`

- sphx_glr_gallery_lines_bars_and_markers_psd_demo.py
- sphx_glr_gallery_images_contours_and_fields_plot_streamplot.py
- sphx_glr_gallery_subplots_axes_and_figures_align_labels_demo.py
- sphx_glr_gallery_subplots_axes_and_figures_demo_constrained_layout.py
- sphx_glr_gallery_subplots_axes_and_figures_gridspec_multicolumn.py
- sphx_glr_gallery_subplots_axes_and_figures_gridspec_nested.py
- sphx_glr_gallery_userdemo_demo_gridspec03.py
- *Customizing Figure Layouts Using GridSpec and Other Functions*
- *Constrained Layout Guide*
- *Tight Layout guide*
- *origin and extent in imshow*

matplotlib.gridspec.GridSpecFromSubplotSpec

class matplotlib.gridspec.**GridSpecFromSubplotSpec**(*nrows*, *ncols*, *subplot_spec*, *wspace=None*, *hspace=None*, *height_ratios=None*, *width_ratios=None*)

Bases: *matplotlib.gridspec.GridSpecBase*

GridSpec whose subplot layout parameters are inherited from the location specified by a given SubplotSpec.

The number of rows and number of columns of the grid need to be set. An instance of SubplotSpec is also needed to be set from which the layout parameters will be inherited. The wspace and hspace of the layout can be optionally specified or the default values (from the figure or rcParams) will be used.

__init__(*self*, *nrows*, *ncols*, *subplot_spec*, *wspace=None*, *hspace=None*, *height_ratios=None*, *width_ratios=None*)

The number of rows and number of columns of the grid need to be set. An instance of SubplotSpec is also needed to be set from which the layout parameters will be inherited. The wspace and hspace of the layout can be optionally specified or the default values (from the figure or rcParams) will be used.

__module__ = 'matplotlib.gridspec'

get_subplot_params(*self*, *figure=None*)

Return a dictionary of subplot layout parameters.

`get_topmost_subplotspec`(*self*)
Return the topmost *SubplotSpec* instance associated with the subplot.

Examples using `matplotlib.gridspec.GridSpecFromSubplotSpec`

- sphx_glr_gallery_subplots_axes_and_figures_demo_constrained_layout.py
- sphx_glr_gallery_subplots_axes_and_figures_gridspec_nested.py
- sphx_glr_gallery_userdemo_demo_gridspec06.py
- *Customizing Figure Layouts Using GridSpec and Other Functions*
- *Constrained Layout Guide*

18.27 `matplotlib.image`

The image module supports basic image loading, rescaling and display operations.

`class matplotlib.image.`**`AxesImage`**(*ax*, *cmap=None*, *norm=None*, *interpolation=None*, *origin=None*, *extent=None*, *filternorm=True*, *filterrad=4.0*, *resample=False*, ***kwargs*)
Bases: `matplotlib.image._ImageBase`

An image attached to an Axes.

Parameters

ax

[*Axes*] The axes the image will belong to.

cmap

[str or *Colormap*, default: `rcParams["image.cmap"]` (default: `'viridis'`)] The Colormap instance or registered colormap name used to map scalar data to colors.

norm

[*Normalize*] Maps luminance to 0-1.

interpolation

[str, default: `rcParams["image.interpolation"]` (default: `'antialiased'`)] Supported values are 'none', 'antialiased', 'nearest', 'bilinear', 'bicubic', 'spline16', 'spline36', 'hanning', 'hamming', 'hermite', 'kaiser', 'quadric', 'catrom', 'gaussian', 'bessel', 'mitchell', 'sinc', 'lanczos'.

origin

[{'upper', 'lower'}, default: `rcParams["image.origin"]` (default: `'upper'`)] Place the [0, 0] index of the array in the upper left or lower left corner of the axes. The convention 'upper' is typically used for matrices and images.

extent

[tuple, optional] The data axes (left, right, bottom, top) for making image plots registered with data plots. Default is to label the pixel centers with the zero-based row and column indices.

filternorm

[bool, default: True] A parameter for the antigrain image resize filter (see the antigrain documentation). If filternorm is set, the filter normalizes integer values and corrects the rounding errors. It doesn't do anything with the source floating point values, it corrects only integers according to the rule of 1.0 which means that any sum of pixel weights must be equal to 1.0. So, the filter function must produce a graph of the proper shape.

filterrad

[float > 0, default: 4] The filter radius for filters that have a radius parameter, i.e. when interpolation is one of: 'sinc', 'lanczos' or 'blackman'.

resample

[bool, default: False] When True, use a full resampling method. When False, only resample when the output image is larger than the input image.

****kwargs**

[*Artist* properties]

Parameters

norm

[*matplotlib.colors.Normalize* (or subclass thereof)] The normalizing object which scales data, typically into the interval `[0, 1]`. If *None*, *norm* defaults to a *colors.Normalize* object which initializes its scaling based on the first data processed.

cmap

[str or *Colormap*] The colormap used to map normalized data values to RGBA colors.

`format_cursor_data`(*self*, *data*)

Return a string representation of *data*.

Note: This method is intended to be overridden by artist subclasses. As an end-user of Matplotlib you will most likely not call this method yourself.

The default implementation converts ints and floats and arrays of ints and floats into a comma-separated string enclosed in square brackets.

See also:

get_cursor_data

`get_cursor_data`(*self*, *event*)
Return the image value at the event position or *None* if the event is outside the image.

See also:

matplotlib.artist.Artist.get_cursor_data

`get_extent`(*self*)
Return the image extent as tuple (left, right, bottom, top).

`get_window_extent`(*self*, *renderer=None*)
Get the axes bounding box in display space.

The bounding box' width and height are nonnegative.

Subclasses should override for inclusion in the bounding box "tight" calculation. Default is to return an empty bounding box at 0, 0.

Be careful when using this function, the results will not update if the artist window extent of the artist changes. The extent can change due to any changes in the transform stack, such as changing the axes limits, the figure size, or the canvas used (as is done when saving a figure). This can lead to unexpected behavior where interactive figures will look fine on the screen, but will save incorrectly.

`make_image`(*self*, *renderer*, *magnification=1.0*, *unsampled=False*)
Normalize, rescale, and colormap this image's data for rendering using *renderer*, with the given *magnification*.

If *unsampled* is True, the image will not be scaled, but an appropriate affine transformation will be returned instead.

Returns

image

[(M, N, 4) uint8 array] The RGBA image, resampled unless *unsampled* is True.

x, y

[float] The upper left corner where the image should be drawn, in pixel space.

trans

[Affine2D] The affine transformation from image to pixel space.

`set_extent`(*self*, *extent*)
Set the image extent.

Parameters

extent

[4-tuple of float] The position and size of the image as tuple `(left, right, bottom, top)` in data coordinates.

Notes

This updates `ax.dataLim`, and, if autoscaling, sets `ax.viewLim` to tightly fit the image, regardless of `dataLim`. Autoscaling state is not changed, so following this with `ax.autoscale_view()` will redo the autoscaling in accord with `dataLim`.

`class matplotlib.image.BboxImage`(*bbox*, *cmap=None*, *norm=None*, *interpolation=None*, *origin=None*, *filternorm=True*, *filterrad=4.0*, *resample=False*, ***kwargs*)
Bases: `matplotlib.image._ImageBase`

The Image class whose size is determined by the given bbox.

cmap is a colors.Colormap instance norm is a colors.Normalize instance to map luminance to 0-1

kwargs are an optional list of Artist keyword args

`contains`(*self*, *mouseevent*)
Test whether the mouse event occurred within the image.

`get_transform`(*self*)
Return the *Transform* instance used by this artist.

`get_window_extent`(*self*, *renderer=None*)
Get the axes bounding box in display space.

The bounding box' width and height are nonnegative.

Subclasses should override for inclusion in the bounding box "tight" calculation. Default is to return an empty bounding box at 0, 0.

Be careful when using this function, the results will not update if the artist window extent of the artist changes. The extent can change due to any changes in the transform stack, such as changing the axes limits, the figure size, or the canvas used (as is done when saving a figure). This can lead to

unexpected behavior where interactive figures will look fine on the screen, but will save incorrectly.

`make_image`(*self*, *renderer*, *magnification=1.0*, *unsampled=False*)
Normalize, rescale, and colormap this image's data for rendering using *renderer*, with the given *magnification*.

If *unsampled* is True, the image will not be scaled, but an appropriate affine transformation will be returned instead.

Returns

image

[(M, N, 4) uint8 array] The RGBA image, resampled unless *unsampled* is True.

x, y

[float] The upper left corner where the image should be drawn, in pixel space.

trans

[Affine2D] The affine transformation from image to pixel space.

`class matplotlib.image.FigureImage`(*fig*, *cmap=None*, *norm=None*, *offsetx=0*, *offsety=0*, *origin=None*, ***kwargs*)
Bases: `matplotlib.image._ImageBase`

An image attached to a figure.

cmap is a colors.Colormap instance norm is a colors.Normalize instance to map luminance to 0-1

kwargs are an optional list of Artist keyword args

`get_extent`(*self*)
Return the image extent as tuple (left, right, bottom, top).

`make_image`(*self*, *renderer*, *magnification=1.0*, *unsampled=False*)
Normalize, rescale, and colormap this image's data for rendering using *renderer*, with the given *magnification*.

If *unsampled* is True, the image will not be scaled, but an appropriate affine transformation will be returned instead.

Returns

image

[(M, N, 4) uint8 array] The RGBA image, resampled unless *unsampled* is True.

x, y

[float] The upper left corner where the image should be drawn, in pixel space.

trans

[Affine2D] The affine transformation from image to pixel space.

`set_data`(*self*, *A*)
Set the image array.

`zorder = 0`

`class matplotlib.image.NonUniformImage`(*ax*, *, *interpolation='nearest'*, ***kwargs*)
Bases: *matplotlib.image.AxesImage*

Parameters

interpolation

[{'nearest', 'bilinear'}, default: 'nearest']

****kwargs**

All other keyword arguments are identical to those of *AxesImage*.

`get_extent`(*self*)
Return the image extent as tuple (left, right, bottom, top).

`property is_grayscale`

`make_image`(*self*, *renderer*, *magnification=1.0*, *unsampled=False*)
Normalize, rescale, and colormap this image's data for rendering using *renderer*, with the given *magnification*.

If *unsampled* is True, the image will not be scaled, but an appropriate affine transformation will be returned instead.

Returns

image

[(M, N, 4) uint8 array] The RGBA image, resampled unless *unsampled* is True.

x, y

[float] The upper left corner where the image should be drawn, in pixel space.

trans

[Affine2D] The affine transformation from image to pixel space.

`set_array`(*self*, **args*)
Retained for backwards compatibility - use set_data instead.

Parameters

A

[array-like]

set_cmap(*self*, *cmap*)
Set the colormap for luminance data.

Parameters

cmap

[*Colormap* or str or None]

set_data(*self*, *x*, *y*, *A*)
Set the grid for the pixel centers, and the pixel values.

Parameters

x, y

[1D array-like] Monotonic arrays of shapes (N,) and (M,), respectively, specifying pixel centers.

A

[array-like] (M, N) ndarray or masked array of values to be colormapped, or (M, N, 3) RGB array, or (M, N, 4) RGBA array.

set_filternorm(*self*, *s*)
Set whether the resize filter normalizes the weights.

See help for *imshow*.

Parameters

filternorm

[bool]

set_filterrad(*self*, *s*)
Set the resize filter radius only applicable to some interpolation schemes -- see help for imshow

Parameters

filterrad

[positive float]

set_interpolation(*self*, *s*)

Parameters

s

[{'nearest', 'bilinear'} or None] If None, use `rcParams["image.interpolation"]` (default: `'antialiased'`).

`set_norm`(*self*, *norm*)

Set the normalization instance.

Parameters

norm

[*Normalize* or None]

Notes

If there are any colorbars using the mappable for this norm, setting the norm of the mappable will reset the norm, locator, and formatters on the colorbar to default.

`class matplotlib.image.PcolorImage`(*ax*, *x=None*, *y=None*, *A=None*, *cmap=None*, *norm=None*, ***kwargs*)

Bases: *matplotlib.image.AxesImage*

Make a pcolor-style plot with an irregular rectangular grid.

This uses a variation of the original irregular image code, and it is used by pcolorfast for the corresponding grid type.

Parameters

ax

[*Axes*] The axes the image will belong to.

x, y

[1D array-like, optional] Monotonic arrays of length N+1 and M+1, respectively, specifying rectangle boundaries. If not given, will default to `range(N + 1)` and `range(M + 1)`, respectively.

A

[array-like] The data to be color-coded. The interpretation depends on the shape:

- (M, N) ndarray or masked array: values to be colormapped
- (M, N, 3): RGB array
- (M, N, 4): RGBA array

cmap

[str or *Colormap*, default: `rcParams["image.cmap"]` (default: `'viridis'`)] The Colormap instance or registered colormap name used to map scalar data to colors.

norm

[*Normalize*] Maps luminance to 0-1.

****kwargs**

[*Artist* properties]

`get_cursor_data`(*self*, *event*)

Return the image value at the event position or *None* if the event is outside the image.

See also:

matplotlib.artist.Artist.get_cursor_data

`property is_grayscale`

`make_image`(*self*, *renderer*, *magnification=1.0*, *unsampled=False*)

Normalize, rescale, and colormap this image's data for rendering using *renderer*, with the given *magnification*.

If *unsampled* is True, the image will not be scaled, but an appropriate affine transformation will be returned instead.

Returns

image

[(M, N, 4) uint8 array] The RGBA image, resampled unless *unsampled* is True.

x, y

[float] The upper left corner where the image should be drawn, in pixel space.

trans

[Affine2D] The affine transformation from image to pixel space.

`set_array`(*self*, **args*)

Retained for backwards compatibility - use set_data instead.

Parameters

A

[array-like]

set_data(*self*, *x*, *y*, *A*)

Set the grid for the rectangle boundaries, and the data values.

Parameters

x, y

[1D array-like, optional] Monotonic arrays of length N+1 and M+1, respectively, specifying rectangle boundaries. If not given, will default to `range(N + 1)` and `range(M + 1)`, respectively.

A

[array-like] The data to be color-coded. The interpretation depends on the shape:

- (M, N) ndarray or masked array: values to be colormapped
- (M, N, 3): RGB array
- (M, N, 4): RGBA array

matplotlib.image.composite_images(*images*, *renderer*, *magnification=1.0*)

Composite a number of RGBA images into one. The images are composited in the order in which they appear in the *images* list.

Parameters

images

[list of Images] Each must have a `make_image` method. For each image, `can_composite` should return `True`, though this is not enforced by this function. Each image must have a purely affine transformation with no shear.

renderer

[*RendererBase*]

magnification

[float, default: 1] The additional magnification to apply for the renderer in use.

Returns

image

[uint8 3d array] The composited RGBA image.

offset_x, offset_y

[float] The (left, bottom) offset where the composited image should be placed in the output figure.

matplotlib.image.**imread**(*fname*, *format=None*)

Read an image from a file into an array.

Parameters

fname

[str or file-like] The image file to read: a filename, a URL or a file-like object opened in read-binary mode.

format

[str, optional] The image file format assumed for reading the data. If not given, the format is deduced from the filename. If nothing can be deduced, PNG is tried.

Returns

`numpy.array`

The image data. The returned array has shape

- (M, N) for grayscale images.
- (M, N, 3) for RGB images.
- (M, N, 4) for RGBA images.

matplotlib.image.**imsave**(*fname*, *arr*, *vmin=None*, *vmax=None*, *cmap=None*, *format=None*, *origin=None*, *dpi=100*, *, *metadata=None*, *pil_kwargs=None*)

Save an array as an image file.

Parameters

fname

[str or path-like or file-like] A path or a file-like object to store the image in. If *format* is not set, then the output format is inferred from the extension of *fname*, if any, and from `rcParams["savefig.format"]` (default: `'png'`) otherwise. If *format* is set, it determines the output format.

arr

[array-like] The image data. The shape can be one of MxN (luminance), MxNx3 (RGB) or MxNx4 (RGBA).

vmin, vmax

[float, optional] *vmin* and *vmax* set the color scaling for the image by fixing the values that map to the colormap color limits. If either *vmin* or *vmax* is None, that limit is determined from the *arr* min/max value.

cmap

[str or *Colormap*, default: rcParams["image.cmap"] (default: 'viridis')] A Colormap instance or registered colormap name. The colormap maps scalar data to colors. It is ignored for RGB(A) data.

format

[str, optional] The file format, e.g. 'png', 'pdf', 'svg', ... The behavior when this is unset is documented under *fname*.

origin

[{'upper', 'lower'}, default: rcParams["image.origin"] (default: 'upper')] Indicates whether the (0, 0) index of the array is in the upper left or lower left corner of the axes.

dpi

[float] The DPI to store in the metadata of the file. This does not affect the resolution of the output image. Depending on file format, this may be rounded to the nearest integer.

metadata

[dict, optional] Metadata in the image file. The supported keys depend on the output format, see the documentation of the respective backends for more information.

pil_kwargs

[dict, optional] Keyword arguments passed to PIL.Image.Image.save. If the 'pnginfo' key is present, it completely overrides *metadata*, including the default 'Software' key.

matplotlib.image.**pil_to_array**(*pilImage*)

Load a PIL image and return it as a numpy int array.

Returns

numpy.array

The array shape depends on the image type:

- (M, N) for grayscale images.
- (M, N, 3) for RGB images.
- (M, N, 4) for RGBA images.

matplotlib.image.**thumbnail**(*infile*, *thumbfile*, *scale=0.1*, *interpolation='bilinear'*, *preview=False*)

Make a thumbnail of image in *infile* with output filename *thumbfile*.

See /gallery/misc/image_thumbnail_sgskip.

Parameters

infile

[str or file-like] The image file. Matplotlib relies on Pillow for image reading, and thus supports a wide range of file formats, including PNG, JPG, TIFF and others.

thumbfile

[str or file-like] The thumbnail filename.

scale

[float, default: 0.1] The scale factor for the thumbnail.

interpolation

[str, default: 'bilinear'] The interpolation scheme used in the resampling. See the *interpolation* parameter of *imshow* for possible values.

preview

[bool, default: False] If True, the default backend (presumably a user interface backend) will be used which will cause a figure to be raised if *show* is called. If it is False, the figure is created using *FigureCanvasBase* and the drawing backend is selected as *Figure.savefig* would normally do.

Returns

Figure

The figure instance containing the thumbnail.

18.28 matplotlib.legend

The legend module defines the Legend class, which is responsible for drawing legends associated with axes and/or figures.

Important: It is unlikely that you would ever create a Legend instance manually. Most users would normally create a legend via the *legend* function. For more details on legends there is also a *legend guide*.

The *Legend* class is a container of legend handles and legend texts.

The legend handler map specifies how to create legend handles from artists (lines, patches, etc.) in the axes or figures. Default legend handlers are defined in the

legend_handler module. While not all artist types are covered by the default legend handlers, custom legend handlers can be defined to support arbitrary objects.

See the *legend guide* for more information.

class matplotlib.legend.**DraggableLegend**(*legend*, *use_blit=False*, *update='loc'*)

Bases: *matplotlib.offsetbox.DraggableOffsetBox*

Wrapper around a *Legend* to support mouse dragging.

Parameters

legend

[*Legend*] The *Legend* instance to wrap.

use_blit

[bool, optional] Use blitting for faster image composition. For details see *FuncAnimation*.

update

[{'loc', 'bbox'}, optional] If "loc", update the *loc* parameter of the legend upon finalizing. If "bbox", update the *bbox_to_anchor* parameter.

finalize_offset(*self*)

class matplotlib.legend.**Legend**(*parent*, *handles*, *labels*, *loc=None*, *numpoints=None*, *markerscale=None*, *markerfirst=True*, *scatterpoints=None*, *scatteryoffsets=None*, *prop=None*, *fontsize=None*, *labelcolor=None*, *borderpad=None*, *labelspacing=None*, *handlelength=None*, *handleheight=None*, *handletextpad=None*, *borderaxespad=None*, *columnspacing=None*, *ncol=1*, *mode=None*, *fancybox=None*, *shadow=None*, *title=None*, *title_fontsize=None*, *framealpha=None*, *edgecolor=None*, *facecolor=None*, *bbox_to_anchor=None*, *bbox_transform=None*, *frameon=None*, *handler_map=None*)

Bases: *matplotlib.artist.Artist*

Place a legend on the axes at location loc.

Parameters

parent

[*Axes* or *Figure*] The artist that contains the legend.

handles

[list of *Artist*] A list of Artists (lines, patches) to be added to the legend.

labels

[list of str] A list of labels to show next to the artists. The length of handles and labels should be the same. If they are not, they are truncated to the smaller of both lengths.

Other Parameters

loc

[str or pair of floats, default: `rcParams["legend.loc"]` (default: `'best'`) ('best' for axes, 'upper right' for figures)] The location of the legend.

The strings `'upper left'`, `'upper right'`, `'lower left'`, `'lower right'` place the legend at the corresponding corner of the axes/figure.

The strings `'upper center'`, `'lower center'`, `'center left'`, `'center right'` place the legend at the center of the corresponding edge of the axes/figure.

The string `'center'` places the legend at the center of the axes/figure.

The string `'best'` places the legend at the location, among the nine locations defined so far, with the minimum overlap with other drawn artists. This option can be quite slow for plots with large amounts of data; your plotting speed may benefit from providing a specific location.

The location can also be a 2-tuple giving the coordinates of the lower-left corner of the legend in axes coordinates (in which case *bbox_to_anchor* will be ignored).

For back-compatibility, `'center right'` (but no other location) can also be spelled `'right'`, and each "string" locations can also be given as a numeric value:

Location String	Location Code
'best'	0
'upper right'	1
'upper left'	2
'lower left'	3
'lower right'	4
'right'	5
'center left'	6
'center right'	7
'lower center'	8
'upper center'	9
'center'	10

bbox_to_anchor

[*BboxBase*, 2-tuple, or 4-tuple of floats] Box that is used to position the legend in conjunction with *loc*. Defaults to `axes.bbox` (if called as a method to *Axes.legend*) or `figure.bbox` (if *Figure.legend*). This argument allows arbitrary placement of the legend.

Bbox coordinates are interpreted in the coordinate system given by *bbox_transform*, with the default transform Axes or Figure coordinates, depending on which `legend` is called.

If a 4-tuple or *BboxBase* is given, then it specifies the bbox `(x, y, width, height)` that the legend is placed in. To put the legend in the best location in the bottom right quadrant of the axes (or figure):

```
loc='best', bbox_to_anchor=(0.5, 0., 0.5, 0.5)
```

A 2-tuple `(x, y)` places the corner of the legend specified by *loc* at x, y. For example, to put the legend's upper right-hand corner in the center of the axes (or figure) the following keywords can be used:

```
loc='upper right', bbox_to_anchor=(0.5, 0.5)
```

ncol

[int, default: 1] The number of columns that the legend has.

prop

[None or *matplotlib.font_manager.FontProperties* or dict] The font properties of the legend. If None (default), the current *matplotlib.rcParams* will be used.

fontsize

[int or {'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large'}] The font size of the legend. If the value is numeric the size will be the absolute font size in points. String values are relative to the current default font size. This argument is only used if *prop* is not specified.

labelcolor

[str or list] Sets the color of the text in the legend. Can be a valid color string (for example, 'red'), or a list of color strings. The labelcolor can also be made to match the color of the line or marker using 'linecolor', 'markerfacecolor' (or 'mfc'), or 'markeredgecolor' (or 'mec').

numpoints

[int, default: `rcParams["legend.numpoints"]` (default: `1`)] The number of marker points in the legend when creating a legend entry for a *Line2D* (line).

scatterpoints

[int, default: `rcParams["legend.scatterpoints"]` (default: `1`)] The number of marker points in the legend when creating a legend entry for a *PathCollection* (scatter plot).

scatteryoffsets

[iterable of floats, default: `[0.375, 0.5, 0.3125]`] The vertical offset (relative to the font size) for the markers created for a scatter plot legend entry. 0.0 is at the base the legend text, and 1.0 is at the top. To draw all markers at the same height, set to `[0.5]`.

markerscale

[float, default: `rcParams["legend.markerscale"]` (default: `1.0`)] The relative size of legend markers compared with the originally drawn ones.

markerfirst

[bool, default: True] If *True*, legend marker is placed to the left of the legend label. If *False*, legend marker is placed to the right of the legend label.

frameon

[bool, default: `rcParams["legend.frameon"]` (default: `True`)] Whether the legend should be drawn on a patch (frame).

fancybox

[bool, default: `rcParams["legend.fancybox"]` (default: `True`)] Whether round edges should be enabled around the *FancyBboxPatch* which makes up the legend's background.

shadow

[bool, default: rcParams["legend.shadow"] (default: False)] Whether to draw a shadow behind the legend.

framealpha

[float, default: rcParams["legend.framealpha"] (default: 0.8)] The alpha transparency of the legend's background. If *shadow* is activated and *framealpha* is None, the default value is ignored.

facecolor

["inherit" or color, default: rcParams["legend.facecolor"] (default: 'inherit')] The legend's background color. If "inherit", use rcParams["axes.facecolor"] (default: 'white').

edgecolor

["inherit" or color, default: rcParams["legend.edgecolor"] (default: '0.8')] The legend's background patch edge color. If "inherit", use take rcParams["axes.edgecolor"] (default: 'black').

mode

[{"expand", None}] If *mode* is set to "expand" the legend will be horizontally expanded to fill the axes area (or *bbox_to_anchor* if defines the legend's size).

bbox_transform

[None or *matplotlib.transforms.Transform*] The transform for the bounding box (*bbox_to_anchor*). For a value of None (default) the Axes' transAxes transform will be used.

title

[str or None] The legend's title. Default is no title (None).

title_fontsize

[int or {'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large'}, default: rcParams["legend.title_fontsize"] (default: None)] The font size of the legend's title.

borderpad

[float, default: rcParams["legend.borderpad"] (default: 0.4)] The fractional whitespace inside the legend border, in font-size units.

labelspacing

[float, default: rcParams["legend.labelspacing"] (default: 0.5)] The vertical space between the legend entries, in font-size units.

handlelength

[float, default: rcParams["legend.handlelength"] (default: 2.0)]
The length of the legend handles, in font-size units.

handletextpad

[float, default: rcParams["legend.handletextpad"] (default: 0.8)]
The pad between the legend handle and text, in font-size units.

borderaxespad

[float, default: rcParams["legend.borderaxespad"] (default: 0.5)]
The pad between the axes and legend border, in font-size units.

columnspacing

[float, default: rcParams["legend.columnspacing"] (default: 2.0)]
The spacing between columns, in font-size units.

handler_map

[dict or None] The custom dictionary mapping instances or types to a legend handler. This *handler_map* updates the default handler map found at *matplotlib.legend.Legend.get_legend_handler_map*.

Notes

Users can specify any arbitrary location for the legend using the *bbox_to_anchor* keyword argument. *bbox_to_anchor* can be a *BboxBase* (or derived therefrom) or a tuple of 2 or 4 floats. See *set_bbox_to_anchor* for more detail.

The legend location can be specified by setting *loc* with a tuple of 2 floats, which is interpreted as the lower-left corner of the legend in the normalized axes coordinate.

```
codes = {'best': 0, 'center': 10, 'center left': 6, 'center right': 7, 'lower center': 8
```

`contains`(*self*, *event*)
Test whether the artist contains the mouse event.

Parameters

mouseevent

[*matplotlib.backend_bases.MouseEvent*]

Returns

contains

[bool] Whether any values are within the radius.

details

[dict] An artist-specific dictionary of details of the event context, such as which points are contained in the pick radius. See the individual Artist subclasses for details.

`draw`(*self*, *renderer*)
Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

Parameters

renderer

[*RendererBase* subclass.]

Notes

This method is overridden in the Artist subclasses.

`draw_frame`(*self*, *b*)
Set whether the legend box patch is drawn.

Parameters

b

[bool]

`get_bbox_to_anchor`(*self*)
Return the bbox that the legend will be anchored to.

`get_children`(*self*)
Return a list of the child *Artist*s of this *Artist*.

`classmethod get_default_handler_map`()
A class method that returns the default handler map.

`get_draggable`(*self*)
Return `True` if the legend is draggable, `False` otherwise.

`get_frame`(*self*)
Return the *Rectangle* used to frame the legend.

`get_frame_on`(*self*)
Get whether the legend box patch is drawn.

`static get_legend_handler`(*legend_handler_map*, *orig_handle*)
Return a legend handler from *legend_handler_map* that corresponds to *orig_handler*.

legend_handler_map should be a dictionary object (that is returned by the get_legend_handler_map method).

It first checks if the *orig_handle* itself is a key in the *legend_handler_map* and return the associated value. Otherwise, it checks for each of the classes in its method-resolution-order. If no matching key is found, it returns `None`.

`get_legend_handler_map`(*self*)
Return the handler map.

`get_lines`(*self*)
Return the list of *Line2D*s in the legend.

`get_patches`(*self*)
Return the list of *Patch*s in the legend.

`get_texts`(*self*)
Return the list of *Text*s in the legend.

`get_tightbbox`(*self*, *renderer*)
Like *Legend.get_window_extent*, but uses the box for the legend.

Parameters

renderer

[*RendererBase* subclass] renderer that will be used to draw the figures (i.e. `fig.canvas.get_renderer()`)

Returns

BboxBase

The bounding box in figure pixel coordinates.

`get_title`(*self*)
Return the *Text* instance for the legend title.

`get_window_extent`(*self*, *renderer=None*)
Get the axes bounding box in display space.

The bounding box' width and height are nonnegative.

Subclasses should override for inclusion in the bounding box "tight" calculation. Default is to return an empty bounding box at 0, 0.

Be careful when using this function, the results will not update if the artist window extent of the artist changes. The extent can change due to any changes in the transform stack, such as changing the axes limits, the figure size, or the canvas used (as is done when saving a figure). This can lead to unexpected behavior where interactive figures will look fine on the screen, but will save incorrectly.

`set_bbox_to_anchor`(*self*, *bbox*, *transform=None*)
Set the bbox that the legend will be anchored to.

Parameters

bbox

[*BboxBase* or tuple] The bounding box can be specified in the following ways:

- A *BboxBase* instance
- A tuple of `(left, bottom, width, height)` in the given transform (normalized axes coordinate if None)
- A tuple of `(left, bottom)` where the width and height will be assumed to be zero.
- *None*, to remove the bbox anchoring, and use the parent bbox.

transform

[*Transform*, optional] A transform to apply to the bounding box. If not specified, this will use a transform to the bounding box of the parent.

`classmethod set_default_handler_map`(*handler_map*)

A class method to set the default handler map.

`set_draggable`(*self*, *state*, *use_blit=False*, *update='loc'*)

Enable or disable mouse dragging support of the legend.

Parameters

state

[bool] Whether mouse dragging is enabled.

use_blit

[bool, optional] Use blitting for faster image composition. For details see *FuncAnimation*.

update

[{'loc', 'bbox'}, optional] The legend parameter to be changed when dragged:

- 'loc': update the *loc* parameter of the legend
- 'bbox': update the *bbox_to_anchor* parameter of the legend

Returns

DraggableLegend **or** ***None***

If *state* is `True` this returns the *DraggableLegend* helper instance. Otherwise this returns *None*.

`set_frame_on`(*self*, *b*)
Set whether the legend box patch is drawn.

Parameters

b

[bool]

`set_title`(*self*, *title*, *prop=None*)
Set the legend title. Fontproperties can be optionally set with *prop* parameter.

`classmethod update_default_handler_map`(*handler_map*)
A class method to update the default handler map.

`zorder = 5`

18.29 matplotlib.legend_handler

Default legend handlers.

It is strongly encouraged to have read the *legend guide* before this documentation.

Legend handlers are expected to be a callable object with a following signature.

```
legend_handler(legend, orig_handle, fontsize, handlebox)
```

Where *legend* is the legend itself, *orig_handle* is the original plot, *fontsize* is the fontsize in pixels, and *handlebox* is a OffsetBox instance. Within the call, you should create relevant artists (using relevant properties from the *legend* and/or *orig_handle*) and add them into the handlebox. The artists needs to be scaled according to the fontsize (note that the size is in pixel, i.e., this is dpi-scaled value).

This module includes definition of several legend handler classes derived from the base class (HandlerBase) with the following method:

```
def legend_artist(self, legend, orig_handle, fontsize, handlebox)
```

`class matplotlib.legend_handler.HandlerBase`(*xpad=0.0*, *ypad=0.0*, *update_func=None*)
A Base class for default legend handlers.

The derived classes are meant to override *create_artists* method, which has a following signature.:

```
def create_artists(self, legend, orig_handle,
                   xdescent, ydescent, width, height, fontsize,
                   trans):
```

The overridden method needs to create artists of the given transform that fits in the given dimension (xdescent, ydescent, width, height) that are scaled by fontsize if necessary.

`adjust_drawing_area`(*self*, *legend*, *orig_handle*, *xdescent*, *ydescent*, *width*, *height*, *fontsize*)

`create_artists`(*self*, *legend*, *orig_handle*, *xdescent*, *ydescent*, *width*, *height*, *fontsize*, *trans*)

`legend_artist`(*self*, *legend*, *orig_handle*, *fontsize*, *handlebox*)

Return the artist that this HandlerBase generates for the given original artist/handle.

Parameters

legend

[*Legend*] The legend for which these legend artists are being created.

orig_handle

[*matplotlib.artist.Artist* or similar] The object for which these legend artists are being created.

fontsize

[int] The fontsize in pixels. The artists being created should be scaled according to the given fontsize.

handlebox

[*matplotlib.offsetbox.OffsetBox*] The box which has been created to hold this legend entry's artists. Artists created in the *legend_artist* method must be added to this handlebox inside this method.

`update_prop`(*self*, *legend_handle*, *orig_handle*, *legend*)

class matplotlib.legend_handler.`HandlerCircleCollection`(*yoffsets=None*, *sizes=None*, ***kw*)

Handler for *CircleCollection*s.

Parameters

numpoints

[int] Number of points to show in legend entry.

yoffsets

[array of floats] Length *numpoints* list of y offsets for each point in legend entry.

Notes

Any other keyword arguments are given to *HandlerNpoints*.

`create_collection`(*self*, *orig_handle*, *sizes*, *offsets*, *transOffset*)

`class matplotlib.legend_handler.HandlerErrorbar`(*xerr_size=0.5*, *yerr_size=None*, *marker_pad=0.3*, *numpoints=None*, ***kw*)

Handler for Errorbars.

Parameters

marker_pad

[float] Padding between points in legend entry.

numpoints

[int] Number of points to show in legend entry.

Notes

Any other keyword arguments are given to *HandlerNpoints*.

`create_artists`(*self*, *legend*, *orig_handle*, *xdescent*, *ydescent*, *width*, *height*, *fontsize*, *trans*)

`get_err_size`(*self*, *legend*, *xdescent*, *ydescent*, *width*, *height*, *fontsize*)

`class matplotlib.legend_handler.HandlerLine2D`(*marker_pad=0.3*, *numpoints=None*, ***kw*)

Handler for *Line2D* instances.

Parameters

marker_pad

[float] Padding between points in legend entry.

numpoints

[int] Number of points to show in legend entry.

Notes

Any other keyword arguments are given to *HandlerNpoints*.

`create_artists`(*self*, *legend*, *orig_handle*, *xdescent*, *ydescent*, *width*, *height*, *fontsize*, *trans*)

`class matplotlib.legend_handler.HandlerLineCollection`(*marker_pad=0.3*, *numpoints=None*, ***kw*)

Handler for *LineCollection* instances.

Parameters

marker_pad

[float] Padding between points in legend entry.

numpoints

[int] Number of points to show in legend entry.

Notes

Any other keyword arguments are given to *HandlerNpoints*.

`create_artists`(*self*, *legend*, *orig_handle*, *xdescent*, *ydescent*, *width*, *height*, *fontsize*, *trans*)

`get_numpoints`(*self*, *legend*)

`class matplotlib.legend_handler.HandlerNpoints`(*marker_pad=0.3*, *numpoints=None*, ***kw*)

A legend handler that shows *numpoints* points in the legend entry.

Parameters

marker_pad

[float] Padding between points in legend entry.

numpoints

[int] Number of points to show in legend entry.

Notes

Any other keyword arguments are given to *HandlerBase*.

`get_numpoints`(*self*, *legend*)

`get_xdata`(*self*, *legend*, *xdescent*, *ydescent*, *width*, *height*, *fontsize*)

class matplotlib.legend_handler.**HandlerNpointsYoffsets**(*numpoints=None*, *yoffsets=None*, ***kw*)

A legend handler that shows *numpoints* in the legend, and allows them to be individually offset in the y-direction.

Parameters

numpoints

[int] Number of points to show in legend entry.

yoffsets

[array of floats] Length *numpoints* list of y offsets for each point in legend entry.

Notes

Any other keyword arguments are given to *HandlerNpoints*.

get_ydata(*self*, *legend*, *xdescent*, *ydescent*, *width*, *height*, *fontsize*)

class matplotlib.legend_handler.**HandlerPatch**(*patch_func=None*, ***kw*)

Handler for *Patch* instances.

Parameters

patch_func

[callable, optional] The function that creates the legend key artist. *patch_func* should have the signature:

```
def patch_func(legend=legend, orig_handle=orig_handle,
               xdescent=xdescent, ydescent=ydescent,
               width=width, height=height, fontsize=fontsize)
```

Subsequently the created artist will have its `update_prop` method called and the appropriate transform will be applied.

Notes

Any other keyword arguments are given to *HandlerBase*.

create_artists(*self*, *legend*, *orig_handle*, *xdescent*, *ydescent*, *width*, *height*, *fontsize*, *trans*)

class matplotlib.legend_handler.**HandlerPathCollection**(*yoffsets=None*, *sizes=None*, ***kw*)

Handler for *PathCollection*s, which are used by *scatter*.

Parameters

numpoints

[int] Number of points to show in legend entry.

yoffsets

[array of floats] Length *numpoints* list of y offsets for each point in legend entry.

Notes

Any other keyword arguments are given to *HandlerNpoints*.

`create_collection`(*self*, *orig_handle*, *sizes*, *offsets*, *transOffset*)

`class matplotlib.legend_handler.HandlerPolyCollection`(*xpad=0.0*, *ypad=0.0*, *update_func=None*)

Handler for *PolyCollection* used in *fill_between* and *stackplot*.

`create_artists`(*self*, *legend*, *orig_handle*, *xdescent*, *ydescent*, *width*, *height*, *fontsize*, *trans*)

`class matplotlib.legend_handler.HandlerRegularPolyCollection`(*yoffsets=None*, *sizes=None*, ***kw*)

Handler for *RegularPolyCollection*s.

Parameters

numpoints

[int] Number of points to show in legend entry.

yoffsets

[array of floats] Length *numpoints* list of y offsets for each point in legend entry.

Notes

Any other keyword arguments are given to *HandlerNpoints*.

`create_artists`(*self*, *legend*, *orig_handle*, *xdescent*, *ydescent*, *width*, *height*, *fontsize*, *trans*)

`create_collection`(*self*, *orig_handle*, *sizes*, *offsets*, *transOffset*)

`get_numpoints`(*self*, *legend*)

`get_sizes`(*self*, *legend*, *orig_handle*, *xdescent*, *ydescent*, *width*, *height*, *fontsize*)

`update_prop`(*self*, *legend_handle*, *orig_handle*, *legend*)

class matplotlib.legend_handler.**HandlerStem**(*marker_pad=0.3*, *numpoints=None*, *bottom=None*, *yoffsets=None*, ***kw*)

Handler for plots produced by *stem*.

Parameters

marker_pad

[float, default: 0.3] Padding between points in legend entry.

numpoints

[int, optional] Number of points to show in legend entry.

bottom

[float, optional]

yoffsets

[array of floats, optional] Length *numpoints* list of y offsets for each point in legend entry.

Notes

Any other keyword arguments are given to *HandlerNpointsYoffsets*.

create_artists(*self*, *legend*, *orig_handle*, *xdescent*, *ydescent*, *width*, *height*, *fontsize*, *trans*)

get_ydata(*self*, *legend*, *xdescent*, *ydescent*, *width*, *height*, *fontsize*)

class matplotlib.legend_handler.**HandlerTuple**(*ndivide=1*, *pad=None*, ***kwargs*)

Handler for Tuple.

Additional kwargs are passed through to *HandlerBase*.

Parameters

ndivide

[int, default: 1] The number of sections to divide the legend area into. If None, use the length of the input tuple.

pad

[float, default: `rcParams["legend.borderpad"]` (default: `0.4`)] Padding in units of fraction of font size.

create_artists(*self*, *legend*, *orig_handle*, *xdescent*, *ydescent*, *width*, *height*, *fontsize*, *trans*)

matplotlib.legend_handler.**update_from_first_child**(*tgt*, *src*)

18.30 matplotlib.lines

The 2D line class which can draw with a variety of line styles, markers and colors.

18.30.1 Classes

Line2D(xdata, ydata[, linewidth, linestyle, ...])	A line - the line can have both a solid linestyle connecting all the vertices, and a marker at each vertex.
VertexSelector(line)	Manage the callbacks to maintain a list of selected vertices for *Line2D*.

matplotlib.lines.Line2D

class matplotlib.lines.**Line2D**(*xdata*, *ydata*, *linewidth=None*, *linestyle=None*, *color=None*, *marker=None*, *markersize=None*, *markeredgewidth=None*, *markeredgecolor=None*, *markerfacecolor=None*, *markerfacecoloralt='none'*, *fillstyle=None*, *antialiased=None*, *dash_capstyle=None*, *solid_capstyle=None*, *dash_joinstyle=None*, *solid_joinstyle=None*, *pickradius=5*, *drawstyle=None*, *markevery=None*, ***kwargs*)

Bases: *matplotlib.artist.Artist*

A line - the line can have both a solid linestyle connecting all the vertices, and a marker at each vertex. Additionally, the drawing of the solid line is influenced by the drawstyle, e.g., one can create "stepped" lines in various styles.

Create a *Line2D* instance with *x* and *y* data in sequences of *xdata*, *ydata*.

Additional keyword arguments are *Line2D* properties:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value
alpha	float or None
animated	bool
antialiased or aa	bool
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color or c	color
contains	unknown
dash_capstyle	{'butt', 'round', 'projecting'}

Table 141 – continued from previous page

Property	Description
dash_joinstyle	{'miter', 'round', 'bevel'}
dashes	sequence of floats (on/off ink in points) or (None, None)
data	(2, N) array or two 1D arrays
drawstyle or ds	{'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'd
figure	*Figure*
fillstyle	{'full', 'left', 'right', 'bottom', 'top', 'none'}
gid	str
in_layout	bool
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float
marker	marker style string, *Path* or *MarkerStyle*
markeredgecolor or mec	color
markeredgewidth or mew	float
markerfacecolor or mfc	color
markerfacecoloralt or mfcalt	color
markersize or ms	float
markevery	None or int or (int, int) or slice or List[int] or float or (float, float)
path_effects	*AbstractPathEffect*
picker	unknown
pickradius	float
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
solid_capstyle	{'butt', 'round', 'projecting'}
solid_joinstyle	{'miter', 'round', 'bevel'}
transform	*matplotlib.transforms.Transform*
url	str
visible	bool
xdata	1D array
ydata	1D array
zorder	float

See *set_linestyle()* for a description of the line styles, *set_marker()* for a description of the markers, and *set_drawstyle()* for a description of the draw styles.

__init__(*self*, *xdata*, *ydata*, *linewidth=None*, *linestyle=None*, *color=None*, *marker=None*, *markersize=None*, *markeredgewidth=None*, *markeredgecolor=None*, *markerfacecolor=None*, *markerfacecoloralt='none'*, *fillstyle=None*, *antialiased=None*, *dash_capstyle=None*, *solid_capstyle=None*, *dash_joinstyle=None*, *solid_joinstyle=None*, *pickradius=5*, *drawstyle=None*, *markevery=None*, ***kwargs*)

Create a *Line2D* instance with *x* and *y* data in sequences of *xdata*, *ydata*.

Additional keyword arguments are *Line2D* properties:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value
alpha	float or None
animated	bool
antialiased or aa	bool
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color or c	color
contains	unknown
dash_capstyle	{'butt', 'round', 'projecting'}
dash_joinstyle	{'miter', 'round', 'bevel'}
dashes	sequence of floats (on/off ink in points) or (None, None)
data	(2, N) array or two 1D arrays
drawstyle or ds	{'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'd
figure	*Figure*
fillstyle	{'full', 'left', 'right', 'bottom', 'top', 'none'}
gid	str
in_layout	bool
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float
marker	marker style string, *Path* or *MarkerStyle*
markeredgecolor or mec	color
markeredgewidth or mew	float
markerfacecolor or mfc	color
markerfacecoloralt or mfcalt	color
markersize or ms	float
markevery	None or int or (int, int) or slice or List[int] or float or (float, float)
path_effects	*AbstractPathEffect*
picker	unknown
pickradius	float
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
solid_capstyle	{'butt', 'round', 'projecting'}
solid_joinstyle	{'miter', 'round', 'bevel'}
transform	*matplotlib.transforms.Transform*
url	str
visible	bool
xdata	1D array
ydata	1D array
zorder	float

See *set_linestyle()* for a description of the line styles, *set_marker()* for a description of the markers, and *set_drawstyle()* for a description of the draw styles.

__module__ = 'matplotlib.lines'

__str__(*self*)
Return str(self).

property axes
The *Axes* instance the artist resides in, or *None*.

contains(*self*, *mouseevent*)
Test whether *mouseevent* occurred on the line.

An event is deemed to have occurred "on" the line if it is less than `self.pickradius` (default: 5 points) away from it. Use *get_pickradius* or *set_pickradius* to get or set the pick radius.

Parameters

mouseevent

[*matplotlib.backend_bases.MouseEvent*]

Returns

contains

[bool] Whether any values are within the radius.

details

[dict] A dictionary `{'ind': pointlist}`, where *pointlist* is a list of points of the line that are within the pickradius around the event position.

TODO: sort returned indices by distance

draw(*self*, *renderer*)
Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

Parameters

renderer

[*RendererBase* subclass.]

Notes

This method is overridden in the Artist subclasses.

```
drawStyleKeys = ['default', 'steps-mid', 'steps-pre', 'steps-post', 'steps']
```

```
drawStyles = {'default': '_draw_lines', 'steps': '_draw_steps_pre', 'steps-mid': '_draw_s
```

```
fillStyles = ('full', 'left', 'right', 'bottom', 'top', 'none')
```

```
filled_markers = ('o', 'v', '^', '<', '>', '8', 's', 'p', '*', 'h', 'H', 'D', 'd', 'P', 'X
```

`get_aa`(*self*)
Alias for *get_antialiased*.

`get_antialiased`(*self*)
Return whether antialiased rendering is used.

`get_c`(*self*)
Alias for *get_color*.

`get_color`(*self*)
Return the line color.

See also *set_color*.

`get_dash_capstyle`(*self*)
Return the cap style for dashed lines.

See also *set_dash_capstyle*.

`get_dash_joinstyle`(*self*)
Return the join style for dashed lines.

See also *set_dash_joinstyle*.

`get_data`(*self*, *orig=True*)
Return the xdata, ydata.

If *orig* is *True*, return the original data.

`get_drawstyle`(*self*)
Return the drawstyle.

See also *set_drawstyle*.

`get_ds`(*self*)
Alias for *get_drawstyle*.

`get_fillstyle`(*self*)
Return the marker fill style.

See also *set_fillstyle*.

`get_linestyle`(*self*)
Return the linestyle.

See also *set_linestyle*.

`get_linewidth`(*self*)
Return the linewidth in points.

See also *set_linewidth*.

`get_ls`(*self*)
Alias for *get_linestyle*.

`get_lw`(*self*)
Alias for *get_linewidth*.

`get_marker`(*self*)
Return the line marker.

See also *set_marker*.

`get_markeredgecolor`(*self*)
Return the marker edge color.

See also *set_markeredgecolor*.

`get_markeredgewidth`(*self*)
Return the marker edge width in points.

See also *set_markeredgewidth*.

`get_markerfacecolor`(*self*)
Return the marker face color.

See also *set_markerfacecolor*.

`get_markerfacecoloralt`(*self*)
Return the alternate marker face color.

See also *set_markerfacecoloralt*.

`get_markersize`(*self*)
Return the marker size in points.

See also *set_markersize*.

`get_markevery`(*self*)
Return the markevery setting for marker subsampling.

See also *set_markevery*.

`get_mec`(*self*)
Alias for *get_markeredgecolor*.

`get_mew`(*self*)
Alias for *get_markeredgewidth*.

`get_mfc`(*self*)
Alias for *get_markerfacecolor*.

`get_mfcalt`(*self*)
Alias for *get_markerfacecoloralt*.

get_ms(*self*)
Alias for *get_markersize*.

get_path(*self*)
Return the *Path* object associated with this line.

get_pickradius(*self*)
Return the pick radius used for containment tests.

See *contains* for more details.

get_solid_capstyle(*self*)
Return the cap style for solid lines.

See also *set_solid_capstyle*.

get_solid_joinstyle(*self*)
Return the join style for solid lines.

See also *set_solid_joinstyle*.

get_window_extent(*self*, *renderer*)
Get the axes bounding box in display space.

The bounding box' width and height are nonnegative.

Subclasses should override for inclusion in the bounding box "tight" calculation. Default is to return an empty bounding box at 0, 0.

Be careful when using this function, the results will not update if the artist window extent of the artist changes. The extent can change due to any changes in the transform stack, such as changing the axes limits, the figure size, or the canvas used (as is done when saving a figure). This can lead to unexpected behavior where interactive figures will look fine on the screen, but will save incorrectly.

get_xdata(*self*, *orig=True*)
Return the xdata.

If *orig* is *True*, return the original data, else the processed data.

get_xydata(*self*)
Return the *xy* data as a Nx2 numpy array.

get_ydata(*self*, *orig=True*)
Return the ydata.

If *orig* is *True*, return the original data, else the processed data.

is_dashed(*self*)
Return whether line has a dashed linestyle.

See also *set_linestyle*.

```
lineStyles = {'': '_draw_nothing', ' ': '_draw_nothing', '-': '_draw_solid', '--': '_dra
```

```
markers = {'.': 'point', ',': 'pixel', 'o': 'circle', 'v': 'triangle_down', '^': 'triang
```

property pickradius
Return the pick radius used for containment tests.

See *contains* for more details.

recache(*self*, *always=False*)

recache_always(*self*)

set_aa(*self*, *b*)
Alias for *set_antialiased*.

set_antialiased(*self*, *b*)
Set whether to use antialiased rendering.

Parameters

b

[bool]

set_c(*self*, *color*)
Alias for *set_color*.

set_color(*self*, *color*)
Set the color of the line.

Parameters

color

[color]

set_dash_capstyle(*self*, *s*)
Set the cap style for dashed lines.

Parameters

s

[{'butt', 'round', 'projecting'}] For examples see /gallery/lines_bars_and_markers/joinstyle.

set_dash_joinstyle(*self*, *s*)
Set the join style for dashed lines.

Parameters

s

[{'miter', 'round', 'bevel'}] For examples see /gallery/lines_bars_and_markers/joinstyle.

`set_dashes`(*self*, *seq*)
Set the dash sequence.

The dash sequence is a sequence of floats of even length describing the length of dashes and spaces in points.

For example, (5, 2, 1, 2) describes a sequence of 5 point and 1 point dashes separated by 2 point spaces.

Parameters

seq

[sequence of floats (on/off ink in points) or (None, None)] If *seq* is empty or `(None, None)`, the linestyle will be set to solid.

`set_data`(*self*, **args*)
Set the x and y data.

Parameters

***args**

[(2, N) array or two 1D arrays]

`set_drawstyle`(*self*, *drawstyle*)
Set the drawstyle of the plot.

The drawstyle determines how the points are connected.

Parameters

drawstyle

[{'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'default'] For 'default', the points are connected with straight lines.

The steps variants connect the points with step-like lines, i.e. horizontal lines with vertical steps. They differ in the location of the step:

- 'steps-pre': The step is at the beginning of the line segment, i.e. the line will be at the y-value of point to the right.
- 'steps-mid': The step is halfway between the points.
- 'steps-post: The step is at the end of the line segment, i.e. the line will be at the y-value of the point to the left.
- 'steps' is equal to 'steps-pre' and is maintained for backward-compatibility.

For examples see /gallery/lines_bars_and_markers/step_demo.

`set_ds`(*self*, *drawstyle*)
Alias for *set_drawstyle*.

`set_fillstyle`(*self*, *fs*)
Set the marker fill style.

Parameters

fs

[{'full', 'left', 'right', 'bottom', 'top', 'none'}] Possible values:

- 'full': Fill the whole marker with the *markerfacecolor*.
- 'left', 'right', 'bottom', 'top': Fill the marker half at the given side with the *markerfacecolor*. The other half of the marker is filled with *markerfacecoloralt*.
- 'none': No filling.

For examples see marker_fill_styles.

`set_linestyle`(*self*, *ls*)
Set the linestyle of the line.

Parameters

ls

[{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}] Possible values:

- A string:

Linestyle	Description
`'-'` or `'solid'`	solid line
`'--'` or `'dashed'`	dashed line
`'-.'` or `'dashdot'`	dash-dotted line
`':'` or `'dotted'`	dotted line
`'None'` or `' '` or `''`	draw nothing

- Alternatively a dash tuple of the following form can be provided:

```
(offset, onoffseq)
```

where `onoffseq` is an even length tuple of on and off ink in points. See also *set_dashes()*.

For examples see /gallery/lines_bars_and_markers/linestyles.

`set_linewidth`(*self*, *w*)
Set the line width in points.

Parameters

w

[float] Line width, in points.

`set_ls`(*self*, *ls*)
Alias for *set_linestyle*.

`set_lw`(*self*, *w*)
Alias for *set_linewidth*.

`set_marker`(*self*, *marker*)
Set the line marker.

Parameters

marker

[marker style string, *Path* or *MarkerStyle*] See *markers* for full description of possible arguments.

`set_markeredgecolor`(*self*, *ec*)
Set the marker edge color.

Parameters

ec

[color]

`set_markeredgewidth`(*self*, *ew*)
Set the marker edge width in points.

Parameters

ew

[float] Marker edge width, in points.

`set_markerfacecolor`(*self*, *fc*)
Set the marker face color.

Parameters

fc

[color]

`set_markerfacecoloralt`(*self*, *fc*)
Set the alternate marker face color.

Parameters

fc

[color]

`set_markersize`(*self*, *sz*)
Set the marker size in points.

Parameters

sz

[float] Marker size, in points.

`set_markevery`(*self*, *every*)
Set the markevery property to subsample the plot when using markers.

e.g., if `every=5`, every 5-th marker will be plotted.

Parameters

every

[None or int or (int, int) or slice or List[int] or float or (float, float) or List[bool]] Which markers to plot.

- every=None, every point will be plotted.
- every=N, every N-th marker will be plotted starting with marker 0.
- every=(start, N), every N-th marker, starting at point start, will be plotted.
- every=slice(start, end, N), every N-th marker, starting at point start, up to but not including point end, will be plotted.
- every=[i, j, m, n], only markers at points i, j, m, and n will be plotted.
- every=[True, False, True], positions that are True will be plotted.
- every=0.1, (i.e. a float) then markers will be spaced at approximately equal distances along the line; the distance along the line between markers is determined by multiplying the display-coordinate distance of the axes bounding-box diagonal by the value of every.
- every=(0.5, 0.1) (i.e. a length-2 tuple of float), the same functionality as every=0.1 is exhibited but the first marker will be 0.5 multiplied by the display-coordinate-diagonal-distance along the line.

For examples see /gallery/lines_bars_and_markers/markevery_demo.

Notes

Setting the markevery property will only show markers at actual data points. When using float arguments to set the markevery property on irregularly spaced data, the markers will likely not appear evenly spaced because the actual data points do not coincide with the theoretical spacing between markers.

When using a start offset to specify the first marker, the offset will be from the first data point which may be different from the first the visible data point if the plot is zoomed in.

If zooming in on a plot when using float arguments then the actual data points that have markers will change because the distance between markers is always determined from the display-coordinates axes-bounding-box-diagonal regardless of the actual axes data limits.

`set_mec`(*self*, *ec*)
Alias for *set_markeredgecolor*.

`set_mew`(*self*, *ew*)
Alias for *set_markeredgewidth*.

`set_mfc`(*self*, *fc*)
Alias for *set_markerfacecolor*.

`set_mfcalt`(*self*, *fc*)
Alias for *set_markerfacecoloralt*.

`set_ms`(*self*, *sz*)
Alias for *set_markersize*.

`set_picker`(*self*, *p*)
Define the picking behavior of the artist.

Parameters

picker

[None or bool or callable] This can be one of the following:

- *None*: Picking is disabled for this artist (default).
- A boolean: If *True* then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist.
- A function: If picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

to determine the hit test. if the mouse event is over the artist, return *hit=True* and props is a dictionary of properties you

want added to the PickEvent attributes.

- *deprecated*: For *Line2D* only, *picker* can also be a float that sets the tolerance for checking whether an event occurred "on" the line; this is deprecated. Use *Line2D.set_pickradius* instead.

set_pickradius(*self*, *d*)
Set the pick radius used for containment tests.

See *contains* for more details.

Parameters

d

[float] Pick radius, in points.

set_solid_capstyle(*self*, *s*)
Set the cap style for solid lines.

Parameters

s

[{'butt', 'round', 'projecting'}] For examples see /gallery/lines_bars_and_markers/joinstyle.

set_solid_joinstyle(*self*, *s*)
Set the join style for solid lines.

Parameters

s

[{'miter', 'round', 'bevel'}] For examples see /gallery/lines_bars_and_markers/joinstyle.

set_transform(*self*, *t*)
Set the Transformation instance used by this artist.

Parameters

t

[*matplotlib.transforms.Transform*]

set_xdata(*self*, *x*)
Set the data array for x.

Parameters

x

[1D array]

`set_ydata`(*self*, *y*)
Set the data array for y.

Parameters

y

[1D array]

`update_from`(*self*, *other*)
Copy properties from *other* to self.

```
validCap = ('butt', 'round', 'projecting')
```

```
validJoin = ('miter', 'round', 'bevel')
```

```
zorder = 2
```

Examples using `matplotlib.lines.Line2D`

- sphx_glr_gallery_lines_bars_and_markers_line_demo_dash_control.py
- sphx_glr_gallery_lines_bars_and_markers_stem_plot.py
- sphx_glr_gallery_statistics_boxplot_demo.py
- sphx_glr_gallery_text_labels_and_annotations_annotation_demo.py
- sphx_glr_gallery_text_labels_and_annotations_custom_legends.py
- sphx_glr_gallery_text_labels_and_annotations_figlegend_demo.py
- sphx_glr_gallery_text_labels_and_annotations_legend_demo.py
- sphx_glr_gallery_text_labels_and_annotations_line_with_text.py
- sphx_glr_gallery_pyplots_annotation_basic.py
- sphx_glr_gallery_pyplots_annotation_polar.py
- sphx_glr_gallery_pyplots_fig_axes_customize_simple.py
- sphx_glr_gallery_pyplots_fig_axes_labels_simple.py
- sphx_glr_gallery_pyplots_fig_x.py
- sphx_glr_gallery_shapes_and_collections_artist_reference.py
- sphx_glr_gallery_shapes_and_collections_path_patch.py
- sphx_glr_gallery_axes_grid1_parasite_simple.py
- sphx_glr_gallery_axisartist_demo_parasite_axes.py

- sphx_glr_gallery_axisartist_demo_parasite_axes2.py
- sphx_glr_gallery_showcase_bachelors_degrees_by_gender.py
- *Decay*
- *The double pendulum problem*
- *Animated line plot*
- *Oscilloscope*
- *MATPLOTLIB UNCHAINED*
- sphx_glr_gallery_event_handling_data_browser.py
- sphx_glr_gallery_event_handling_legend_picking.py
- sphx_glr_gallery_event_handling_looking_glass.py
- sphx_glr_gallery_event_handling_pick_event_demo.py
- sphx_glr_gallery_event_handling_pick_event_demo2.py
- sphx_glr_gallery_event_handling_poly_editor.py
- sphx_glr_gallery_event_handling_resample.py
- sphx_glr_gallery_misc_anchored_artists.py
- sphx_glr_gallery_misc_cursor_demo.py
- sphx_glr_gallery_misc_patheffect_demo.py
- sphx_glr_gallery_misc_set_and_get.py
- sphx_glr_gallery_misc_svg_filter_line.py
- sphx_glr_gallery_specialty_plots_skewt.py
- sphx_glr_gallery_ticks_and_spines_multiple_yaxis_with_spines.py
- sphx_glr_gallery_units_artist_tests.py
- sphx_glr_gallery_userdemo_simple_legend02.py
- sphx_glr_gallery_widgets_buttons.py
- sphx_glr_gallery_widgets_check_buttons.py
- sphx_glr_gallery_widgets_radio_buttons.py
- sphx_glr_gallery_widgets_slider_demo.py
- sphx_glr_gallery_widgets_span_selector.py
- sphx_glr_gallery_widgets_textbox.py
- *Usage Guide*
- *Pyplot tutorial*
- *Artist tutorial*

- *Legend guide*
- *Blitting tutorial*
- *Transformations Tutorial*

matplotlib.lines.VertexSelector

class matplotlib.lines.**VertexSelector**(*line*)

Bases: object

Manage the callbacks to maintain a list of selected vertices for *Line2D*. Derived classes should override *process_selected()* to do something with the picks.

Here is an example which highlights the selected verts with red circles:

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.lines as lines

class HighlightSelected(lines.VertexSelector):
    def __init__(self, line, fmt='ro', **kwargs):
        lines.VertexSelector.__init__(self, line)
        self.markers, = self.axes.plot([], [], fmt, **kwargs)

    def process_selected(self, ind, xs, ys):
        self.markers.set_data(xs, ys)
        self.canvas.draw()

fig, ax = plt.subplots()
x, y = np.random.rand(2, 30)
line, = ax.plot(x, y, 'bs-', picker=5)

selector = HighlightSelected(line)
plt.show()
```

Initialize the class with a *Line2D* instance. The line should already be added to some *matplotlib.axes.Axes* instance and should have the picker property set.

__dict__ = mappingproxy({'__module__': 'matplotlib.lines', '__doc__': "\n Manage the call

__init__(*self*, *line*)

Initialize the class with a *Line2D* instance. The line should already be added to some *matplotlib.axes.Axes* instance and should have the picker property set.

__module__ = 'matplotlib.lines'

__weakref__

list of weak references to the object (if defined)

onpick(*self*, *event*)

When the line is picked, update the set of selected indices.

process_selected(*self*, *ind*, *xs*, *ys*)
Default "do nothing" implementation of the *process_selected()* method.

Parameters

ind

[list of int] The indices of the selected vertices.

xs, ys

[array-like] The coordinates of the selected vertices.

Examples using matplotlib.lines.VertexSelector

18.30.2 Functions

segment_hits(cx, cy, x, y, radius)	Return the indices of the segments in the polyline with coordinates (*cx*, *cy*) that are within a distance *radius* of the point (*x*, *y*).

matplotlib.lines.segment_hits

matplotlib.lines.**segment_hits**(*cx*, *cy*, *x*, *y*, *radius*)
Return the indices of the segments in the polyline with coordinates (*cx*, *cy*) that are within a distance *radius* of the point (*x*, *y*).

Examples using matplotlib.lines.segment_hits

18.31 matplotlib.markers

Functions to handle markers; used by the marker functionality of *plot* and *scatter*.

All possible markers are defined here:

marker	symbol	description
"."		point
","		pixel
"o"		circle
"v"		triangle_down
"^"		triangle_up

Table 144 -

marker	symbol	description
"<"		triangle_left
">"		triangle_right
"1"		tri_down
"2"		tri_up
"3"		tri_left
"4"		tri_right
"8"		octagon
"s"		square
"p"		pentagon
"P"		plus (filled)
"*"		star
"h"		hexagon1
"H"		hexagon2
"+"		plus
"x"		x
"X"		x (filled)
"D"		diamond
"d"		thin_diamond
"|"		vline
"_"		hline
0 (TICKLEFT)		tickleft
1 (TICKRIGHT)		tickright
2 (TICKUP)		tickup
3 (TICKDOWN)		tickdown
4 (CARETLEFT)		caretleft
5 (CARETRIGHT)		caretright
6 (CARETUP)		caretup
7 (CARETDOWN)		caretdown
8 (CARETLEFTBASE)		caretleft (centered at base)
9 (CARETRIGHTBASE)		caretright (centered at base)
10 (CARETUPBASE)		caretup (centered at base)
11 (CARETDOWNBASE)		caretdown (centered at base)
"None", " " or ""		nothing

Table 144 -

marker	symbol	description
`'$...$'`		Render the string using mathtext. E.g `"$f$"` for marker showing t
`verts`		A list of (x, y) pairs used for Path vertices. The center of the mark
path		A *`Path`* instance.
`(numsides, 0, angle)`		A regular polygon with `numsides` sides, rotated by `angle`.
`(numsides, 1, angle)`		A star-like symbol with `numsides` sides, rotated by `angle`.
`(numsides, 2, angle)`		An asterisk with `numsides` sides, rotated by `angle`.

`None` is the default which means 'nothing', however this table is referred to from other docs for the valid inputs from marker inputs and in those cases `None` still means 'default'.

Note that special symbols can be defined via the *STIX math font*, e.g. `"$\u266B$"`. For an overview over the STIX font symbols refer to the STIX font table. Also see the /gallery/text_labels_and_annotations/stix_fonts_demo.

Integer numbers from `0` to `11` create lines and triangles. Those are equally accessible via capitalized variables, like `CARETDOWNBASE`. Hence the following are equivalent:

```
plt.plot([1, 2, 3], marker=11)
plt.plot([1, 2, 3], marker=matplotlib.markers.CARETDOWNBASE)
```

Examples showing the use of markers:

- /gallery/lines_bars_and_markers/marker_reference
- /gallery/shapes_and_collections/marker_path
- /gallery/lines_bars_and_markers/scatter_star_poly

18.31.1 Classes

`MarkerStyle`([marker, fillstyle])	A class representing marker types.

matplotlib.markers.MarkerStyle

class matplotlib.markers.**MarkerStyle**(*marker=None*, *fillstyle=None*)

Bases: `object`

A class representing marker types.

Attributes

markers

[list] All known markers.

filled_markers

[list] All known filled markers. This is a subset of *markers*.

fillstyles

[list] The supported fillstyles.

Parameters

marker

[str or array-like or None, default: None] *None* means no marker. For other possible marker values see the module docstring *matplotlib.markers*.

fillstyle

[str, default: 'full'] One of 'full', 'left', 'right', 'bottom', 'top', 'none'.

`__bool__`(*self*)

`__dict__ = mappingproxy({'__module__': 'matplotlib.markers', '__doc__': '\n A class repre`

`__init__`(*self*, *marker=None*, *fillstyle=None*)

Parameters

marker

[str or array-like or None, default: None] *None* means no marker. For other possible marker values see the module docstring *matplotlib.markers*.

fillstyle

[str, default: 'full'] One of 'full', 'left', 'right', 'bottom', 'top', 'none'.

`__module__ = 'matplotlib.markers'`

`__weakref__`
list of weak references to the object (if defined)

`filled_markers = ('o', 'v', '^', '<', '>', '8', 's', 'p', '*', 'h', 'H', 'D', 'd', 'P', 'X`

`fillstyles = ('full', 'left', 'right', 'bottom', 'top', 'none')`

`get_alt_path`(*self*)
Return a *Path* for the alternate part of the marker.

For unfilled markers, this is *None*; for filled markers, this is the area to be drawn with *markerfacecoloralt*.

`get_alt_transform`(*self*)
Return the transform to be applied to the *Path* from *MarkerStyle.get_alt_path()*.

get_capstyle(*self*)

get_fillstyle(*self*)

get_joinstyle(*self*)

get_marker(*self*)

get_path(*self*)
Return a *Path* for the primary part of the marker.

For unfilled markers this is the whole marker, for filled markers, this is the area to be drawn with *markerfacecolor*.

get_snap_threshold(*self*)

get_transform(*self*)
Return the transform to be applied to the *Path* from *MarkerStyle.get_path()*.

is_filled(*self*)

markers = {'.': 'point', ',': 'pixel', 'o': 'circle', 'v': 'triangle_down', '^': 'triang

set_fillstyle(*self*, *fillstyle*)
Set the fillstyle.

Parameters

fillstyle

[{'full', 'left', 'right', 'bottom', 'top', 'none'}] The part of the marker surface that is colored with markerfacecolor.

set_marker(*self*, *marker*)
Set the marker.

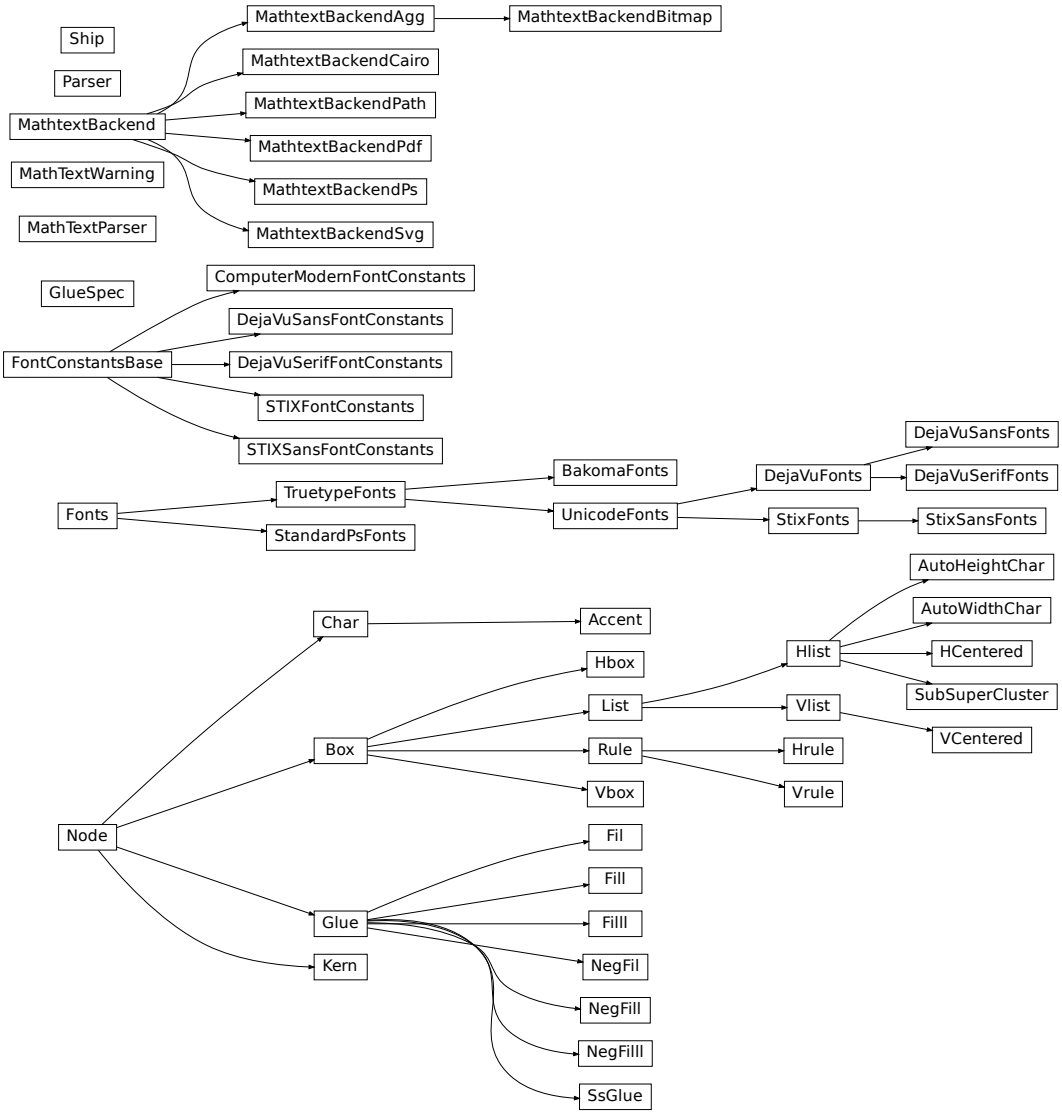
Parameters

marker

[str or array-like or None, default: None] *None* means no marker. For other possible marker values see the module docstring *matplotlib.markers*.

Examples using `matplotlib.markers.MarkerStyle`

18.32 matplotlib.mathtext



A module for parsing a subset of the TeX math syntax and rendering it to a Matplotlib backend.

For a tutorial of its usage, see *Writing mathematical expressions*. This document is primarily concerned with implementation details.

The module uses pyparsing to parse the TeX expression.

The Bakoma distribution of the TeX Computer Modern fonts, and STIX fonts are supported. There is experimental support for using arbitrary fonts, but results may vary without proper tweaking and metrics for those fonts.

class matplotlib.mathtext.**Accent**(*c*, *state*, *math=True*)

Bases: *matplotlib.mathtext.Char*

The font metrics need to be dealt with differently for accents, since they are already offset correctly from the baseline in TrueType fonts.

grow(*self*)

Grows one level larger. There is no limit to how big something can get.

render(*self*, *x*, *y*)

Render the character to the canvas.

shrink(*self*)

Shrinks one level smaller. There are only three levels of sizes, after which things will no longer get smaller.

class matplotlib.mathtext.**AutoHeightChar**(*c*, *height*, *depth*, *state*, *always=False*, *factor=None*)

Bases: *matplotlib.mathtext.Hlist*

A character as close to the given height and depth as possible.

When using a font with multiple height versions of some characters (such as the BaKoMa fonts), the correct glyph will be selected, otherwise this will always just return a scaled version of the glyph.

class matplotlib.mathtext.**AutoWidthChar**(*c*, *width*, *state*, *always=False*, *char_class=<class 'matplotlib.mathtext.Char'>*)

Bases: *matplotlib.mathtext.Hlist*

A character as close to the given width as possible.

When using a font with multiple width versions of some characters (such as the BaKoMa fonts), the correct glyph will be selected, otherwise this will always just return a scaled version of the glyph.

class matplotlib.mathtext.**BakomaFonts**(**args*, ***kwargs*)

Bases: *matplotlib.mathtext.TruetypeFonts*

Use the Bakoma TrueType fonts for rendering.

Symbols are strewn about a number of font files, each of which has its own proprietary 8-bit encoding.

Parameters

default_font_prop: `~.font_manager.FontProperties`

The default non-math font, or the base font for Unicode (generic) font rendering.

mathtext_backend: `MathtextBackend` subclass

Backend to which rendering is actually delegated.

`alias = '\\]'`

`get_sized_alternatives_for_symbol`(*self*, *fontname*, *sym*)
Override if your font provides multiple sizes of the same symbol. Should return a list of symbols matching *sym* in various sizes. The expression renderer will select the most appropriate size for a given situation from this list.

`target = ']'`

`class matplotlib.mathtext.Box`(*width*, *height*, *depth*)
Bases: *matplotlib.mathtext.Node*

A node with a physical location.

`grow`(*self*)
Grows one level larger. There is no limit to how big something can get.

`render`(*self*, *x1*, *y1*, *x2*, *y2*)

`shrink`(*self*)
Shrinks one level smaller. There are only three levels of sizes, after which things will no longer get smaller.

`class matplotlib.mathtext.Char`(*c*, *state*, *math=True*)
Bases: *matplotlib.mathtext.Node*

A single character.

Unlike TeX, the font information and metrics are stored with each *Char* to make it easier to lookup the font metrics when needed. Note that TeX boxes have a width, height, and depth, unlike Type1 and TrueType which use a full bounding box and an advance in the x-direction. The metrics must be converted to the TeX model, and the advance (if different from width) must be converted into a *Kern* node when the *Char* is added to its parent *Hlist*.

`get_kerning`(*self*, *next*)
Return the amount of kerning between this and the given character.

This method is called when characters are strung together into *Hlist* to create *Kern* nodes.

`grow`(*self*)
Grows one level larger. There is no limit to how big something can get.

`is_slanted`(*self*)

`render`(*self*, *x*, *y*)
Render the character to the canvas

shrink(*self*)
Shrinks one level smaller. There are only three levels of sizes, after which things will no longer get smaller.

class matplotlib.mathtext.ComputerModernFontConstants
Bases: *matplotlib.mathtext.FontConstantsBase*

delta = 0.075

delta_integral = 0.3

delta_slanted = 0.3

script_space = 0.075

sub1 = 0.2

sub2 = 0.3

subdrop = 0.2

sup1 = 0.45

class matplotlib.mathtext.DejaVuFonts(**args*, ***kwargs*)
Bases: *matplotlib.mathtext.UnicodeFonts*

Parameters

default_font_prop: `~.font_manager.FontProperties`

The default non-math font, or the base font for Unicode (generic) font rendering.

mathtext_backend: `MathtextBackend` subclass

Backend to which rendering is actually delegated.

use_cmex = False

class matplotlib.mathtext.DejaVuSansFontConstants
Bases: *matplotlib.mathtext.FontConstantsBase*

class matplotlib.mathtext.DejaVuSansFonts(**args*, ***kwargs*)
Bases: *matplotlib.mathtext.DejaVuFonts*

A font handling class for the DejaVu Sans fonts

If a glyph is not found it will fallback to Stix Sans

Parameters

default_font_prop: `~.font_manager.FontProperties`

The default non-math font, or the base font for Unicode (generic) font rendering.

mathtext_backend: `MathtextBackend` subclass

Backend to which rendering is actually delegated.

class matplotlib.mathtext.**DejaVuSerifFontConstants**
Bases: *matplotlib.mathtext.FontConstantsBase*

class matplotlib.mathtext.**DejaVuSerifFonts**(**args*, ***kwargs*)
Bases: *matplotlib.mathtext.DejaVuFonts*

A font handling class for the DejaVu Serif fonts

If a glyph is not found it will fallback to Stix Serif

Parameters

default_font_prop: `~.font_manager.FontProperties`

The default non-math font, or the base font for Unicode (generic) font rendering.

mathtext_backend: `MathtextBackend` subclass

Backend to which rendering is actually delegated.

matplotlib.mathtext.**Error**(*msg*)
Helper class to raise parser errors.

class matplotlib.mathtext.**Fil**(***kwargs*)
Bases: *matplotlib.mathtext.Glue*

[*Deprecated*]

Notes

Deprecated since version 3.3:

class matplotlib.mathtext.**Fill**(***kwargs*)
Bases: *matplotlib.mathtext.Glue*

[*Deprecated*]

Notes

Deprecated since version 3.3:

class matplotlib.mathtext.**Filll**(***kwargs*)
Bases: *matplotlib.mathtext.Glue*

[*Deprecated*]

Notes

Deprecated since version 3.3:

class matplotlib.mathtext.**FontConstantsBase**

Bases: object

A set of constants that controls how certain things, such as sub- and superscripts are laid out. These are all metrics that can't be reliably retrieved from the font metrics in the font itself.

```
delta = 0.025

delta_integral = 0.1

delta_slanted = 0.2

script_space = 0.05

sub1 = 0.3

sub2 = 0.5

subdrop = 0.4

sup1 = 0.7
```

class matplotlib.mathtext.**Fonts**(*default_font_prop*, *mathtext_backend*)

Bases: object

An abstract base class for a system of fonts to use for mathtext.

The class must be able to take symbol keys and font file names and return the character metrics. It also delegates to a backend class to do the actual drawing.

Parameters

default_font_prop: `~.font_manager.FontProperties`

The default non-math font, or the base font for Unicode (generic) font rendering.

mathtext_backend: `MathtextBackend` subclass

Backend to which rendering is actually delegated.

destroy(*self*)

Fix any cyclical references before the object is about to be destroyed.

get_kern(*self*, *font1*, *fontclass1*, *sym1*, *fontsize1*, *font2*, *fontclass2*, *sym2*, *fontsize2*, *dpi*)

Get the kerning distance for font between *sym1* and *sym2*.

fontX: one of the TeX font names:

```
tt, it, rm, cal, sf, bf or default/regular (non-math)
```

fontclassX: TODO

symX: a symbol in raw TeX form. e.g., '1', 'x' or 'sigma'

fontsizeX: the fontsize in points

dpi: the current dots-per-inch

`get_metrics`(*self*, *font*, *font_class*, *sym*, *fontsize*, *dpi*, *math=True*)

font: one of the TeX font names:

```
tt, it, rm, cal, sf, bf or default/regular (non-math)
```

font_class: TODO

sym: a symbol in raw TeX form. e.g., '1', 'x' or 'sigma'

fontsize: font size in points

dpi: current dots-per-inch

math: whether sym is a math character

Returns an object with the following attributes:

- *advance*: The advance distance (in points) of the glyph.
- *height*: The height of the glyph in points.
- *width*: The width of the glyph in points.
- *xmin*, *xmax*, *ymin*, *ymax* - the ink rectangle of the glyph
- *iceberg* - the distance from the baseline to the top of the glyph. This corresponds to TeX's definition of "height".

`get_results`(*self*, *box*)

Get the data needed by the backend to render the math expression. The return value is backend-specific.

`get_sized_alternatives_for_symbol`(*self*, *fontname*, *sym*)

Override if your font provides multiple sizes of the same symbol. Should return a list of symbols matching *sym* in various sizes. The expression renderer will select the most appropriate size for a given situation from this list.

`get_underline_thickness`(*self*, *font*, *fontsize*, *dpi*)

Get the line thickness that matches the given font. Used as a base unit for drawing lines such as in a fraction or radical.

`get_used_characters`(*self*)

Get the set of characters that were used in the math expression. Used by backends that need to subset fonts so they know which glyphs to include.

`get_xheight`(*self*, *font*, *fontsize*, *dpi*)

Get the xheight for the given *font* and *fontsize*.

render_glyph(*self*, *ox*, *oy*, *facename*, *font_class*, *sym*, *fontsize*, *dpi*)
Draw a glyph at

- *ox*, *oy*: position
- *facename*: One of the TeX face names
- *font_class*:
- *sym*: TeX symbol name or single character
- *fontsize*: fontsize in points
- *dpi*: The dpi to draw at.

render_rect_filled(*self*, *x1*, *y1*, *x2*, *y2*)
Draw a filled rectangle from (*x1*, *y1*) to (*x2*, *y2*).

set_canvas_size(*self*, *w*, *h*, *d*)
Set the size of the buffer used to render the math expression. Only really necessary for the bitmap backends.

class matplotlib.mathtext.Glue(*glue_type*, *copy=<deprecated parameter>*)
Bases: *matplotlib.mathtext.Node*

Most of the information in this object is stored in the underlying _GlueSpec class, which is shared between multiple glue objects. (This is a memory optimization which probably doesn't matter anymore, but it's easier to stick to what TeX does.)

property glue_subtype

grow(*self*)
Grows one level larger. There is no limit to how big something can get.

shrink(*self*)
Shrinks one level smaller. There are only three levels of sizes, after which things will no longer get smaller.

class matplotlib.mathtext.GlueSpec(**kwargs*)
Bases: object

[*Deprecated*] See *Glue*.

Notes

Deprecated since version 3.3.

copy(*self*)

classmethod factory(*glue_type*)

class matplotlib.mathtext.HCentered(*elements*)
Bases: *matplotlib.mathtext.Hlist*

A convenience class to create an *Hlist* whose contents are centered within its enclosing box.

class matplotlib.mathtext.Hbox(*width*)

Bases: *matplotlib.mathtext.Box*

A box with only width (zero height and depth).

class matplotlib.mathtext.Hlist(*elements*, *w=0.0*, *m='additional'*, *do_kern=True*)

Bases: *matplotlib.mathtext.List*

A horizontal list of boxes.

hpack(*self*, *w=0.0*, *m='additional'*)

Compute the dimensions of the resulting boxes, and adjust the glue if one of those dimensions is pre-specified. The computed sizes normally enclose all of the material inside the new box; but some items may stick out if negative glue is used, if the box is overfull, or if a \vbox includes other boxes that have been shifted left.

Parameters

w

[float, default: 0] A width.

m

[{'exactly', 'additional'}, default: 'additional'] Whether to produce a box whose width is 'exactly' *w*; or a box with the natural width of the contents, plus *w* ('additional').

Notes

The defaults produce a box with the natural width of the contents.

kern(*self*)

Insert *Kern* nodes between *Char* nodes to set kerning.

The *Char* nodes themselves determine the amount of kerning they need (in *get_kerning*), and this function just creates the correct linked list.

class matplotlib.mathtext.Hrule(*state*, *thickness=None*)

Bases: *matplotlib.mathtext.Rule*

Convenience class to create a horizontal rule.

class matplotlib.mathtext.Kern(*width*)

Bases: *matplotlib.mathtext.Node*

A *Kern* node has a width field to specify a (normally negative) amount of spacing. This spacing correction appears in horizontal lists between letters like A and V when the font designer said that it looks better to move them closer together

or further apart. A kern node can also appear in a vertical list, when its *width* denotes additional spacing in the vertical direction.

`depth = 0`

`grow`(*self*)
Grows one level larger. There is no limit to how big something can get.

`height = 0`

`shrink`(*self*)
Shrinks one level smaller. There are only three levels of sizes, after which things will no longer get smaller.

`class matplotlib.mathtext.List`(*elements*)
Bases: *matplotlib.mathtext.Box*

A list of nodes (either horizontal or vertical).

`grow`(*self*)
Grows one level larger. There is no limit to how big something can get.

`shrink`(*self*)
Shrinks one level smaller. There are only three levels of sizes, after which things will no longer get smaller.

`class matplotlib.mathtext.MathTextParser`(*output*)
Bases: `object`

Create a MathTextParser for the given backend *output*.

`get_depth`(*self*, *texstr*, *dpi=120*, *fontsize=14*)

Parameters

texstr

[str] A valid mathtext string, e.g., r'IQ: $sigma_i=15$'.

dpi

[float] The dots-per-inch setting used to render the text.

Returns

int

Offset of the baseline from the bottom of the image, in pixels.

`parse`(*self*, *s*, *dpi=72*, *prop=None*)
Parse the given math expression *s* at the given *dpi*. If *prop* is provided, it is a *FontProperties* object specifying the "default" font to use in the math expression, used for all non-math text.

The results are cached, so multiple calls to *parse* with the same expression should be fast.

to_mask(*self*, *texstr*, *dpi=120*, *fontsize=14*)

Parameters

texstr

[str] A valid mathtext string, e.g., r'IQ: $sigma_i=15$'.

dpi

[float] The dots-per-inch setting used to render the text.

fontsize

[int] The font size in points

Returns

array

[2D uint8 alpha] Mask array of rasterized tex.

depth

[int] Offset of the baseline from the bottom of the image, in pixels.

to_png(*self*, *filename*, *texstr*, *color='black'*, *dpi=120*, *fontsize=14*)

Render a tex expression to a PNG file.

Parameters

filename

A writable filename or fileobject.

texstr

[str] A valid mathtext string, e.g., r'IQ: $sigma_i=15$'.

color

[color] The text color.

dpi

[float] The dots-per-inch setting used to render the text.

fontsize

[int] The font size in points.

Returns

int

Offset of the baseline from the bottom of the image, in pixels.

to_rgba(*self*, *texstr*, *color='black'*, *dpi=120*, *fontsize=14*)

Parameters

texstr

[str] A valid mathtext string, e.g., r'IQ: $sigma_i=15$'.

color

[color] The text color.

dpi

[float] The dots-per-inch setting used to render the text.

fontsize

[int] The font size in points.

Returns

array

[(M, N, 4) array] RGBA color values of rasterized tex, colorized with *color*.

depth

[int] Offset of the baseline from the bottom of the image, in pixels.

exception matplotlib.mathtext.**MathTextWarning**

Bases: Warning

class matplotlib.mathtext.**MathtextBackend**

Bases: object

The base class for the mathtext backend-specific code. *MathtextBackend* subclasses interface between mathtext and specific Matplotlib graphics backends.

Subclasses need to override the following:

- *render_glyph()*
- *render_rect_filled()*
- *get_results()*

And optionally, if you need to use a FreeType hinting style:

- *get_hinting_type()*

get_hinting_type(*self*)

Get the FreeType hinting type to use with this particular backend.

get_results(*self*, *box*)
Return a backend-specific tuple to return to the backend after all processing is done.

render_glyph(*self*, *ox*, *oy*, *info*)
Draw a glyph described by *info* to the reference point (*ox*, *oy*).

render_rect_filled(*self*, *x1*, *y1*, *x2*, *y2*)
Draw a filled black rectangle from (*x1*, *y1*) to (*x2*, *y2*).

set_canvas_size(*self*, *w*, *h*, *d*)
Set the dimension of the drawing canvas.

class matplotlib.mathtext.MathtextBackendAgg
Bases: *matplotlib.mathtext.MathtextBackend*

Render glyphs and rectangles to an FTImage buffer, which is later transferred to the Agg image by the Agg backend.

get_hinting_type(*self*)
Get the FreeType hinting type to use with this particular backend.

get_results(*self*, *box*, *used_characters*)
Return a backend-specific tuple to return to the backend after all processing is done.

render_glyph(*self*, *ox*, *oy*, *info*)
Draw a glyph described by *info* to the reference point (*ox*, *oy*).

render_rect_filled(*self*, *x1*, *y1*, *x2*, *y2*)
Draw a filled black rectangle from (*x1*, *y1*) to (*x2*, *y2*).

set_canvas_size(*self*, *w*, *h*, *d*)
Set the dimension of the drawing canvas.

class matplotlib.mathtext.MathtextBackendBitmap
Bases: *matplotlib.mathtext.MathtextBackendAgg*

get_results(*self*, *box*, *used_characters*)
Return a backend-specific tuple to return to the backend after all processing is done.

class matplotlib.mathtext.MathtextBackendCairo
Bases: *matplotlib.mathtext.MathtextBackend*

Store information to write a mathtext rendering to the Cairo backend.

get_results(*self*, *box*, *used_characters*)
Return a backend-specific tuple to return to the backend after all processing is done.

render_glyph(*self*, *ox*, *oy*, *info*)
Draw a glyph described by *info* to the reference point (*ox*, *oy*).

render_rect_filled(*self*, *x1*, *y1*, *x2*, *y2*)
Draw a filled black rectangle from (*x1*, *y1*) to (*x2*, *y2*).

class matplotlib.mathtext.MathtextBackendPath
Bases: *matplotlib.mathtext.MathtextBackend*

Store information to write a mathtext rendering to the text path machinery.

get_results(*self*, *box*, *used_characters*)
Return a backend-specific tuple to return to the backend after all processing is done.

render_glyph(*self*, *ox*, *oy*, *info*)
Draw a glyph described by *info* to the reference point (*ox*, *oy*).

render_rect_filled(*self*, *x1*, *y1*, *x2*, *y2*)
Draw a filled black rectangle from (*x1*, *y1*) to (*x2*, *y2*).

class matplotlib.mathtext.MathtextBackendPdf
Bases: *matplotlib.mathtext.MathtextBackend*

Store information to write a mathtext rendering to the PDF backend.

get_results(*self*, *box*, *used_characters*)
Return a backend-specific tuple to return to the backend after all processing is done.

render_glyph(*self*, *ox*, *oy*, *info*)
Draw a glyph described by *info* to the reference point (*ox*, *oy*).

render_rect_filled(*self*, *x1*, *y1*, *x2*, *y2*)
Draw a filled black rectangle from (*x1*, *y1*) to (*x2*, *y2*).

class matplotlib.mathtext.MathtextBackendPs
Bases: *matplotlib.mathtext.MathtextBackend*

Store information to write a mathtext rendering to the PostScript backend.

get_results(*self*, *box*, *used_characters*)
Return a backend-specific tuple to return to the backend after all processing is done.

render_glyph(*self*, *ox*, *oy*, *info*)
Draw a glyph described by *info* to the reference point (*ox*, *oy*).

render_rect_filled(*self*, *x1*, *y1*, *x2*, *y2*)
Draw a filled black rectangle from (*x1*, *y1*) to (*x2*, *y2*).

class matplotlib.mathtext.MathtextBackendSvg
Bases: *matplotlib.mathtext.MathtextBackend*

Store information to write a mathtext rendering to the SVG backend.

get_results(*self*, *box*, *used_characters*)
Return a backend-specific tuple to return to the backend after all processing is done.

render_glyph(*self*, *ox*, *oy*, *info*)
Draw a glyph described by *info* to the reference point (*ox*, *oy*).

render_rect_filled(*self*, *x1*, *y1*, *x2*, *y2*)
Draw a filled black rectangle from (*x1*, *y1*) to (*x2*, *y2*).

class matplotlib.mathtext.NegFil(**kwargs)
Bases: *matplotlib.mathtext.Glue*

[*Deprecated*]

Notes

Deprecated since version 3.3:

class matplotlib.mathtext.NegFill(**kwargs)
Bases: *matplotlib.mathtext.Glue*

[*Deprecated*]

Notes

Deprecated since version 3.3:

class matplotlib.mathtext.NegFilll(**kwargs)
Bases: *matplotlib.mathtext.Glue*

[*Deprecated*]

Notes

Deprecated since version 3.3:

class matplotlib.mathtext.Node
Bases: object

A node in the TeX box model.

get_kerning(*self*, *next*)

grow(*self*)
Grows one level larger. There is no limit to how big something can get.

render(*self*, *x*, *y*)

shrink(*self*)
Shrinks one level smaller. There are only three levels of sizes, after which things will no longer get smaller.

class matplotlib.mathtext.Parser
Bases: object

A pyparsing-based parser for strings containing math expressions.

Raw text may also appear outside of pairs of $.

The grammar is based directly on that in TeX, though it cuts a few corners.

class State(*font_output*, *font*, *font_class*, *fontsize*, *dpi*)

Bases: object

Stores the state of the parser.

States are pushed and popped from a stack as necessary, and the "current" state is always at the top of the stack.

copy(*self*)

property font

accent(*self*, *s*, *loc*, *toks*)

accentprefixed(*self*, *s*, *loc*, *toks*)

auto_delim(*self*, *s*, *loc*, *toks*)

binom(*self*, *s*, *loc*, *toks*)

c_over_c(*self*, *s*, *loc*, *toks*)

customspace(*self*, *s*, *loc*, *toks*)

dfrac(*self*, *s*, *loc*, *toks*)

end_group(*self*, *s*, *loc*, *toks*)

font(*self*, *s*, *loc*, *toks*)

frac(*self*, *s*, *loc*, *toks*)

function(*self*, *s*, *loc*, *toks*)

genfrac(*self*, *s*, *loc*, *toks*)

get_state(*self*)

Get the current *State* of the parser.

group(*self*, *s*, *loc*, *toks*)

is_between_brackets(*self*, *s*, *loc*)

is_dropsub(*self*, *nucleus*)

is_overunder(*self*, *nucleus*)

is_slanted(*self*, *nucleus*)

main(*self*, *s*, *loc*, *toks*)

math(*self*, *s*, *loc*, *toks*)

math_string(*self*, *s*, *loc*, *toks*)

non_math(*self*, *s*, *loc*, *toks*)

operatorname(*self*, *s*, *loc*, *toks*)

overline(*self*, *s*, *loc*, *toks*)

parse(*self*, *s*, *fonts_object*, *fontsize*, *dpi*)
Parse expression *s* using the given *fonts_object* for output, at the given *fontsize* and *dpi*.

Returns the parse tree of *Node* instances.

pop_state(*self*)
Pop a *State* off of the stack.

push_state(*self*)
Push a new *State* onto the stack, copying the current state.

required_group(*self*, *s*, *loc*, *toks*)

simple_group(*self*, *s*, *loc*, *toks*)

space(*self*, *s*, *loc*, *toks*)

sqrt(*self*, *s*, *loc*, *toks*)

start_group(*self*, *s*, *loc*, *toks*)

subsuper(*self*, *s*, *loc*, *toks*)

symbol(*self*, *s*, *loc*, *toks*)

unknown_symbol(*self*, *s*, *loc*, *toks*)

class matplotlib.mathtext.**Rule**(*width*, *height*, *depth*, *state*)
Bases: *matplotlib.mathtext.Box*

A solid black rectangle.

It has *width*, *depth*, and *height* fields just as in an *Hlist*. However, if any of these dimensions is inf, the actual value will be determined by running the rule up to the boundary of the innermost enclosing box. This is called a "running dimension". The width is never running in an *Hlist*; the height and depth are never running in a *Vlist*.

render(*self*, *x*, *y*, *w*, *h*)

class matplotlib.mathtext.**STIXFontConstants**
Bases: *matplotlib.mathtext.FontConstantsBase*

delta = 0.05

delta_integral = 0.3

delta_slanted = 0.3

script_space = 0.1

sub2 = 0.6

sup1 = 0.8

class matplotlib.mathtext.**STIXSansFontConstants**
Bases: *matplotlib.mathtext.FontConstantsBase*

delta_integral = 0.3

delta_slanted = 0.6

script_space = 0.05

sup1 = 0.8

class matplotlib.mathtext.Ship
Bases: object

Ship boxes to output once they have been set up, this sends them to output.

Since boxes can be inside of boxes inside of boxes, the main work of *Ship* is done by two mutually recursive routines, *hlist_out* and *vlist_out*, which traverse the *Hlist* nodes and *Vlist* nodes inside of horizontal and vertical boxes. The global variables used in TeX to store state as it processes have become member variables here.

static clamp(*value*)

hlist_out(*self*, *box*)

vlist_out(*self*, *box*)

class matplotlib.mathtext.SsGlue(**kwargs*)
Bases: *matplotlib.mathtext.Glue*

[*Deprecated*]

Notes

Deprecated since version 3.3:

class matplotlib.mathtext.StandardPsFonts(*default_font_prop*)
Bases: *matplotlib.mathtext.Fonts*

Use the standard postscript fonts for rendering to backend_ps

Unlike the other font classes, BakomaFont and UnicodeFont, this one requires the Ps backend.

Parameters

default_font_prop: `~.font_manager.FontProperties`

The default non-math font, or the base font for Unicode (generic) font rendering.

mathtext_backend: `MathtextBackend` subclass

Backend to which rendering is actually delegated.

basepath = '/usr/local/lib64/python3.8/site-packages/matplotlib/mpl-data/fonts/afm'

fontmap = {'cal': 'pzcmi8a', 'rm': 'pncr8a', 'tt': 'pcrr8a', 'it': 'pncri8a', 'sf': 'phv

`get_kern`(*self*, *font1*, *fontclass1*, *sym1*, *fontsize1*, *font2*, *fontclass2*, *sym2*, *fontsize2*, *dpi*)
Get the kerning distance for font between *sym1* and *sym2*.

fontX: one of the TeX font names:

```
tt, it, rm, cal, sf, bf or default/regular (non-math)
```

fontclassX: TODO

symX: a symbol in raw TeX form. e.g., '1', 'x' or 'sigma'

fontsizeX: the fontsize in points

dpi: the current dots-per-inch

`get_underline_thickness`(*self*, *font*, *fontsize*, *dpi*)
Get the line thickness that matches the given font. Used as a base unit for drawing lines such as in a fraction or radical.

`get_xheight`(*self*, *font*, *fontsize*, *dpi*)
Get the xheight for the given *font* and *fontsize*.

`class matplotlib.mathtext.StixFonts`(**args*, ***kwargs*)
Bases: *matplotlib.mathtext.UnicodeFonts*

A font handling class for the STIX fonts.

In addition to what UnicodeFonts provides, this class:

- supports "virtual fonts" which are complete alpha numeric character sets with different font styles at special Unicode code points, such as "Blackboard".
- handles sized alternative characters for the STIXSizeX fonts.

Parameters

default_font_prop: `~.font_manager.FontProperties`

The default non-math font, or the base font for Unicode (generic) font rendering.

mathtext_backend: `MathtextBackend` subclass

Backend to which rendering is actually delegated.

`cm_fallback = False`

`get_sized_alternatives_for_symbol`(*self*, *fontname*, *sym*)
Override if your font provides multiple sizes of the same symbol. Should return a list of symbols matching *sym* in various sizes. The expression renderer will select the most appropriate size for a given situation from this list.

`use_cmex = False`

class matplotlib.mathtext.StixSansFonts(*args, **kwargs)

Bases: *matplotlib.mathtext.StixFonts*

A font handling class for the STIX fonts (that uses sans-serif characters by default).

Parameters

default_font_prop: `~.font_manager.FontProperties`

The default non-math font, or the base font for Unicode (generic) font rendering.

mathtext_backend: `MathtextBackend` subclass

Backend to which rendering is actually delegated.

class matplotlib.mathtext.SubSuperCluster

Bases: *matplotlib.mathtext.Hlist*

A hack to get around that fact that this code does a two-pass parse like TeX. This lets us store enough information in the hlist itself, namely the nucleus, sub- and super-script, such that if another script follows that needs to be attached, it can be reconfigured on the fly.

class matplotlib.mathtext.TruetypeFonts(*default_font_prop*, *mathtext_backend*)

Bases: *matplotlib.mathtext.Fonts*

A generic base class for all font setups that use Truetype fonts (through FT2Font).

Parameters

default_font_prop: `~.font_manager.FontProperties`

The default non-math font, or the base font for Unicode (generic) font rendering.

mathtext_backend: `MathtextBackend` subclass

Backend to which rendering is actually delegated.

destroy(*self*)

Fix any cyclical references before the object is about to be destroyed.

get_kern(*self*, *font1*, *fontclass1*, *sym1*, *fontsize1*, *font2*, *fontclass2*, *sym2*, *fontsize2*, *dpi*)

Get the kerning distance for font between *sym1* and *sym2*.

fontX: one of the TeX font names:

```
tt, it, rm, cal, sf, bf or default/regular (non-math)
```

fontclassX: TODO

symX: a symbol in raw TeX form. e.g., '1', 'x' or 'sigma'

fontsizeX: the fontsize in points

dpi: the current dots-per-inch

`get_underline_thickness`(*self*, *font*, *fontsize*, *dpi*)
Get the line thickness that matches the given font. Used as a base unit for drawing lines such as in a fraction or radical.

`get_xheight`(*self*, *fontname*, *fontsize*, *dpi*)
Get the xheight for the given *font* and *fontsize*.

`class matplotlib.mathtext.UnicodeFonts`(**args*, ***kwargs*)
Bases: *matplotlib.mathtext.TruetypeFonts*

An abstract base class for handling Unicode fonts.

While some reasonably complete Unicode fonts (such as DejaVu) may work in some situations, the only Unicode font I'm aware of with a complete set of math symbols is STIX.

This class will "fallback" on the Bakoma fonts when a required symbol can not be found in the font.

Parameters

default_font_prop: `~.font_manager.FontProperties`

The default non-math font, or the base font for Unicode (generic) font rendering.

mathtext_backend: `MathtextBackend` subclass

Backend to which rendering is actually delegated.

`get_sized_alternatives_for_symbol`(*self*, *fontname*, *sym*)
Override if your font provides multiple sizes of the same symbol. Should return a list of symbols matching *sym* in various sizes. The expression renderer will select the most appropriate size for a given situation from this list.

`use_cmex = True`

`class matplotlib.mathtext.VCentered`(*elements*)
Bases: *matplotlib.mathtext.Vlist*

A convenience class to create a *Vlist* whose contents are centered within its enclosing box.

`class matplotlib.mathtext.Vbox`(*height*, *depth*)
Bases: *matplotlib.mathtext.Box*

A box with only height (zero width).

`class matplotlib.mathtext.Vlist`(*elements*, *h=0.0*, *m='additional'*)
Bases: *matplotlib.mathtext.List*

A vertical list of boxes.

vpack(*self*, *h=0.0*, *m='additional'*, *l=inf*)

Compute the dimensions of the resulting boxes, and to adjust the glue if one of those dimensions is pre-specified.

Parameters

h

[float, default: 0] A height.

m

[{'exactly', 'additional'}, default: 'additional'] Whether to produce a box whose height is 'exactly' *w*; or a box with the natural height of the contents, plus *w* ('additional').

l

[float, default: np.inf] The maximum height.

Notes

The defaults produce a box with the natural height of the contents.

class matplotlib.mathtext.Vrule(*state*)

Bases: *matplotlib.mathtext.Rule*

Convenience class to create a vertical rule.

matplotlib.mathtext.get_unicode_index(*symbol*, *math=True*)

Return the integer index (from the Unicode table) of *symbol*.

Parameters

symbol

[str] A single unicode character, a TeX command (e.g. r'pi') or a Type1 symbol name (e.g. 'phi').

math

[bool, default: True] If False, always treat as a single unicode character.

matplotlib.mathtext.math_to_image(*s*, *filename_or_obj*, *prop=None*, *dpi=None*, *format=None*)

Given a math expression, renders it in a closely-clipped bounding box to an image file.

Parameters

s

[str] A math expression. The math portion must be enclosed in dollar signs.

filename_or_obj

[str or path-like or file-like] Where to write the image data.

prop

[*FontProperties*, optional] The size and style of the text.

dpi

[float, optional] The output dpi. If not set, the dpi is determined as for *Figure.savefig*.

format

[str, optional] The output format, e.g., 'svg', 'pdf', 'ps' or 'png'. If not set, the format is determined as for *Figure.savefig*.

18.33 matplotlib.mlab

Numerical python functions written for compatibility with MATLAB commands with the same names. Most numerical python functions can be found in the numpy and scipy libraries. What remains here is code for performing spectral computations.

18.33.1 Spectral functions

cohere

Coherence (normalized cross spectral density)

csd

Cross spectral density using Welch's average periodogram

detrend

Remove the mean or best fit line from an array

psd

Power spectral density using Welch's average periodogram

specgram

Spectrogram (spectrum over segments of time)

complex_spectrum

Return the complex-valued frequency spectrum of a signal

magnitude_spectrum

Return the magnitude of the frequency spectrum of a signal

angle_spectrum

Return the angle (wrapped phase) of the frequency spectrum of a signal

phase_spectrum

Return the phase (unwrapped angle) of the frequency spectrum of a signal

detrend_mean

Remove the mean from a line.

detrend_linear

Remove the best fit line from a line.

detrend_none

Return the original line.

stride_windows

Get all windows in an array in a memory-efficient manner

stride_repeat

Repeat an array in a memory-efficient manner

apply_window

Apply a window along a given axis

class matplotlib.mlab.**GaussianKDE**(*dataset*, *bw_method=None*)

Bases: object

Representation of a kernel-density estimate using Gaussian kernels.

Parameters

dataset

[array-like] Datapoints to estimate from. In case of univariate data this is a 1-D array, otherwise a 2-D array with shape (# of dims, # of data).

bw_method

[str, scalar or callable, optional] The method used to calculate the estimator bandwidth. This can be 'scott', 'silverman', a scalar constant or a callable. If a scalar, this will be used directly as `kde.factor`. If a callable, it should take a *GaussianKDE* instance as only parameter and return a scalar. If None (default), 'scott' is used.

Attributes

dataset

[ndarray] The dataset with which `gaussian_kde` was initialized.

dim

[int] Number of dimensions.

num_dp

[int] Number of datapoints.

factor

[float] The bandwidth factor, obtained from `kde.covariance_factor`, with which the covariance matrix is multiplied.

covariance

[ndarray] The covariance matrix of *dataset*, scaled by the calculated bandwidth (`kde.factor`).

inv_cov

[ndarray] The inverse of *covariance*.

Methods

kde.evaluate(points)	(ndarray) Evaluate the estimated pdf on a provided set of points.
kde(points)	(ndarray) Same as kde.evaluate(points)

`covariance_factor`(*self*)

`evaluate`(*self*, *points*)

Evaluate the estimated pdf on a set of points.

Parameters

points

[(# of dimensions, # of points)-array] Alternatively, a (# of dimensions,) vector can be passed in and treated as a single point.

Returns

(# of points,)-array

The values at each point.

Raises

ValueError

[if the dimensionality of the input points is different] than the dimensionality of the KDE.

`scotts_factor`(*self*)

`silverman_factor`(*self*)

`matplotlib.mlab.`**`angle_spectrum`**(*x*, *Fs=None*, *window=None*, *pad_to=None*, *sides=None*)

Compute the angle of the frequency spectrum (wrapped phase spectrum) of *x*. Data is padded to a length of *pad_to* and the windowing function *window* is applied to the signal.

Parameters

x

[1-D array or sequence] Array or sequence containing the data

Fs

[float, default: 2] The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, *freqs*, in cycles per time unit.

window

[callable or ndarray, default: *window_hanning*] A function or a vector of length *NFFT*. To create window vectors see *window_hanning*, *window_none*, `numpy.blackman`, `numpy.hamming`, `numpy.bartlett`, `scipy.signal`, `scipy.signal.get_window`, etc. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

sides

[{'default', 'onesided', 'twosided'}, optional] Which sides of the spectrum to return. 'default' is one-sided for real data and two-sided for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

pad_to

[int, optional] The number of points to which the data segment is padded when performing the FFT. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to fft(). The default is None, which sets *pad_to* equal to the length of the input signal (i.e. no padding).

Returns

spectrum

[1-D array] The angle of the frequency spectrum (wrapped phase spectrum).

freqs

[1-D array] The frequencies corresponding to the elements in *spectrum*.

See also:

psd

Returns the power spectral density.

complex_spectrum

Returns the complex-valued frequency spectrum.

magnitude_spectrum

Returns the absolute value of the *complex_spectrum*.

angle_spectrum

Returns the angle of the *complex_spectrum*.

phase_spectrum

Returns the phase (unwrapped angle) of the *complex_spectrum*.

specgram

Can return the complex spectrum of segments within the signal.

matplotlib.mlab.**apply_window**(*x*, *window*, *axis=0*, *return_window=None*)

[*Deprecated*] Apply the given window to the given 1D or 2D array along the given axis.

Parameters

x

[1D or 2D array or sequence] Array or sequence containing the data.

window

[function or array.] Either a function to generate a window or an array with length *x*.shape[*axis*]

axis

[int] The axis over which to do the repetition. Must be 0 or 1. The default is 0

return_window

[bool] If true, also return the 1D values of the window that was applied

Notes

Deprecated since version 3.2.

matplotlib.mlab.**cohere**(*x*, *y*, *NFFT=256*, *Fs=2*, *detrend=<function detrend_none at 0x7f5434697820>*, *window=<function window_hanning at 0x7f5434697160>*, *noverlap=0*, *pad_to=None*, *sides='default'*, *scale_by_freq=None*)

The coherence between *x* and *y*. Coherence is the normalized cross spectral density:

$$C_{xy} = \frac{|P_{xy}|^2}{P_{xx}P_{yy}}$$

Parameters

x, y

Array or sequence containing the data

Fs

[float, default: 2] The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, *freqs*, in cycles per time unit.

window

[callable or ndarray, default: *window_hanning*] A function or a vector of length *NFFT*. To create window vectors see *window_hanning*, *window_none*, numpy.blackman, numpy.hamming, numpy.bartlett, scipy.signal, scipy.signal.get_window, etc. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

sides

[{'default', 'onesided', 'twosided'}, optional] Which sides of the spectrum to return. 'default' is one-sided for real data and two-sided for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

pad_to

[int, optional] The number of points to which the data segment is padded when performing the FFT. This can be different from *NFFT*, which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum

distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to fft(). The default is None, which sets *pad_to* equal to *NFFT*

NFFT

[int, default: 256] The number of data points used in each block for the FFT. A power 2 is most efficient. This should *NOT* be used to get zero padding, or the scaling of the result will be incorrect; use *pad_to* for this instead.

detrend

[{'none', 'mean', 'linear'} or callable, default 'none'] The function applied to each segment before fft-ing, designed to remove the mean or linear trend. Unlike in MATLAB, where the *detrend* parameter is a vector, in Matplotlib is it a function. The *mlab* module defines *detrend_none*, *detrend_mean*, and *detrend_linear*, but you can use a custom function as well. You can also use a string to choose one of the functions: 'none' calls *detrend_none*. 'mean' calls *detrend_mean*. 'linear' calls *detrend_linear*.

scale_by_freq

[bool, default: True] Whether the resulting density values should be scaled by the scaling frequency, which gives density in units of Hz^-1. This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

noverlap

[int] The number of points of overlap between blocks. The default value is 0 (no overlap).

Returns

The return value is the tuple (*Cxy*, *f*), where *f* are the frequencies of the coherence vector. For cohere, scaling the individual densities by the sampling frequency has no effect, since the factors cancel out.

See also:

psd(), *csd()*

For information about the methods used to compute P_{xy}, P_{xx} and P_{yy}.

matplotlib.mlab.**complex_spectrum**(*x*, *Fs=None*, *window=None*, *pad_to=None*, *sides=None*)

Compute the complex-valued frequency spectrum of *x*. Data is padded to a length of *pad_to* and the windowing function *window* is applied to the signal.

Parameters

x

[1-D array or sequence] Array or sequence containing the data

Fs

[float, default: 2] The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, *freqs*, in cycles per time unit.

window

[callable or ndarray, default: *window_hanning*] A function or a vector of length *NFFT*. To create window vectors see *window_hanning*, *window_none*, numpy.blackman, numpy.hamming, numpy.bartlett, scipy.signal, scipy.signal.get_window, etc. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

sides

[{'default', 'onesided', 'twosided'}, optional] Which sides of the spectrum to return. 'default' is one-sided for real data and two-sided for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

pad_to

[int, optional] The number of points to which the data segment is padded when performing the FFT. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to fft(). The default is None, which sets *pad_to* equal to the length of the input signal (i.e. no padding).

Returns

spectrum

[1-D array] The complex-valued frequency spectrum.

freqs

[1-D array] The frequencies corresponding to the elements in *spectrum*.

See also:

psd

Returns the power spectral density.

complex_spectrum

Returns the complex-valued frequency spectrum.

magnitude_spectrum

Returns the absolute value of the *complex_spectrum*.

angle_spectrum

Returns the angle of the *complex_spectrum*.

phase_spectrum

Returns the phase (unwrapped angle) of the *complex_spectrum*.

specgram

Can return the complex spectrum of segments within the signal.

matplotlib.mlab.**csd**(*x*, *y*, *NFFT=None*, *Fs=None*, *detrend=None*, *window=None*, *noverlap=None*, *pad_to=None*, *sides=None*, *scale_by_freq=None*)

Compute the cross-spectral density.

The cross spectral density P_{xy} by Welch's average periodogram method. The vectors *x* and *y* are divided into *NFFT* length segments. Each segment is detrended by function *detrend* and windowed by function *window*. *noverlap* gives the length of the overlap between segments. The product of the direct FFTs of *x* and *y* are averaged over each segment to compute P_{xy}, with a scaling to correct for power loss due to windowing.

If len(*x*) < *NFFT* or len(*y*) < *NFFT*, they will be zero padded to *NFFT*.

Parameters

x, y

[1-D arrays or sequences] Arrays or sequences containing the data

Fs

[float, default: 2] The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, *freqs*, in cycles per time unit.

window

[callable or ndarray, default: *window_hanning*] A function or a vector of length *NFFT*. To create window vectors

see *window_hanning*, *window_none*, numpy.blackman, numpy.hamming, numpy.bartlett, scipy.signal, scipy.signal.get_window, etc. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

sides

[{'default', 'onesided', 'twosided'}, optional] Which sides of the spectrum to return. 'default' is one-sided for real data and two-sided for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

pad_to

[int, optional] The number of points to which the data segment is padded when performing the FFT. This can be different from *NFFT*, which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to fft(). The default is None, which sets *pad_to* equal to *NFFT*

NFFT

[int, default: 256] The number of data points used in each block for the FFT. A power 2 is most efficient. This should *NOT* be used to get zero padding, or the scaling of the result will be incorrect; use *pad_to* for this instead.

detrend

[{'none', 'mean', 'linear'} or callable, default 'none'] The function applied to each segment before fft-ing, designed to remove the mean or linear trend. Unlike in MATLAB, where the *detrend* parameter is a vector, in Matplotlib is it a function. The *mlab* module defines *detrend_none*, *detrend_mean*, and *detrend_linear*, but you can use a custom function as well. You can also use a string to choose one of the functions: 'none' calls *detrend_none*. 'mean' calls *detrend_mean*. 'linear' calls *detrend_linear*.

scale_by_freq

[bool, default: True] Whether the resulting density values should be scaled by the scaling frequency, which gives density in units of Hz^-1. This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

noverlap

[int] The number of points of overlap between segments. The default value is 0 (no overlap).

Returns

Pxy

[1-D array] The values for the cross spectrum P_{xy} before scaling (real valued)

freqs

[1-D array] The frequencies corresponding to the elements in *Pxy*

See also:

psd

equivalent to setting `y = x`.

References

Bendat & Piersol -- Random Data: Analysis and Measurement Procedures, John Wiley & Sons (1986)

`matplotlib.mlab.`**detrend**(*x*, *key=None*, *axis=None*)

Return x with its trend removed.

Parameters

x

[array or sequence] Array or sequence containing the data.

key

[{'default', 'constant', 'mean', 'linear', 'none'} or function] The detrending algorithm to use. 'default', 'mean', and 'constant' are the same as *detrend_mean*. 'linear' is the same as *detrend_linear*. 'none' is the same as *detrend_none*. The default is 'mean'. See the corresponding functions for more details regarding the algorithms. Can also be a function that carries out the detrend operation.

axis

[int] The axis along which to do the detrending.

See also:

detrend_mean

Implementation of the 'mean' algorithm.

detrend_linear

Implementation of the 'linear' algorithm.

detrend_none

Implementation of the 'none' algorithm.

matplotlib.mlab.**detrend_linear**(*y*)

Return x minus best fit line; 'linear' detrending.

Parameters

y

[0-D or 1-D array or sequence] Array or sequence containing the data

axis

[int] The axis along which to take the mean. See numpy.mean for a description of this argument.

See also:

detrend_mean

Another detrend algorithm.

detrend_none

Another detrend algorithm.

detrend

A wrapper around all the detrend algorithms.

matplotlib.mlab.**detrend_mean**(*x*, *axis=None*)

Return x minus the mean(x).

Parameters

x

[array or sequence] Array or sequence containing the data Can have any dimensionality

axis

[int] The axis along which to take the mean. See numpy.mean for a description of this argument.

See also:

detrend_linear

Another detrend algorithm.

detrend_none

Another detrend algorithm.

detrend

A wrapper around all the detrend algorithms.

matplotlib.mlab.**detrend_none**(*x*, *axis=None*)

Return x: no detrending.

Parameters

x

[any object] An object containing the data

axis

[int] This parameter is ignored. It is included for compatibility with detrend_mean

See also:

detrend_mean

Another detrend algorithm.

detrend_linear

Another detrend algorithm.

detrend

A wrapper around all the detrend algorithms.

matplotlib.mlab.**magnitude_spectrum**(*x*, *Fs=None*, *window=None*, *pad_to=None*, *sides=None*)

Compute the magnitude (absolute value) of the frequency spectrum of *x*. Data is padded to a length of *pad_to* and the windowing function *window* is applied to the signal.

Parameters

x

[1-D array or sequence] Array or sequence containing the data

Fs

[float, default: 2] The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, *freqs*, in cycles per time unit.

window

[callable or ndarray, default: *window_hanning*] A function or a vector of length *NFFT*. To create window vectors see *window_hanning*, *window_none*, numpy.blackman, numpy.hamming, numpy.bartlett, scipy.signal, scipy.signal.get_window, etc. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

sides

[{'default', 'onesided', 'twosided'}, optional] Which sides of the spectrum to return. 'default' is one-sided for real data and two-sided for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

pad_to

[int, optional] The number of points to which the data segment is padded when performing the FFT. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to fft(). The default is None, which sets *pad_to* equal to the length of the input signal (i.e. no padding).

Returns

spectrum

[1-D array] The magnitude (absolute value) of the frequency spectrum.

freqs

[1-D array] The frequencies corresponding to the elements in *spectrum*.

See also:

psd

Returns the power spectral density.

complex_spectrum

Returns the complex-valued frequency spectrum.

magnitude_spectrum

Returns the absolute value of the *complex_spectrum*.

angle_spectrum

Returns the angle of the *complex_spectrum*.

phase_spectrum

Returns the phase (unwrapped angle) of the *complex_spectrum*.

specgram

Can return the complex spectrum of segments within the signal.

matplotlib.mlab.**phase_spectrum**(*x*, *Fs=None*, *window=None*, *pad_to=None*, *sides=None*)

Compute the phase of the frequency spectrum (unwrapped phase spectrum) of *x*. Data is padded to a length of *pad_to* and the windowing function *window* is applied to the signal.

Parameters

x

[1-D array or sequence] Array or sequence containing the data

Fs

[float, default: 2] The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, *freqs*, in cycles per time unit.

window

[callable or ndarray, default: *window_hanning*] A function or a vector of length *NFFT*. To create window vectors see *window_hanning*, *window_none*, numpy.blackman, numpy.hamming, numpy.bartlett, scipy.signal, scipy.signal.get_window, etc. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

sides

[{'default', 'onesided', 'twosided'}, optional] Which sides of the spectrum to return. 'default' is one-sided for real data and two-sided for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

pad_to

[int, optional] The number of points to which the data segment is padded when performing the FFT. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to fft(). The default is None, which sets *pad_to* equal to the length of the input signal (i.e. no padding).

Returns

spectrum

[1-D array] The phase of the frequency spectrum (unwrapped phase spectrum).

freqs

[1-D array] The frequencies corresponding to the elements in *spectrum*.

See also:

psd

Returns the power spectral density.

complex_spectrum

Returns the complex-valued frequency spectrum.

magnitude_spectrum

Returns the absolute value of the *complex_spectrum*.

angle_spectrum

Returns the angle of the *complex_spectrum*.

phase_spectrum

Returns the phase (unwrapped angle) of the *complex_spectrum*.

specgram

Can return the complex spectrum of segments within the signal.

matplotlib.mlab.**psd**(*x*, *NFFT=None*, *Fs=None*, *detrend=None*, *window=None*, *noverlap=None*, *pad_to=None*, *sides=None*, *scale_by_freq=None*)

Compute the power spectral density.

The power spectral density P_{xx} by Welch's average periodogram method. The vector *x* is divided into *NFFT* length segments. Each segment is detrended by function *detrend* and windowed by function *window*. *noverlap* gives the length of the overlap between segments. The $|\mathrm{fft}(i)|^2$ of each segment i are averaged to compute P_{xx}.

If len(*x*) < *NFFT*, it will be zero padded to *NFFT*.

Parameters

x

[1-D array or sequence] Array or sequence containing the data

Fs

[float, default: 2] The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, *freqs*, in cycles per time unit.

window

[callable or ndarray, default: *window_hanning*] A function or a vector of length *NFFT*. To create window vectors see *window_hanning*, *window_none*, numpy.blackman, numpy.hamming, numpy.bartlett, scipy.signal, scipy.signal.get_window, etc. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

sides

[{'default', 'onesided', 'twosided'}, optional] Which sides of the spectrum to return. 'default' is one-sided for real data and two-sided for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

pad_to

[int, optional] The number of points to which the data segment is padded when performing the FFT. This can be different from *NFFT*, which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to fft(). The default is None, which sets *pad_to* equal to *NFFT*

NFFT

[int, default: 256] The number of data points used in each block for the FFT. A power 2 is most efficient. This should *NOT* be used to get zero padding, or the scaling of the result will be incorrect; use *pad_to* for this instead.

detrend

[{'none', 'mean', 'linear'} or callable, default 'none'] The function applied to each segment before fft-ing, designed to remove the mean or linear trend. Unlike in MATLAB, where the *detrend* parameter is a vector, in Matplotlib is it a function. The *mlab* module defines *detrend_none*, *detrend_mean*, and *detrend_linear*, but you can use a custom function as well. You can also use a string to choose one of the functions: 'none' calls *detrend_none*. 'mean' calls *detrend_mean*. 'linear' calls *detrend_linear*.

scale_by_freq

[bool, default: True] Whether the resulting density values should be scaled by the scaling frequency, which gives density in units of Hz^-1. This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

noverlap

[int] The number of points of overlap between segments. The default value is 0 (no overlap).

Returns

Pxx

[1-D array] The values for the power spectrum P_{xx} (real valued)

freqs

[1-D array] The frequencies corresponding to the elements in *Pxx*

See also:

specgram

specgram differs in the default overlap; in not returning the mean of the segment periodograms; and in returning the times of the segments.

magnitude_spectrum

returns the magnitude spectrum.

csd

returns the spectral density between two signals.

References

Bendat & Piersol -- Random Data: Analysis and Measurement Procedures, John Wiley & Sons (1986)

matplotlib.mlab.**specgram**(*x*, *NFFT=None*, *Fs=None*, *detrend=None*, *window=None*, *noverlap=None*, *pad_to=None*, *sides=None*, *scale_by_freq=None*, *mode=None*)

Compute a spectrogram.

Compute and plot a spectrogram of data in x. Data are split into NFFT length segments and the spectrum of each section is computed. The windowing function window is applied to each segment, and the amount of overlap of each segment is specified with noverlap.

Parameters

x

[array-like] 1-D array or sequence.

Fs

[float, default: 2] The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, *freqs*, in cycles per time unit.

window

[callable or ndarray, default: *window_hanning*] A function or a vector of length *NFFT*. To create window vectors see *window_hanning*, *window_none*, numpy.blackman, numpy.hamming, numpy.bartlett, scipy.signal, scipy.signal.get_window, etc. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

sides

[{'default', 'onesided', 'twosided'}, optional] Which sides of the spectrum to return. 'default' is one-sided for real data and two-sided for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

pad_to

[int, optional] The number of points to which the data segment is padded when performing the FFT. This can be different from *NFFT*, which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to fft(). The default is None, which sets *pad_to* equal to *NFFT*

NFFT

[int, default: 256] The number of data points used in each block for the FFT. A power 2 is most efficient. This should *NOT* be used to get zero padding, or the scaling of the result will be incorrect; use *pad_to* for this instead.

detrend

[{'none', 'mean', 'linear'} or callable, default 'none'] The function applied to each segment before fft-ing, designed to remove the mean or linear trend. Unlike in MATLAB, where the *detrend* parameter is a vector, in Matplotlib is it a function. The *mlab* module defines *detrend_none*, *detrend_mean*, and *detrend_linear*, but you can use a custom function as well. You can also use a string to choose one of the functions: 'none' calls *detrend_none*. 'mean' calls *detrend_mean*. 'linear' calls *detrend_linear*.

scale_by_freq

[bool, default: True] Whether the resulting density values should be scaled by the scaling frequency, which gives density in units of Hz^-1. This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

noverlap

[int, optional] The number of points of overlap between blocks. The default value is 128.

mode

[str, default: 'psd']

What sort of spectrum to use:

'psd'

Returns the power spectral density.

'complex'

Returns the complex-valued frequency spectrum.

'magnitude'

Returns the magnitude spectrum.

'angle'

Returns the phase spectrum without unwrapping.

'phase'

Returns the phase spectrum with unwrapping.

Returns

spectrum

[array-like] 2-D array, columns are the periodograms of successive segments.

freqs

[array-like] 1-D array, frequencies corresponding to the rows in *spectrum*.

t

[array-like] 1-D array, the times corresponding to midpoints of segments (i.e the columns in *spectrum*).

See also:

psd

differs in the overlap and in the return values.

complex_spectrum

similar, but with complex valued frequencies.

magnitude_spectrum

similar single segment when mode is 'magnitude'.

angle_spectrum

similar to single segment when mode is 'angle'.

phase_spectrum

similar to single segment when mode is 'phase'.

Notes

detrend and scale_by_freq only apply when *mode* is set to 'psd'.

matplotlib.mlab.**stride_repeat**(*x*, *n*, *axis=0*)

[*Deprecated*] Repeat the values in an array in a memory-efficient manner. Array x is stacked vertically n times.

Warning: It is not safe to write to the output array. Multiple elements may point to the same piece of memory, so modifying one value may change others.

Parameters

x

[1D array or sequence] Array or sequence containing the data.

n

[int] The number of time to repeat the array.

axis

[int] The axis along which the data will run.

Notes

Deprecated since version 3.2.

References

stackoverflow: Repeat NumPy array without replicating data?

matplotlib.mlab.**stride_windows**(*x*, *n*, *noverlap=None*, *axis=0*)

Get all windows of x with length n as a single array, using strides to avoid data duplication.

> **Warning:** It is not safe to write to the output array. Multiple elements may point to the same piece of memory, so modifying one value may change others.

Parameters

x

[1D array or sequence] Array or sequence containing the data.

n

[int] The number of data points in each window.

noverlap

[int] The overlap between adjacent windows. Default is 0 (no overlap)

axis

[int] The axis along which the windows will run.

References

stackoverflow: Rolling window for 1D arrays in Numpy? stackoverflow: Using strides for an efficient moving average filter

matplotlib.mlab.**window_hanning**(*x*)

Return x times the hanning window of len(x).

See also:

window_none

Another window algorithm.

matplotlib.mlab.**window_none**(*x*)

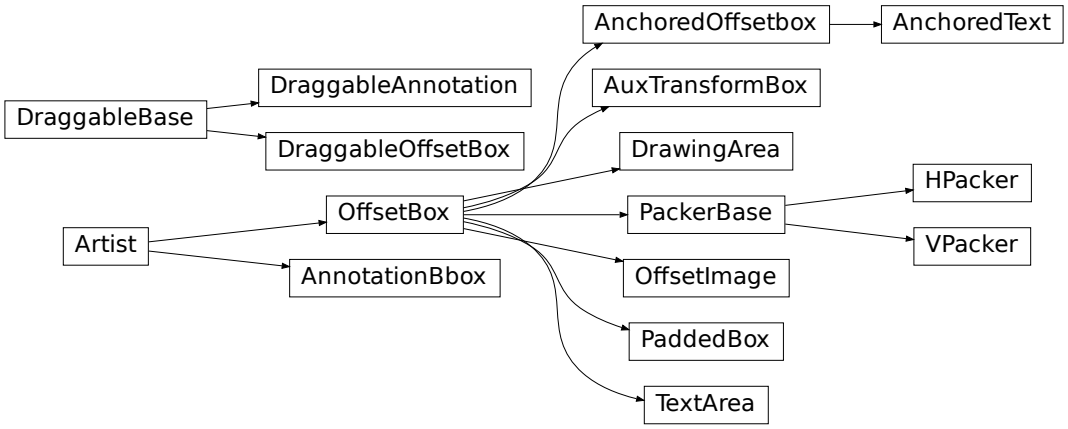
No window function; simply return x.

See also:

window_hanning

Another window algorithm.

18.34 matplotlib.offsetbox



Container classes for *Artist*s.

OffsetBox

The base of all container artists defined in this module.

AnchoredOffsetbox, *AnchoredText*

Anchor and align an arbitrary *Artist* or a text relative to the parent axes or a specific anchor point.

DrawingArea

A container with fixed width and height. Children have a fixed position inside the container and may be clipped.

HPacker, *VPacker*

Containers for layouting their children vertically or horizontally.

PaddedBox

A container to add a padding around an *Artist*.

TextArea

Contains a single *Text* instance.

class matplotlib.offsetbox.**AnchoredOffsetbox**(*loc*, *pad=0.4*, *borderpad=0.5*, *child=None*, *prop=None*, *frameon=True*, *bbox_to_anchor=None*, *bbox_transform=None*, ***kwargs*)

Bases: *matplotlib.offsetbox.OffsetBox*

An offset box placed according to location *loc*.

AnchoredOffsetbox has a single child. When multiple children are needed, use an extra OffsetBox to enclose them. By default, the offset box is anchored against its parent axes. You may explicitly specify the *bbox_to_anchor*.

Parameters

loc

[str] The box location. Supported values:

- 'upper right'
- 'upper left'
- 'lower left'
- 'lower right'
- 'center left'
- 'center right'
- 'lower center'
- 'upper center'
- 'center'

For backward compatibility, numeric values are accepted as well. See the parameter *loc* of *Legend* for details.

pad

[float, default: 0.4] Padding around the child as fraction of the fontsize.

borderpad

[float, default: 0.5] Padding between the offsetbox frame and the *bbox_to_anchor*.

child

[*OffsetBox*] The box that will be anchored.

prop

[*FontProperties*] This is only used as a reference for paddings. If not given, `rcParams["legend.fontsize"]` (default: `'medium'`) is used.

frameon

[bool] Whether to draw a frame around the box.

bbox_to_anchor

[*BboxBase*, 2-tuple, or 4-tuple of floats] Box that is used to position the legend in conjunction with *loc*.

bbox_transform

[None or *matplotlib.transforms.Transform*] The transform for the bounding box (*bbox_to_anchor*).

****kwargs**

All other parameters are passed on to *OffsetBox*.

Notes

See *Legend* for a detailed description of the anchoring mechanism.

`codes = {'center': 10, 'center left': 6, 'center right': 7, 'lower center': 8, 'lower le`

`draw`(*self*, *renderer*)
Update the location of children if necessary and draw them to the given *renderer*.

`get_bbox_to_anchor`(*self*)
Return the bbox that the box is anchored to.

`get_child`(*self*)
Return the child.

`get_children`(*self*)
Return the list of children.

`get_extent`(*self*, *renderer*)
Return the extent of the box as (width, height, x, y).

This is the extent of the child plus the padding.

`get_window_extent`(*self*, *renderer*)
Return the bounding box in display space.

`set_bbox_to_anchor`(*self*, *bbox*, *transform=None*)
Set the bbox that the box is anchored to.

bbox can be a Bbox instance, a list of [left, bottom, width, height], or a list of [left, bottom] where the width and height will be assumed to be zero. The bbox will be transformed to display coordinate by the given transform.

`set_child`(*self*, *child*)
Set the child to be anchored.

`update_frame`(*self*, *bbox*, *fontsize=None*)

`zorder = 5`

`class matplotlib.offsetbox.AnchoredText`(*s*, *loc*, *pad=0.4*, *borderpad=0.5*, *prop=None*, ***kwargs*)
Bases: *matplotlib.offsetbox.AnchoredOffsetbox*

AnchoredOffsetbox with Text.

Parameters

s

[str] Text.

loc

[str] Location code. See *AnchoredOffsetbox*.

pad

[float, default: 0.4] Padding around the text as fraction of the fontsize.

borderpad

[float, default: 0.5] Spacing between the offsetbox frame and the *bbox_to_anchor*.

prop

[dict, optional] Dictionary of keyword parameters to be passed to the *Text* instance contained inside AnchoredText.

****kwargs**

All other parameters are passed to *AnchoredOffsetbox*.

`class matplotlib.offsetbox.AnnotationBbox`(*offsetbox*, *xy*, *xybox=None*, *xycoords='data'*, *boxcoords=None*, *frameon=True*, *pad=0.4*, *annotation_clip=None*, *box_alignment=0.5, 0.5*, *bbox_props=None*, *arrowprops=None*, *fontsize=None*, ***kwargs*)
Bases: *matplotlib.artist.Artist*, `matplotlib.text._AnnotationBase`

Container for an *OffsetBox* referring to a specific position *xy*.

Optionally an arrow pointing from the offsetbox to *xy* can be drawn.

This is like *Annotation*, but with *OffsetBox* instead of *Text*.

Parameters

offsetbox

[*OffsetBox*]

xy

[(float, float)] The point *(x, y)* to annotate. The coordinate system is determined by *xycoords*.

xybox

[(float, float), default: *xy*] The position *(x, y)* to place the text at. The coordinate system is determined by *boxcoords*.

xycoords

[str or *Artist* or *Transform* or callable or (float, float), default: 'data'] The coordinate system that *xy* is given in. See the parameter *xycoords* in *Annotation* for a detailed description.

boxcoords

[str or *Artist* or *Transform* or callable or (float, float), default: value of *xycoords*] The coordinate system that *xybox* is given in. See the parameter *textcoords* in *Annotation* for a detailed description.

frameon

[bool, default: True] Whether to draw a frame around the box.

pad

[float, default: 0.4] Padding around the offsetbox.

box_alignment

[(float, float)] A tuple of two floats for a vertical and horizontal alignment of the offset box w.r.t. the *boxcoords*. The lower-left corner is (0, 0) and upper-right corner is (1, 1).

****kwargs**

Other parameters are identical to *Annotation*.

`property anncoords`

`contains(`*self*, *mouseevent*`)`

Test whether the artist contains the mouse event.

Parameters

mouseevent

[*matplotlib.backend_bases.MouseEvent*]

Returns

contains

[bool] Whether any values are within the radius.

details

[dict] An artist-specific dictionary of details of the event context, such as which points are contained in the pick radius. See the individual Artist subclasses for details.

`draw`(*self*, *renderer*)
Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

Parameters

renderer

[*RendererBase* subclass.]

Notes

This method is overridden in the Artist subclasses.

`get_children`(*self*)
Return a list of the child *Artist*s of this *Artist*.

`get_fontsize`(*self*, *s=<deprecated parameter>*)
Return the fontsize in points.

`get_tightbbox`(*self*, *renderer*)
get tight bounding box in display space.

`get_window_extent`(*self*, *renderer*)
get the bounding box in display space.

`set_figure`(*self*, *fig*)
Set the *Figure* instance the artist belongs to.

Parameters

fig

[*Figure*]

set_fontsize(*self*, *s=None*)
Set the fontsize in points.

If *s* is not given, reset to `rcParams["legend.fontsize"]` (default: `'medium'`).

update_positions(*self*, *renderer*)
Update the pixel positions of the annotated point and the text.

property xyann

zorder = 3

class matplotlib.offsetbox.**AuxTransformBox**(*aux_transform*)
Bases: *matplotlib.offsetbox.OffsetBox*

Offset Box with the aux_transform. Its children will be transformed with the aux_transform first then will be offsetted. The absolute coordinate of the aux_transform is meaning as it will be automatically adjust so that the left-lower corner of the bounding box of children will be set to (0, 0) before the offset transform.

It is similar to drawing area, except that the extent of the box is not predetermined but calculated from the window extent of its children. Furthermore, the extent of the children will be calculated in the transformed coordinate.

add_artist(*self*, *a*)
Add an *Artist* to the container box.

draw(*self*, *renderer*)
Update the location of children if necessary and draw them to the given *renderer*.

get_extent(*self*, *renderer*)
Return a tuple `width, height, xdescent, ydescent` of the box.

get_offset(*self*)
Return offset of the container.

get_transform(*self*)
Return the *Transform* applied to the children

get_window_extent(*self*, *renderer*)
Return the bounding box in display space.

set_offset(*self*, *xy*)
Set the offset of the container.

Parameters

xy

[(float, float)] The (x, y) coordinates of the offset in display units.

set_transform(*self*, *t*)
set_transform is ignored.

class matplotlib.offsetbox.**DraggableAnnotation**(*annotation*, *use_blit=False*)

Bases: *matplotlib.offsetbox.DraggableBase*

save_offset(*self*)

update_offset(*self*, *dx*, *dy*)

class matplotlib.offsetbox.**DraggableBase**(*ref_artist*, *use_blit=False*)

Bases: object

Helper base class for a draggable artist (legend, offsetbox).

Derived classes must override the following methods:

```
def save_offset(self):
    '''
    Called when the object is picked for dragging; should save the
    reference position of the artist.
    '''

def update_offset(self, dx, dy):
    '''
    Called during the dragging; (*dx*, *dy*) is the pixel offset from
    the point where the mouse drag started.
    '''
```

Optionally, you may override the following method:

```
def finalize_offset(self):
    '''Called when the mouse is released.'''
```

In the current implementation of *DraggableLegend* and *DraggableAnnotation*, *update_offset* places the artists in display coordinates, and *finalize_offset* recalculates their position in axes coordinate and set a relevant attribute.

artist_picker(*self*, *artist*, *evt*)

[*Deprecated*]

Notes

Deprecated since version 3.3:

disconnect(*self*)

Disconnect the callbacks.

finalize_offset(*self*)

on_motion(*self*, *evt*)

on_motion_blit(*self*, *evt*)

[*Deprecated*]

Notes

Deprecated since version 3.3:

`on_pick`(*self*, *evt*)

`on_release`(*self*, *event*)

`save_offset`(*self*)

`update_offset`(*self*, *dx*, *dy*)

`class matplotlib.offsetbox.DraggableOffsetBox`(*ref_artist*, *offsetbox*, *use_blit=False*)

Bases: *matplotlib.offsetbox.DraggableBase*

`get_loc_in_canvas`(*self*)

`save_offset`(*self*)

`update_offset`(*self*, *dx*, *dy*)

`class matplotlib.offsetbox.DrawingArea`(*width*, *height*, *xdescent=0.0*, *ydescent=0.0*, *clip=False*)

Bases: *matplotlib.offsetbox.OffsetBox*

The DrawingArea can contain any Artist as a child. The DrawingArea has a fixed width and height. The position of children relative to the parent is fixed. The children can be clipped at the boundaries of the parent.

Parameters

width, height

[float] Width and height of the container box.

xdescent, ydescent

[float] Descent of the box in x- and y-direction.

clip

[bool] Whether to clip the children to the box.

`add_artist`(*self*, *a*)
Add an *Artist* to the container box.

`property clip_children`
If the children of this DrawingArea should be clipped by DrawingArea bounding box.

`draw`(*self*, *renderer*)
Update the location of children if necessary and draw them to the given *renderer*.

`get_extent`(*self*, *renderer*)
Return width, height, xdescent, ydescent of box.

`get_offset`(*self*)
Return offset of the container.

`get_transform`(*self*)
Return the *Transform* applied to the children.

`get_window_extent`(*self*, *renderer*)
Return the bounding box in display space.

`set_offset`(*self*, *xy*)
Set the offset of the container.

Parameters

xy

[(float, float)] The (x, y) coordinates of the offset in display units.

`set_transform`(*self*, *t*)
set_transform is ignored.

`class matplotlib.offsetbox.HPacker`(*pad=None*, *sep=None*, *width=None*, *height=None*, *align='baseline'*, *mode='fixed'*, *children=None*)
Bases: *matplotlib.offsetbox.PackerBase*

The HPacker has its children packed horizontally. It automatically adjusts the relative positions of children at draw time.

Parameters

pad

[float, optional] The boundary padding in points.

sep

[float, optional] The spacing between items in points.

width, height

[float, optional] Width and height of the container box in pixels, calculated if *None*.

align

[{'top', 'bottom', 'left', 'right', 'center', 'baseline'}] Alignment of boxes.

mode

[{'fixed', 'expand', 'equal'}] The packing mode.

- 'fixed' packs the given `Artists` tight with *sep* spacing.
- 'expand' uses the maximal available space to distribute the artists with equal spacing in between.

- 'equal': Each artist an equal fraction of the available space and is left-aligned (or top-aligned) therein.

children

[list of *Artist*] The artists to pack.

Notes

pad and *sep* are in points and will be scaled with the renderer dpi, while *width* and *height* are in in pixels.

`get_extent_offsets`(*self*, *renderer*)

Update offset of the children and return the extent of the box.

Parameters

renderer

[*RendererBase* subclass]

Returns

width

height

xdescent

ydescent

list of (xoffset, yoffset) pairs

`class matplotlib.offsetbox.OffsetBox`(**args*, ***kwargs*)

Bases: *matplotlib.artist.Artist*

The OffsetBox is a simple container artist.

The child artists are meant to be drawn at a relative position to its parent.

Being an artist itself, all parameters are passed on to *Artist*.

`property axes`

The *Axes* instance the artist resides in, or *None*.

`contains`(*self*, *mouseevent*)

Delegate the mouse event contains-check to the children.

As a container, the *OffsetBox* does not respond itself to mouseevents.

Parameters

mouseevent

[*matplotlib.backend_bases.MouseEvent*]

Returns

contains

[bool] Whether any values are within the radius.

details

[dict] An artist-specific dictionary of details of the event context, such as which points are contained in the pick radius. See the individual Artist subclasses for details.

See also:

Artist.contains

`draw`(*self*, *renderer*)
Update the location of children if necessary and draw them to the given *renderer*.

`get_children`(*self*)
Return a list of the child *Artist*s.

`get_extent`(*self*, *renderer*)
Return a tuple `width, height, xdescent, ydescent` of the box.

`get_extent_offsets`(*self*, *renderer*)
Update offset of the children and return the extent of the box.

Parameters

renderer

[*RendererBase* subclass]

Returns

width

height

xdescent

ydescent

list of (xoffset, yoffset) pairs

`get_offset`(*self*, *width*, *height*, *xdescent*, *ydescent*, *renderer*)
Return the offset as a tuple (x, y).

The extent parameters have to be provided to handle the case where the offset is dynamically determined by a callable (see *set_offset*).

Parameters

width, height, xdescent, ydescent

Extent parameters.

renderer

[*RendererBase* subclass]

`get_visible_children`(*self*)
Return a list of the visible child *Artist*s.

`get_window_extent`(*self*, *renderer*)
Return the bounding box (*Bbox*) in display space.

`set_figure`(*self*, *fig*)
Set the *Figure* for the *OffsetBox* and all its children.

Parameters

fig

[*Figure*]

`set_height`(*self*, *height*)
Set the height of the box.

Parameters

height

[float]

`set_offset`(*self*, *xy*)
Set the offset.

Parameters

xy

[(float, float) or callable] The (x, y) coordinates of the offset in display units. These can either be given explicitly as a tuple (x, y), or by providing a function that converts the extent into the offset. This function must have the signature:

```
def offset(width, height, xdescent, ydescent, renderer) -> (float,
↪ float)
```

`set_width`(*self*, *width*)
Set the width of the box.

Parameters

width

[float]

class matplotlib.offsetbox.**OffsetImage**(*arr*, *zoom=1*, *cmap=None*, *norm=None*, *interpolation=None*, *origin=None*, *filternorm=True*, *filterrad=4.0*, *resample=False*, *dpi_cor=True*, ***kwargs*)

Bases: *matplotlib.offsetbox.OffsetBox*

draw(*self*, *renderer*)
Update the location of children if necessary and draw them to the given *renderer*.

get_children(*self*)
Return a list of the child *Artist*s.

get_data(*self*)

get_extent(*self*, *renderer*)
Return a tuple `width, height, xdescent, ydescent` of the box.

get_offset(*self*)
Return offset of the container.

get_window_extent(*self*, *renderer*)
Return the bounding box in display space.

get_zoom(*self*)

set_data(*self*, *arr*)

set_zoom(*self*, *zoom*)

class matplotlib.offsetbox.**PackerBase**(*pad=None*, *sep=None*, *width=None*, *height=None*, *align=None*, *mode=None*, *children=None*)

Bases: *matplotlib.offsetbox.OffsetBox*

Parameters

pad

[float, optional] The boundary padding in points.

sep

[float, optional] The spacing between items in points.

width, height

[float, optional] Width and height of the container box in pixels, calculated if *None*.

align

[{'top', 'bottom', 'left', 'right', 'center', 'baseline'}] Alignment of boxes.

mode

[{'fixed', 'expand', 'equal'}] The packing mode.

- 'fixed' packs the given `Artists` tight with *sep* spacing.
- 'expand' uses the maximal available space to distribute the artists with equal spacing in between.
- 'equal': Each artist an equal fraction of the available space and is left-aligned (or top-aligned) therein.

children

[list of *Artist*] The artists to pack.

Notes

pad and *sep* are in points and will be scaled with the renderer dpi, while *width* and *height* are in in pixels.

class matplotlib.offsetbox.PaddedBox(*child*, *pad=None*, *draw_frame=False*, *patch_attrs=None*)

Bases: *matplotlib.offsetbox.OffsetBox*

A container to add a padding around an *Artist*.

The *PaddedBox* contains a *FancyBboxPatch* that is used to visualize it when rendering.

Parameters

child

[*Artist*] The contained *Artist*.

pad

[float] The padding in points. This will be scaled with the renderer dpi. In contrast *width* and *height* are in *pixels* and thus not scaled.

draw_frame

[bool] Whether to draw the contained *FancyBboxPatch*.

patch_attrs

[dict or None] Additional parameters passed to the contained *FancyBboxPatch*.

`draw`(*self*, *renderer*)
Update the location of children if necessary and draw them to the given *renderer*.

`draw_frame`(*self*, *renderer*)

`get_extent_offsets`(*self*, *renderer*)
Update offset of the children and return the extent of the box.

Parameters

renderer

[*RendererBase* subclass]

Returns

width

height

xdescent

ydescent

list of (xoffset, yoffset) pairs

`update_frame`(*self*, *bbox*, *fontsize=None*)

`class matplotlib.offsetbox.TextArea`(*s*, *textprops=None*, *multilinebaseline=None*, *minimumdescent=True*)
Bases: *matplotlib.offsetbox.OffsetBox*

The TextArea is contains a single Text instance. The text is placed at (0, 0) with baseline+left alignment. The width and height of the TextArea instance is the width and height of the its child text.

Parameters

s

[str] The text to be displayed.

textprops

[dict, optional] Dictionary of keyword parameters to be passed to the *Text* instance contained inside TextArea.

multilinebaseline

[bool, optional] If `True`, baseline for multiline text is adjusted so that it is (approximately) center-aligned with singleline text.

minimumdescent

[bool, optional] If `True`, the box has a minimum descent of "p".

`draw`(*self*, *renderer*)
Update the location of children if necessary and draw them to the given *renderer*.

`get_extent`(*self*, *renderer*)
Return a tuple `width, height, xdescent, ydescent` of the box.

`get_minimumdescent`(*self*)
Get minimumdescent.

`get_multilinebaseline`(*self*)
Get multilinebaseline.

`get_offset`(*self*)
Return offset of the container.

`get_text`(*self*)
Return the string representation of this area's text.

`get_window_extent`(*self*, *renderer*)
Return the bounding box in display space.

`set_minimumdescent`(*self*, *t*)
Set minimumdescent.

If True, extent of the single line text is adjusted so that it has minimum descent of "p"

`set_multilinebaseline`(*self*, *t*)
Set multilinebaseline.

If True, baseline for multiline text is adjusted so that it is (approximately) center-aligned with single-line text.

`set_offset`(*self*, *xy*)
Set the offset of the container.

Parameters

xy

[(float, float)] The (x, y) coordinates of the offset in display units.

`set_text`(*self*, *s*)
Set the text of this area as a string.

`set_transform`(*self*, *t*)
set_transform is ignored.

`class matplotlib.offsetbox.VPacker`(*pad=None*, *sep=None*, *width=None*, *height=None*, *align='baseline'*, *mode='fixed'*, *children=None*)
Bases: *matplotlib.offsetbox.PackerBase*

The VPacker has its children packed vertically. It automatically adjusts the relative positions of children at drawing time.

Parameters

pad

[float, optional] The boundary padding in points.

sep

[float, optional] The spacing between items in points.

width, height

[float, optional] Width and height of the container box in pixels, calculated if *None*.

align

[{'top', 'bottom', 'left', 'right', 'center', 'baseline'}] Alignment of boxes.

mode

[{'fixed', 'expand', 'equal'}] The packing mode.

- 'fixed' packs the given `Artists` tight with *sep* spacing.
- 'expand' uses the maximal available space to distribute the artists with equal spacing in between.
- 'equal': Each artist an equal fraction of the available space and is left-aligned (or top-aligned) therein.

children

[list of *Artist*] The artists to pack.

Notes

pad and *sep* are in points and will be scaled with the renderer dpi, while *width* and *height* are in in pixels.

`get_extent_offsets`(*self*, *renderer*)

Update offset of the children and return the extent of the box.

Parameters

renderer

[*RendererBase* subclass]

Returns

width

height

xdescent

ydescent

list of (xoffset, yoffset) pairs

matplotlib.offsetbox.**bbox_artist**(**args*, ***kwargs*)

18.35 matplotlib.patches

18.35.1 Classes

Arc(xy, width, height[, angle, theta1, theta2])	An elliptical arc, i.e.
Arrow(x, y, dx, dy[, width])	An arrow patch.
ArrowStyle(stylename, **kw)	*ArrowStyle* is a container class which defines several arrowstyle classes, which is used to create an arrow path along a given path.
BoxStyle(stylename, **kw)	*BoxStyle* is a container class which defines several boxstyle classes, which are used for *FancyBboxPatch*.
Circle(xy[, radius])	A circle patch.
CirclePolygon(xy[, radius, resolution])	A polygon-approximation of a circle patch.
ConnectionPatch(xyA, xyB, coordsA[, ...])	A patch that connects two points (possibly in different axes).
ConnectionStyle(stylename, **kw)	*ConnectionStyle* is a container class which defines several connectionstyle classes, which is used to create a path between two points.
Ellipse(xy, width, height[, angle])	A scale-free ellipse.
FancyArrow(x, y, dx, dy[, width, ...])	Like Arrow, but lets you set head width and head height independently.
FancyArrowPatch([posA, posB, path, ...])	A fancy arrow patch.
FancyBboxPatch(xy, width, height[, ...])	A fancy box around a rectangle with lower left at *xy* = (*x*, *y*) with specified width and height.
Patch([edgecolor, facecolor, color, ...])	A patch is a 2D artist with a face color and an edge color.
PathPatch(path, **kwargs)	A general polycurve path patch.
Polygon(xy[, closed])	A general polygon patch.
Rectangle(xy, width, height[, angle])	A rectangle defined via an anchor point *xy* and its *width* and *height*.

continues on next page

Table 146 – continued from previous page

RegularPolygon(xy, numVertices[, radius, ...])	A regular polygon patch.
Shadow(patch, ox, oy[, props])	Create a shadow of the given *patch*.
Wedge(center, r, theta1, theta2[, width])	Wedge shaped patch.

matplotlib.patches.Arc

`class matplotlib.patches.Arc(`*xy*, *width*, *height*, *angle=0.0*, *theta1=0.0*, *theta2=360.0*, ***kwargs*`)`

Bases: *matplotlib.patches.Ellipse*

An elliptical arc, i.e. a segment of an ellipse.

Due to internal optimizations, there are certain restrictions on using Arc:

- The arc cannot be filled.
- The arc must be used in an *Axes* instance. It can not be added directly to a *Figure* because it is optimized to only render the segments that are inside the axes bounding box with high resolution.

Parameters

xy

[(float, float)] The center of the ellipse.

width

[float] The length of the horizontal axis.

height

[float] The length of the vertical axis.

angle

[float] Rotation of the ellipse in degrees (counterclockwise).

theta1, theta2

[float, default: 0, 360] Starting and ending angles of the arc in degrees. These values are relative to *angle*, e.g. if *angle* = 45 and *theta1* = 90 the absolute starting angle is 135. Default *theta1* = 0, *theta2* = 360, i.e. a complete ellipse. The arc is drawn in the counterclockwise direction. Angles greater than or equal to 360, or smaller than 0, are represented by an equivalent angle in the range [0, 360), by taking the input value mod 360.

Other Parameters

****kwargs**

[*Patch* properties] Most *Patch* properties are supported as keyword arguments, with the exception of *fill* and *facecolor* because filling is not supported.

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and retur
alpha	float or None
animated	bool
antialiased or aa	unknown
capstyle	{'butt', 'round', 'projecting'}
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color	color
contains	unknown
edgecolor or ec	color or None or 'auto'
facecolor or fc	color or None
figure	*Figure*
fill	bool
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float or None
path_effects	*AbstractPathEffect*
picker	None or bool or callable
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
visible	bool
zorder	float

`__init__`(*self*, *xy*, *width*, *height*, *angle=0.0*, *theta1=0.0*, *theta2=360.0*, ***kwargs*)

Parameters

xy

[(float, float)] The center of the ellipse.

width

[float] The length of the horizontal axis.

height

[float] The length of the vertical axis.

angle

[float] Rotation of the ellipse in degrees (counterclockwise).

theta1, theta2

[float, default: 0, 360] Starting and ending angles of the arc in degrees. These values are relative to *angle*, e.g. if *angle* = 45 and *theta1* = 90 the absolute starting angle is 135. Default *theta1* = 0, *theta2* = 360, i.e. a complete ellipse. The arc is drawn in the counterclockwise direction. Angles greater than or equal to 360, or smaller than 0, are represented by an equivalent angle in the range [0, 360), by taking the input value mod 360.

Other Parameters

****kwargs**

[*Patch* properties] Most *Patch* properties are supported as keyword arguments, with the exception of *fill* and *facecolor* because filling is not supported.

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and retur
alpha	float or None
animated	bool
antialiased or aa	unknown
capstyle	{'butt', 'round', 'projecting'}
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color	color
contains	unknown
edgecolor or ec	color or None or 'auto'
facecolor or fc	color or None
figure	*Figure*
fill	bool
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object

Table 148 – continued from previous page

Property	Description
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float or None
path_effects	*AbstractPathEffect*
picker	None or bool or callable
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
visible	bool
zorder	float

`__module__ = 'matplotlib.patches'`

`__str__`(*self*)
Return str(self).

`draw`(*self*, *renderer*)
Draw the arc to the given *renderer*.

Notes

Ellipses are normally drawn using an approximation that uses eight cubic Bezier splines. The error of this approximation is 1.89818e-6, according to this unverified source:

> Lancaster, Don. *Approximating a Circle or an Ellipse Using Four Bezier Cubic Splines.*
>
> https://www.tinaja.com/glib/ellipse4.pdf

There is a use case where very large ellipses must be drawn with very high accuracy, and it is too expensive to render the entire ellipse with enough segments (either splines or line segments). Therefore, in the case where either radius of the ellipse is large enough that the error of the spline approximation will be visible (greater than one pixel offset from the ideal), a different technique is used.

In that case, only the visible parts of the ellipse are drawn, with each visible arc using a fixed number of spline segments (8). The algorithm proceeds as follows:

1. The points where the ellipse intersects the axes bounding box are located. (This is done be performing an inverse transformation on the axes bbox such that it is relative to the unit circle -- this makes the intersection calculation much easier than doing rotated ellipse intersection directly).

This uses the "line intersecting a circle" algorithm from:

Vince, John. *Geometry for Computer Graphics: Formulae, Examples & Proofs.* London: Springer-Verlag, 2005.

2. The angles of each of the intersection points are calculated.
3. Proceeding counterclockwise starting in the positive x-direction, each of the visible arc-segments between the pairs of vertices are drawn using the Bezier arc approximation technique implemented in *Path.arc*.

Examples using `matplotlib.patches.Arc`

- sphx_glr_gallery_units_ellipse_with_units.py
- *Sample plots in Matplotlib*

matplotlib.patches.Arrow

`class matplotlib.patches.Arrow`(*x*, *y*, *dx*, *dy*, *width=1.0*, ***kwargs*)

Bases: *matplotlib.patches.Patch*

An arrow patch.

Draws an arrow from (*x*, *y*) to (*x* + *dx*, *y* + *dy*). The width of the arrow is scaled by *width*.

Parameters

x

[float] x coordinate of the arrow tail.

y

[float] y coordinate of the arrow tail.

dx

[float] Arrow length in the x direction.

dy

[float] Arrow length in the y direction.

width

[float, default: 1] Scale factor for the width of the arrow. With a default value of 1, the tail width is 0.2 and head width is 0.6.

****kwargs**

Keyword arguments control the *Patch* properties:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and retur
alpha	float or None
animated	bool
antialiased or aa	unknown
capstyle	{'butt', 'round', 'projecting'}
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color	color
contains	unknown
edgecolor or ec	color or None or 'auto'
facecolor or fc	color or None
figure	*Figure*
fill	bool
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float or None
path_effects	*AbstractPathEffect*
picker	None or bool or callable
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
visible	bool
zorder	float

See also:

FancyArrow

Patch that allows independent control of the head and tail properties.

`__init__`(*self*, *x*, *y*, *dx*, *dy*, *width=1.0*, ***kwargs*)

Draws an arrow from (*x*, *y*) to (*x* + *dx*, *y* + *dy*). The width of the arrow is scaled by *width*.

Parameters

x

[float] x coordinate of the arrow tail.

y

[float] y coordinate of the arrow tail.

dx

[float] Arrow length in the x direction.

dy

[float] Arrow length in the y direction.

width

[float, default: 1] Scale factor for the width of the arrow. With a default value of 1, the tail width is 0.2 and head width is 0.6.

****kwargs**

Keyword arguments control the *Patch* properties:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and retur
alpha	float or None
animated	bool
antialiased or aa	unknown
capstyle	{'butt', 'round', 'projecting'}
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color	color
contains	unknown
edgecolor or ec	color or None or 'auto'
facecolor or fc	color or None
figure	*Figure*
fill	bool
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float or None
path_effects	*AbstractPathEffect*
picker	None or bool or callable
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str

Table 150 – continued from previous page

Property	Description
visible	bool
zorder	float

See also:

FancyArrow

Patch that allows independent control of the head and tail properties.

__module__ = 'matplotlib.patches'

__str__(*self*)
Return str(self).

get_patch_transform(*self*)
Return the *Transform* instance mapping patch coordinates to data coordinates.

For example, one may define a patch of a circle which represents a radius of 5 by providing coordinates for a unit circle, and a transform which scales the coordinates (the patch coordinate) by 5.

get_path(*self*)
Return the path of this patch.

Examples using `matplotlib.patches.Arrow`

- sphx_glr_gallery_shapes_and_collections_arrow_guide.py
- sphx_glr_gallery_shapes_and_collections_artist_reference.py

matplotlib.patches.ArrowStyle

class matplotlib.patches.**ArrowStyle**(*stylename*, ***kw*)
Bases: `matplotlib.patches._Style`

ArrowStyle is a container class which defines several arrowstyle classes, which is used to create an arrow path along a given path. These are mainly used with *FancyArrowPatch*.

A arrowstyle object can be either created as:

```
ArrowStyle.Fancy(head_length=.4, head_width=.4, tail_width=.4)
```

or:

```
ArrowStyle("Fancy", head_length=.4, head_width=.4, tail_width=.4)
```

or:

```
ArrowStyle("Fancy, head_length=.4, head_width=.4, tail_width=.4")
```

The following classes are defined

Class	Name	Attrs
Curve	-	None
CurveB	->	head_length=0.4, head_width=0.2
BracketB	-[	widthB=1.0, lengthB=0.2, angleB=None
CurveFilledB	-|>	head_length=0.4, head_width=0.2
CurveA	<-	head_length=0.4, head_width=0.2
CurveAB	<->	head_length=0.4, head_width=0.2
CurveFilledA	<|-	head_length=0.4, head_width=0.2
CurveFilledAB	<|-|>	head_length=0.4, head_width=0.2
BracketA	]-	widthA=1.0, lengthA=0.2, angleA=None
BracketAB	]-[	widthA=1.0, lengthA=0.2, angleA=None, widthB=1.0, lengthB=0.2, angleB=None
Fancy	fancy	head_length=0.4, head_width=0.4, tail_width=0.4
Simple	simple	head_length=0.5, head_width=0.5, tail_width=0.2
Wedge	wedge	tail_width=0.3, shrink_factor=0.5
BarAB	|-|	widthA=1.0, angleA=None, widthB=1.0, angleB=None

An instance of any arrow style class is a callable object, whose call signature is:

```
__call__(self, path, mutation_size, linewidth, aspect_ratio=1.)
```

and it returns a tuple of a *Path* instance and a boolean value. *path* is a *Path* instance along which the arrow will be drawn. *mutation_size* and *aspect_ratio* have the same meaning as in *BoxStyle*. *linewidth* is a line width to be stroked. This is meant to be used to correct the location of the head so that it does not overshoot the destination point, but not all classes support it.

Return the instance of the subclass with the given style name.

`class BarAB(`*widthA=1.0*, *angleA=None*, *widthB=1.0*, *angleB=None*`)`

Bases: `matplotlib.patches.ArrowStyle._Bracket`

An arrow with vertical bars `|` at both ends.

Parameters

widthA

[float, default: 1.0] Width of the bracket.

angleA

[float, default: None] Angle between the bracket and the line.

widthB

[float, default: 1.0] Width of the bracket.

angleB

[float, default: None] Angle between the bracket and the line.

`__init__`(*self*, *widthA=1.0*, *angleA=None*, *widthB=1.0*, *angleB=None*)

Parameters

widthA

[float, default: 1.0] Width of the bracket.

angleA

[float, default: None] Angle between the bracket and the line.

widthB

[float, default: 1.0] Width of the bracket.

angleB

[float, default: None] Angle between the bracket and the line.

`__module__ = 'matplotlib.patches'`

`class BracketA`(*widthA=1.0*, *lengthA=0.2*, *angleA=None*)

Bases: `matplotlib.patches.ArrowStyle._Bracket`

An arrow with an outward square bracket at its start.

Parameters

widthA

[float, default: 1.0] Width of the bracket.

lengthA

[float, default: 0.2] Length of the bracket.

angleA

[float, default: None] Angle between the bracket and the line.

`__init__`(*self*, *widthA=1.0*, *lengthA=0.2*, *angleA=None*)

Parameters

widthA

[float, default: 1.0] Width of the bracket.

lengthA

[float, default: 0.2] Length of the bracket.

angleA

[float, default: None] Angle between the bracket and the line.

`__module__ = 'matplotlib.patches'`

`class BracketAB(`*widthA=1.0*, *lengthA=0.2*, *angleA=None*, *widthB=1.0*, *lengthB=0.2*, *angleB=None*`)`

Bases: `matplotlib.patches.ArrowStyle._Bracket`

An arrow with outward square brackets at both ends.

Parameters

widthA

[float, default: 1.0] Width of the bracket.

lengthA

[float, default: 0.2] Length of the bracket.

angleA

[float, default: None] Angle between the bracket and the line.

widthB

[float, default: 1.0] Width of the bracket.

lengthB

[float, default: 0.2] Length of the bracket.

angleB

[float, default: None] Angle between the bracket and the line.

`__init__(`*self*, *widthA=1.0*, *lengthA=0.2*, *angleA=None*, *widthB=1.0*, *lengthB=0.2*, *angleB=None*`)`

Parameters

widthA

[float, default: 1.0] Width of the bracket.

lengthA

[float, default: 0.2] Length of the bracket.

angleA

[float, default: None] Angle between the bracket and the line.

widthB

[float, default: 1.0] Width of the bracket.

lengthB

[float, default: 0.2] Length of the bracket.

angleB

[float, default: None] Angle between the bracket and the line.

`__module__ = 'matplotlib.patches'`

`class BracketB(`*widthB=1.0*, *lengthB=0.2*, *angleB=None*`)`

Bases: `matplotlib.patches.ArrowStyle._Bracket`

An arrow with an outward square bracket at its end.

Parameters

widthB

[float, default: 1.0] Width of the bracket.

lengthB

[float, default: 0.2] Length of the bracket.

angleB

[float, default: None] Angle between the bracket and the line.

`__init__(`*self*, *widthB=1.0*, *lengthB=0.2*, *angleB=None*`)`

Parameters

widthB

[float, default: 1.0] Width of the bracket.

lengthB

[float, default: 0.2] Length of the bracket.

angleB

[float, default: None] Angle between the bracket and the line.

`__module__ = 'matplotlib.patches'`

`class Curve`

Bases: `matplotlib.patches.ArrowStyle._Curve`

A simple curve without any arrow head.

The arrows are drawn if *beginarrow* and/or *endarrow* are true. *head_length* and *head_width* determines the size of the arrow relative to the *mutation scale*. The arrowhead at the begin (or end) is closed if fillbegin (or fillend) is True.

`__init__(`*self*`)`

The arrows are drawn if *beginarrow* and/or *endarrow* are true. *head_length* and *head_width* determines the size of the arrow relative to the *mutation scale*. The arrowhead at the begin (or end) is closed if fillbegin (or fillend) is True.

__module__ = 'matplotlib.patches'

class CurveA(*head_length=0.4*, *head_width=0.2*)
Bases: matplotlib.patches.ArrowStyle._Curve

An arrow with a head at its begin point.

Parameters

head_length

[float, default: 0.4] Length of the arrow head.

head_width

[float, default: 0.2] Width of the arrow head.

__init__(*self*, *head_length=0.4*, *head_width=0.2*)

Parameters

head_length

[float, default: 0.4] Length of the arrow head.

head_width

[float, default: 0.2] Width of the arrow head.

__module__ = 'matplotlib.patches'

class CurveAB(*head_length=0.4*, *head_width=0.2*)
Bases: matplotlib.patches.ArrowStyle._Curve

An arrow with heads both at the begin and the end point.

Parameters

head_length

[float, default: 0.4] Length of the arrow head.

head_width

[float, default: 0.2] Width of the arrow head.

__init__(*self*, *head_length=0.4*, *head_width=0.2*)

Parameters

head_length

[float, default: 0.4] Length of the arrow head.

head_width

[float, default: 0.2] Width of the arrow head.

__module__ = 'matplotlib.patches'

class CurveB(*head_length=0.4*, *head_width=0.2*)
Bases: matplotlib.patches.ArrowStyle._Curve

An arrow with a head at its end point.

Parameters

head_length

[float, default: 0.4] Length of the arrow head.

head_width

[float, default: 0.2] Width of the arrow head.

__init__(*self*, *head_length=0.4*, *head_width=0.2*)

Parameters

head_length

[float, default: 0.4] Length of the arrow head.

head_width

[float, default: 0.2] Width of the arrow head.

__module__ = 'matplotlib.patches'

class CurveFilledA(*head_length=0.4*, *head_width=0.2*)
Bases: matplotlib.patches.ArrowStyle._Curve

An arrow with filled triangle head at the begin.

Parameters

head_length

[float, default: 0.4] Length of the arrow head.

head_width

[float, default: 0.2] Width of the arrow head.

__init__(*self*, *head_length=0.4*, *head_width=0.2*)

Parameters

head_length

[float, default: 0.4] Length of the arrow head.

head_width

[float, default: 0.2] Width of the arrow head.

__module__ = 'matplotlib.patches'

class CurveFilledAB(*head_length=0.4*, *head_width=0.2*)
Bases: matplotlib.patches.ArrowStyle._Curve

An arrow with filled triangle heads at both ends.

Parameters

head_length

[float, default: 0.4] Length of the arrow head.

head_width

[float, default: 0.2] Width of the arrow head.

`__init__`(*self*, *head_length=0.4*, *head_width=0.2*)

Parameters

head_length

[float, default: 0.4] Length of the arrow head.

head_width

[float, default: 0.2] Width of the arrow head.

`__module__ = 'matplotlib.patches'`

`class CurveFilledB`(*head_length=0.4*, *head_width=0.2*)

Bases: `matplotlib.patches.ArrowStyle._Curve`

An arrow with filled triangle head at the end.

Parameters

head_length

[float, default: 0.4] Length of the arrow head.

head_width

[float, default: 0.2] Width of the arrow head.

`__init__`(*self*, *head_length=0.4*, *head_width=0.2*)

Parameters

head_length

[float, default: 0.4] Length of the arrow head.

head_width

[float, default: 0.2] Width of the arrow head.

`__module__ = 'matplotlib.patches'`

`class Fancy`(*head_length=0.4*, *head_width=0.4*, *tail_width=0.4*)

Bases: `matplotlib.patches.ArrowStyle._Base`

A fancy arrow. Only works with a quadratic Bezier curve.

Parameters

head_length

[float, default: 0.4] Length of the arrow head.

head_width

[float, default: 0.4] Width of the arrow head.

tail_width

[float, default: 0.4] Width of the arrow tail.

`__init__`(*self*, *head_length=0.4*, *head_width=0.4*, *tail_width=0.4*)

Parameters

head_length

[float, default: 0.4] Length of the arrow head.

head_width

[float, default: 0.4] Width of the arrow head.

tail_width

[float, default: 0.4] Width of the arrow tail.

`__module__ = 'matplotlib.patches'`

`transmute`(*self*, *path*, *mutation_size*, *linewidth*)

The transmute method is the very core of the ArrowStyle class and must be overridden in the subclasses. It receives the path object along which the arrow will be drawn, and the mutation_size, with which the arrow head etc. will be scaled. The linewidth may be used to adjust the path so that it does not pass beyond the given points. It returns a tuple of a Path instance and a boolean. The boolean value indicate whether the path can be filled or not. The return value can also be a list of paths and list of booleans of a same length.

`class Simple`(*head_length=0.5*, *head_width=0.5*, *tail_width=0.2*)

Bases: `matplotlib.patches.ArrowStyle._Base`

A simple arrow. Only works with a quadratic Bezier curve.

Parameters

head_length

[float, default: 0.5] Length of the arrow head.

head_width

[float, default: 0.5] Width of the arrow head.

tail_width

[float, default: 0.2] Width of the arrow tail.

`__init__`(*self*, *head_length=0.5*, *head_width=0.5*, *tail_width=0.2*)

Parameters

head_length

[float, default: 0.5] Length of the arrow head.

head_width

[float, default: 0.5] Width of the arrow head.

tail_width

[float, default: 0.2] Width of the arrow tail.

`__module__ = 'matplotlib.patches'`

`transmute`(*self*, *path*, *mutation_size*, *linewidth*)

The transmute method is the very core of the ArrowStyle class and must be overridden in the subclasses. It receives the path object along which the arrow will be drawn, and the mutation_size, with which the arrow head etc. will be scaled. The linewidth may be used to adjust the path so that it does not pass beyond the given points. It returns a tuple of a Path instance and a boolean. The boolean value indicate whether the path can be filled or not. The return value can also be a list of paths and list of booleans of a same length.

`class Wedge`(*tail_width=0.3*, *shrink_factor=0.5*)

Bases: `matplotlib.patches.ArrowStyle._Base`

Wedge(?) shape. Only works with a quadratic Bezier curve. The begin point has a width of the tail_width and the end point has a width of 0. At the middle, the width is shrink_factor*tail_width.

Parameters

tail_width

[float, default: 0.3] Width of the tail.

shrink_factor

[float, default: 0.5] Fraction of the arrow width at the middle point.

`__init__`(*self*, *tail_width=0.3*, *shrink_factor=0.5*)

Parameters

tail_width

[float, default: 0.3] Width of the tail.

shrink_factor

[float, default: 0.5] Fraction of the arrow width at the middle point.

`__module__ = 'matplotlib.patches'`

`transmute`(*self*, *path*, *mutation_size*, *linewidth*)

The transmute method is the very core of the ArrowStyle class and must be overridden in the subclasses. It receives the path object along which the arrow will be drawn, and the mutation_size, with which the arrow head etc. will be scaled. The linewidth may be used to adjust the path so that it does not pass beyond the given points. It returns a tuple of a Path instance and a boolean. The boolean value indicate whether the path can be filled or not. The return value can also be a list of paths and list of booleans of a same length.

```
__module__ = 'matplotlib.patches'
```

Examples using `matplotlib.patches.ArrowStyle`

- sphx_glr_gallery_text_labels_and_annotations_fancyarrow_demo.py

matplotlib.patches.BoxStyle

`class matplotlib.patches.BoxStyle`(*stylename*, ***kw*)

Bases: `matplotlib.patches._Style`

BoxStyle is a container class which defines several boxstyle classes, which are used for *FancyBboxPatch*.

A style object can be created as:

```
BoxStyle.Round(pad=0.2)
```

or:

```
BoxStyle("Round", pad=0.2)
```

or:

```
BoxStyle("Round, pad=0.2")
```

The following boxstyle classes are defined.

Class	Name	Attrs
Circle	`circle`	pad=0.3
DArrow	`darrow`	pad=0.3
LArrow	`larrow`	pad=0.3
RArrow	`rarrow`	pad=0.3
Round	`round`	pad=0.3, rounding_size=None
Round4	`round4`	pad=0.3, rounding_size=None
Roundtooth	`roundtooth`	pad=0.3, tooth_size=None
Sawtooth	`sawtooth`	pad=0.3, tooth_size=None
Square	`square`	pad=0.3

An instance of any boxstyle class is an callable object, whose call signature is:

```
__call__(self, x0, y0, width, height, mutation_size, aspect_ratio=1.)
```

and returns a *Path* instance. *x0*, *y0*, *width* and *height* specify the location and size of the box to be drawn. *mutation_scale* determines the overall size of the mutation (by which I mean the transformation of the rectangle to the fancy box). *mutation_aspect* determines the aspect-ratio of the mutation.

Return the instance of the subclass with the given style name.

`class Circle(`*pad=0.3*`)`

Bases: `matplotlib.patches.BoxStyle._Base`

A circular box.

Parameters

pad

[float, default: 0.3] The amount of padding around the original box.

`__init__(`*self*, *pad=0.3*`)`

Initialize self. See help(type(self)) for accurate signature.

`__module__ = 'matplotlib.patches'`

`transmute(`*self*, *x0*, *y0*, *width*, *height*, *mutation_size*`)`

Return the *Path* outlining the given rectangle.

`class DArrow(`*pad=0.3*`)`

Bases: `matplotlib.patches.BoxStyle._Base`

A box in the shape of a two-way arrow.

Parameters

pad

[float, default: 0.3] The amount of padding around the original box.

`__init__(`*self*, *pad=0.3*`)`

Initialize self. See help(type(self)) for accurate signature.

`__module__ = 'matplotlib.patches'`

`transmute(`*self*, *x0*, *y0*, *width*, *height*, *mutation_size*`)`

Return the *Path* outlining the given rectangle.

`class LArrow(`*pad=0.3*`)`

Bases: `matplotlib.patches.BoxStyle._Base`

A box in the shape of a left-pointing arrow.

Parameters

pad

[float, default: 0.3] The amount of padding around the original box.

`__init__`(*self*, *pad=0.3*)
Initialize self. See help(type(self)) for accurate signature.

`__module__ = 'matplotlib.patches'`

`transmute`(*self*, *x0*, *y0*, *width*, *height*, *mutation_size*)
Return the *Path* outlining the given rectangle.

`class RArrow`(*pad=0.3*)
Bases: *matplotlib.patches.BoxStyle.LArrow*

A box in the shape of a right-pointing arrow.

Parameters

pad

[float, default: 0.3] The amount of padding around the original box.

`__init__`(*self*, *pad=0.3*)
Initialize self. See help(type(self)) for accurate signature.

`__module__ = 'matplotlib.patches'`

`transmute`(*self*, *x0*, *y0*, *width*, *height*, *mutation_size*)
Return the *Path* outlining the given rectangle.

`class Round`(*pad=0.3*, *rounding_size=None*)
Bases: `matplotlib.patches.BoxStyle._Base`

A box with round corners.

Parameters

pad

[float, default: 0.3] The amount of padding around the original box.

rounding_size

[float, default: *pad*] Radius of the corners.

`__init__`(*self*, *pad=0.3*, *rounding_size=None*)
Initialize self. See help(type(self)) for accurate signature.

`__module__ = 'matplotlib.patches'`

transmute(*self*, *x0*, *y0*, *width*, *height*, *mutation_size*)
Return the *Path* outlining the given rectangle.

class Round4(*pad=0.3*, *rounding_size=None*)
Bases: matplotlib.patches.BoxStyle._Base

A box with rounded edges.

Parameters

pad

[float, default: 0.3] The amount of padding around the original box.

rounding_size

[float, default: *pad*/2] Rounding of edges.

__init__(*self*, *pad=0.3*, *rounding_size=None*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'matplotlib.patches'

transmute(*self*, *x0*, *y0*, *width*, *height*, *mutation_size*)
Return the *Path* outlining the given rectangle.

class Roundtooth(*pad=0.3*, *tooth_size=None*)
Bases: *matplotlib.patches.BoxStyle.Sawtooth*

A box with a rounded sawtooth outline.

Parameters

pad

[float, default: 0.3] The amount of padding around the original box.

tooth_size

[float, default: *pad*/2] Size of the sawtooth.

__init__(*self*, *pad=0.3*, *tooth_size=None*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'matplotlib.patches'

transmute(*self*, *x0*, *y0*, *width*, *height*, *mutation_size*)
Return the *Path* outlining the given rectangle.

class Sawtooth(*pad=0.3*, *tooth_size=None*)
Bases: matplotlib.patches.BoxStyle._Base

A box with a sawtooth outline.

Parameters

pad

[float, default: 0.3] The amount of padding around the original box.

tooth_size

[float, default: *pad*/2] Size of the sawtooth.

`__init__`(*self*, *pad=0.3*, *tooth_size=None*)
Initialize self. See help(type(self)) for accurate signature.

`__module__ = 'matplotlib.patches'`

`transmute`(*self*, *x0*, *y0*, *width*, *height*, *mutation_size*)
Return the *Path* outlining the given rectangle.

`class Square`(*pad=0.3*)
Bases: `matplotlib.patches.BoxStyle._Base`

A square box.

Parameters

pad

[float, default: 0.3] The amount of padding around the original box.

`__init__`(*self*, *pad=0.3*)
Initialize self. See help(type(self)) for accurate signature.

`__module__ = 'matplotlib.patches'`

`transmute`(*self*, *x0*, *y0*, *width*, *height*, *mutation_size*)
Return the *Path* outlining the given rectangle.

`__module__ = 'matplotlib.patches'`

Examples using `matplotlib.patches.BoxStyle`

- sphx_glr_gallery_shapes_and_collections_artist_reference.py
- sphx_glr_gallery_shapes_and_collections_fancybox_demo.py

matplotlib.patches.Circle

class matplotlib.patches.Circle(*xy*, *radius=5*, ***kwargs*)

Bases: *matplotlib.patches.Ellipse*

A circle patch.

Create a true circle at center *xy* = (*x*, *y*) with given *radius*.

Unlike *CirclePolygon* which is a polygonal approximation, this uses Bezier splines and is much closer to a scale-free circle.

Valid keyword arguments are:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and retur
alpha	float or None
animated	bool
antialiased or aa	unknown
capstyle	{'butt', 'round', 'projecting'}
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color	color
contains	unknown
edgecolor or ec	color or None or 'auto'
facecolor or fc	color or None
figure	*Figure*
fill	bool
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float or None
path_effects	*AbstractPathEffect*
picker	None or bool or callable
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
visible	bool
zorder	float

__init__(*self*, *xy*, *radius=5*, ***kwargs*)
Create a true circle at center *xy* = (*x*, *y*) with given *radius*.

Unlike *CirclePolygon* which is a polygonal approximation, this uses Bezier splines and is much closer to a scale-free circle.

Valid keyword arguments are:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and retur
alpha	float or None
animated	bool
antialiased or aa	unknown
capstyle	{'butt', 'round', 'projecting'}
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color	color
contains	unknown
edgecolor or ec	color or None or 'auto'
facecolor or fc	color or None
figure	*Figure*
fill	bool
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float or None
path_effects	*AbstractPathEffect*
picker	None or bool or callable
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
visible	bool
zorder	float

__module__ = 'matplotlib.patches'

__str__(*self*)
Return str(self).

get_radius(*self*)
Return the radius of the circle.

`property radius`
Return the radius of the circle.

`set_radius`(*self*, *radius*)
Set the radius of the circle.

Parameters

radius

[float]

Examples using `matplotlib.patches.Circle`

- sphx_glr_gallery_images_contours_and_fields_image_clip_path.py
- sphx_glr_gallery_subplots_axes_and_figures_axes_box_aspect.py
- sphx_glr_gallery_text_labels_and_annotations_demo_annotation_box.py
- sphx_glr_gallery_text_labels_and_annotations_fancyarrow_demo.py
- sphx_glr_gallery_shapes_and_collections_artist_reference.py
- sphx_glr_gallery_shapes_and_collections_dolphin.py
- sphx_glr_gallery_shapes_and_collections_donut.py
- sphx_glr_gallery_shapes_and_collections_patch_collection.py
- sphx_glr_gallery_style_sheets_ggplot.py
- sphx_glr_gallery_style_sheets_grayscale.py
- sphx_glr_gallery_style_sheets_style_sheets_reference.py
- sphx_glr_gallery_axes_grid1_simple_anchored_artists.py
- sphx_glr_gallery_showcase_anatomy.py
- sphx_glr_gallery_event_handling_looking_glass.py
- sphx_glr_gallery_misc_anchored_artists.py
- sphx_glr_gallery_misc_custom_projection.py
- sphx_glr_gallery_mplot3d_pathpatch3d.py
- sphx_glr_gallery_specialty_plots_radar_chart.py
- sphx_glr_gallery_userdemo_anchored_box02.py
- *Artist tutorial*
- *Legend guide*
- *Transformations Tutorial*

matplotlib.patches.CirclePolygon

class matplotlib.patches.CirclePolygon(*xy*, *radius=5*, *resolution=20*, ***kwargs*)

Bases: *matplotlib.patches.RegularPolygon*

A polygon-approximation of a circle patch.

Create a circle at *xy* = (*x*, *y*) with given *radius*.

This circle is approximated by a regular polygon with *resolution* sides. For a smoother circle drawn with splines, see *Circle*.

Valid keyword arguments are:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and retur
alpha	float or None
animated	bool
antialiased or aa	unknown
capstyle	{'butt', 'round', 'projecting'}
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color	color
contains	unknown
edgecolor or ec	color or None or 'auto'
facecolor or fc	color or None
figure	*Figure*
fill	bool
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float or None
path_effects	*AbstractPathEffect*
picker	None or bool or callable
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
visible	bool
zorder	float

__init__(*self*, *xy*, *radius=5*, *resolution=20*, ***kwargs*)

Create a circle at *xy* = (*x*, *y*) with given *radius*.

This circle is approximated by a regular polygon with *resolution* sides. For a smoother circle drawn with splines, see *Circle*.

Valid keyword arguments are:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and retur
alpha	float or None
animated	bool
antialiased or aa	unknown
capstyle	{'butt', 'round', 'projecting'}
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color	color
contains	unknown
edgecolor or ec	color or None or 'auto'
facecolor or fc	color or None
figure	*Figure*
fill	bool
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float or None
path_effects	*AbstractPathEffect*
picker	None or bool or callable
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
visible	bool
zorder	float

__module__ = 'matplotlib.patches'

__str__(*self*)

Return str(self).

Examples using `matplotlib.patches.CirclePolygon`

matplotlib.patches.ConnectionPatch

class matplotlib.patches.**ConnectionPatch**(*xyA*, *xyB*, *coordsA*, *coordsB=None*, *axesA=None*, *axesB=None*, *arrowstyle='-'*, *connectionstyle='arc3'*, *patchA=None*, *patchB=None*, *shrinkA=0.0*, *shrinkB=0.0*, *mutation_scale=10.0*, *mutation_aspect=None*, *clip_on=False*, *dpi_cor=1.0*, ***kwargs*)

Bases: *matplotlib.patches.FancyArrowPatch*

A patch that connects two points (possibly in different axes).

Connect point *xyA* in *coordsA* with point *xyB* in *coordsB*.

Valid keys are

Key	Description
arrowstyle	the arrow style
connectionstyle	the connection style
relpos	default is (0.5, 0.5)
patchA	default is bounding box of the text
patchB	default is None
shrinkA	default is 2 points
shrinkB	default is 2 points
mutation_scale	default is text size (in points)
mutation_aspect	default is 1.
?	any key for *matplotlib.patches.PathPatch*

coordsA and *coordsB* are strings that indicate the coordinates of *xyA* and *xyB*.

Property	Description
'figure points'	points from the lower left corner of the figure
'figure pixels'	pixels from the lower left corner of the figure
'figure fraction'	0, 0 is lower left of figure and 1, 1 is upper right
'axes points'	points from lower left corner of axes
'axes pixels'	pixels from lower left corner of axes
'axes fraction'	0, 0 is lower left of axes and 1, 1 is upper right
'data'	use the coordinate system of the object being annotated (default)
'offset points'	offset (in points) from the *xy* value
'polar'	you can specify *theta*, *r* for the annotation, even in cartesian plots. Note that if you are using a polar axes, you do not need to specify polar for the coordinate system since that is the native "data" coordinate system.

Alternatively they can be set to any valid *Transform*.

Note: Using *ConnectionPatch* across two *Axes* instances is not directly compatible with *constrained layout*. Add the artist directly to the *Figure* instead of adding it to a specific Axes.

```
fig, ax = plt.subplots(1, 2, constrained_layout=True)
con = ConnectionPatch(..., axesA=ax[0], axesB=ax[1])
fig.add_artist(con)
```

`__init__`(*self*, *xyA*, *xyB*, *coordsA*, *coordsB=None*, *axesA=None*, *axesB=None*, *arrowstyle='-'*, *connectionstyle='arc3'*, *patchA=None*, *patchB=None*, *shrinkA=0.0*, *shrinkB=0.0*, *mutation_scale=10.0*, *mutation_aspect=None*, *clip_on=False*, *dpi_cor=1.0*, ***kwargs*)

Connect point *xyA* in *coordsA* with point *xyB* in *coordsB*.

Valid keys are

Key	Description
arrowstyle	the arrow style
connectionstyle	the connection style
relpos	default is (0.5, 0.5)
patchA	default is bounding box of the text
patchB	default is None
shrinkA	default is 2 points
shrinkB	default is 2 points
mutation_scale	default is text size (in points)
mutation_aspect	default is 1.
?	any key for *matplotlib.patches.PathPatch*

coordsA and *coordsB* are strings that indicate the coordinates of *xyA* and *xyB*.

Property	Description
'figure points'	points from the lower left corner of the figure
'figure pixels'	pixels from the lower left corner of the figure
'figure fraction'	0, 0 is lower left of figure and 1, 1 is upper right
'axes points'	points from lower left corner of axes
'axes pixels'	pixels from lower left corner of axes
'axes fraction'	0, 0 is lower left of axes and 1, 1 is upper right
'data'	use the coordinate system of the object being annotated (default)
'offset points'	offset (in points) from the *xy* value
'polar'	you can specify *theta*, *r* for the annotation, even in cartesian plots. Note that if you are using a polar axes, you do not need to specify polar for the coordinate system since that is the native "data" coordinate system.

Alternatively they can be set to any valid *Transform*.

Note: Using *ConnectionPatch* across two *Axes* instances is not directly compatible with *constrained layout*. Add the artist directly to the *Figure* instead of adding it to a specific Axes.

```
fig, ax = plt.subplots(1, 2, constrained_layout=True)
con = ConnectionPatch(..., axesA=ax[0], axesB=ax[1])
fig.add_artist(con)
```

__module__ = 'matplotlib.patches'

__str__(*self*)
Return str(self).

draw(*self*, *renderer*)
Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

Parameters

renderer

[*RendererBase* subclass.]

Notes

This method is overridden in the Artist subclasses.

get_annotation_clip(*self*)
Return the clipping behavior.

See *set_annotation_clip* for the meaning of the return value.

get_path_in_displaycoord(*self*)
Return the mutated path of the arrow in display coordinates.

set_annotation_clip(*self*, *b*)
Set the clipping behavior.

Parameters

b

[bool or None]

- *False*: The annotation will always be drawn regardless of its position.

- *True*: The annotation will only be drawn if `self.xy` is inside the axes.
- *None*: The annotation will only be drawn if `self.xy` is inside the axes and `self.xycoords == "data"`.

Examples using `matplotlib.patches.ConnectionPatch`

- sphx_glr_gallery_pie_and_polar_charts_bar_of_pie.py
- sphx_glr_gallery_userdemo_connect_simple01.py
- *Constrained Layout Guide*

matplotlib.patches.ConnectionStyle

class matplotlib.patches.**ConnectionStyle**(*stylename*, ***kw*)

Bases: `matplotlib.patches._Style`

ConnectionStyle is a container class which defines several connectionstyle classes, which is used to create a path between two points. These are mainly used with *FancyArrowPatch*.

A connectionstyle object can be either created as:

```
ConnectionStyle.Arc3(rad=0.2)
```

or:

```
ConnectionStyle("Arc3", rad=0.2)
```

or:

```
ConnectionStyle("Arc3, rad=0.2")
```

The following classes are defined

Class	Name	Attrs
Angle	`angle`	angleA=90, angleB=0, rad=0.0
Angle3	`angle3`	angleA=90, angleB=0
Arc	`arc`	angleA=0, angleB=0, armA=None, armB=None, rad=0.0
Arc3	`arc3`	rad=0.0
Bar	`bar`	armA=0.0, armB=0.0, fraction=0.3, angle=None

An instance of any connection style class is an callable object, whose call signature is:

```
__call__(self, posA, posB,
         patchA=None, patchB=None,
         shrinkA=2., shrinkB=2.)
```

and it returns a *Path* instance. *posA* and *posB* are tuples of (x, y) coordinates of the two points to be connected. *patchA* (or *patchB*) is given, the returned path is clipped so that it start (or end) from the boundary of the patch. The path is further shrunk by *shrinkA* (or *shrinkB*) which is given in points.

Return the instance of the subclass with the given style name.

`class Angle(`*angleA=90*, *angleB=0*, *rad=0.0*`)`

Bases: `matplotlib.patches.ConnectionStyle._Base`

Creates a piecewise continuous quadratic Bezier path between two points. The path has a one passing-through point placed at the intersecting point of two lines which cross the start and end point, and have a slope of angleA and angleB, respectively. The connecting edges are rounded with *rad*.

angleA

starting angle of the path

angleB

ending angle of the path

rad

rounding radius of the edge

`__init__(`*self*, *angleA=90*, *angleB=0*, *rad=0.0*`)`

angleA

starting angle of the path

angleB

ending angle of the path

rad

rounding radius of the edge

`__module__ = 'matplotlib.patches'`

`connect(`*self*, *posA*, *posB*`)`

`class Angle3(`*angleA=90*, *angleB=0*`)`

Bases: `matplotlib.patches.ConnectionStyle._Base`

Creates a simple quadratic Bezier curve between two points. The middle control points is placed at the intersecting point of two lines which cross the start and end point, and have a slope of angleA and angleB, respectively.

angleA

starting angle of the path

angleB

ending angle of the path

`__init__`(*self*, *angleA=90*, *angleB=0*)

angleA

starting angle of the path

angleB

ending angle of the path

`__module__ = 'matplotlib.patches'`

`connect`(*self*, *posA*, *posB*)

`class Arc`(*angleA=0*, *angleB=0*, *armA=None*, *armB=None*, *rad=0.0*)

Bases: `matplotlib.patches.ConnectionStyle._Base`

Creates a piecewise continuous quadratic Bezier path between two points. The path can have two passing-through points, a point placed at the distance of armA and angle of angleA from point A, another point with respect to point B. The edges are rounded with *rad*.

***angleA* :**

starting angle of the path

***angleB* :**

ending angle of the path

***armA* :**

length of the starting arm

***armB* :**

length of the ending arm

***rad* :**

rounding radius of the edges

`__init__`(*self*, *angleA=0*, *angleB=0*, *armA=None*, *armB=None*, *rad=0.0*)

***angleA* :**

starting angle of the path

***angleB* :**

ending angle of the path

***armA* :**

length of the starting arm

***armB* :**

length of the ending arm

***rad* :**

rounding radius of the edges

`__module__ = 'matplotlib.patches'`

`connect`(*self*, *posA*, *posB*)

`class Arc3`(*rad=0.0*)

Bases: `matplotlib.patches.ConnectionStyle._Base`

Creates a simple quadratic Bezier curve between two points. The curve is created so that the middle control point (C1) is located at the same distance from the start (C0) and end points(C2) and the distance of the C1 to the line connecting C0-C2 is *rad* times the distance of C0-C2.

rad

curvature of the curve.

`__init__`(*self*, *rad=0.0*)

rad

curvature of the curve.

`__module__ = 'matplotlib.patches'`

`connect`(*self*, *posA*, *posB*)

`class Bar`(*armA=0.0*, *armB=0.0*, *fraction=0.3*, *angle=None*)

Bases: `matplotlib.patches.ConnectionStyle._Base`

A line with *angle* between A and B with *armA* and *armB*. One of the arms is extended so that they are connected in a right angle. The length of armA is determined by (*armA* + *fraction* x AB distance). Same for armB.

Parameters

armA

[float] minimum length of armA

armB

[float] minimum length of armB

fraction

[float] a fraction of the distance between two points that will be added to armA and armB.

angle

[float or None] angle of the connecting line (if None, parallel to A and B)

`__init__`(*self*, *armA=0.0*, *armB=0.0*, *fraction=0.3*, *angle=None*)

Parameters

armA

[float] minimum length of armA

armB

[float] minimum length of armB

fraction

[float] a fraction of the distance between two points that will be added to armA and armB.

angle

[float or None] angle of the connecting line (if None, parallel to A and B)

```
__module__ = 'matplotlib.patches'
```

```
connect(self, posA, posB)
```

```
__module__ = 'matplotlib.patches'
```

Examples using `matplotlib.patches.ConnectionStyle`

matplotlib.patches.Ellipse

```
class matplotlib.patches.Ellipse(xy, width, height, angle=0, **kwargs)
```

Bases: *matplotlib.patches.Patch*

A scale-free ellipse.

Parameters

xy

[(float, float)] xy coordinates of ellipse centre.

width

[float] Total length (diameter) of horizontal axis.

height

[float] Total length (diameter) of vertical axis.

angle

[float, default: 0] Rotation in degrees anti-clockwise.

Notes

Valid keyword arguments are:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and retur
alpha	float or None
animated	bool
antialiased or aa	unknown
capstyle	{'butt', 'round', 'projecting'}
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color	color
contains	unknown
edgecolor or ec	color or None or 'auto'
facecolor or fc	color or None
figure	*Figure*
fill	bool
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float or None
path_effects	*AbstractPathEffect*
picker	None or bool or callable
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
visible	bool
zorder	float

`__init__`(*self*, *xy*, *width*, *height*, *angle=0*, ***kwargs*)

Parameters

xy

[(float, float)] xy coordinates of ellipse centre.

width

[float] Total length (diameter) of horizontal axis.

height

[float] Total length (diameter) of vertical axis.

angle

[float, default: 0] Rotation in degrees anti-clockwise.

Notes

Valid keyword arguments are:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and retur
alpha	float or None
animated	bool
antialiased or aa	unknown
capstyle	{'butt', 'round', 'projecting'}
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color	color
contains	unknown
edgecolor or ec	color or None or 'auto'
facecolor or fc	color or None
figure	*Figure*
fill	bool
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float or None
path_effects	*AbstractPathEffect*
picker	None or bool or callable
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
visible	bool
zorder	float

__module__ = 'matplotlib.patches'

__str__(*self*)
Return str(self).

property angle
Return the angle of the ellipse.

property center
Return the center of the ellipse.

get_angle(*self*)
Return the angle of the ellipse.

get_center(*self*)
Return the center of the ellipse.

get_height(*self*)
Return the height of the ellipse.

get_patch_transform(*self*)
Return the *Transform* instance mapping patch coordinates to data coordinates.

For example, one may define a patch of a circle which represents a radius of 5 by providing coordinates for a unit circle, and a transform which scales the coordinates (the patch coordinate) by 5.

get_path(*self*)
Return the path of the ellipse.

get_width(*self*)
Return the width of the ellipse.

property height
Return the height of the ellipse.

set_angle(*self*, *angle*)
Set the angle of the ellipse.

Parameters

angle

[float]

set_center(*self*, *xy*)
Set the center of the ellipse.

Parameters

xy

[(float, float)]

`set_height`(*self*, *height*)
Set the height of the ellipse.

Parameters

height

[float]

`set_width`(*self*, *width*)
Set the width of the ellipse.

Parameters

width

[float]

`property width`
Return the width of the ellipse.

Examples using `matplotlib.patches.Ellipse`

- sphx_glr_gallery_images_contours_and_fields_image_clip_path.py
- sphx_glr_gallery_subplots_axes_and_figures_axes_box_aspect.py
- sphx_glr_gallery_statistics_confidence_ellipse.py
- sphx_glr_gallery_text_labels_and_annotations_annotation_demo.py
- sphx_glr_gallery_text_labels_and_annotations_demo_annotation_box.py
- sphx_glr_gallery_text_labels_and_annotations_fancyarrow_demo.py
- sphx_glr_gallery_shapes_and_collections_artist_reference.py
- sphx_glr_gallery_shapes_and_collections_dolphin.py
- sphx_glr_gallery_shapes_and_collections_donut.py
- sphx_glr_gallery_shapes_and_collections_ellipse_demo.py
- sphx_glr_gallery_shapes_and_collections_hatch_demo.py
- sphx_glr_gallery_shapes_and_collections_patch_collection.py
- sphx_glr_gallery_style_sheets_ggplot.py
- sphx_glr_gallery_style_sheets_grayscale.py
- sphx_glr_gallery_style_sheets_style_sheets_reference.py
- sphx_glr_gallery_showcase_anatomy.py
- sphx_glr_gallery_event_handling_looking_glass.py

- sphx_glr_gallery_misc_anchored_artists.py
- sphx_glr_gallery_misc_custom_projection.py
- sphx_glr_gallery_mplot3d_pathpatch3d.py
- sphx_glr_gallery_specialty_plots_radar_chart.py
- sphx_glr_gallery_units_ellipse_with_units.py
- sphx_glr_gallery_userdemo_anchored_box02.py
- sphx_glr_gallery_userdemo_anchored_box03.py
- sphx_glr_gallery_userdemo_anchored_box04.py
- sphx_glr_gallery_userdemo_annotate_explain.py
- sphx_glr_gallery_userdemo_simple_annotate01.py
- *Legend guide*
- *Transformations Tutorial*

matplotlib.patches.FancyArrow

class matplotlib.patches.**FancyArrow**(*x*, *y*, *dx*, *dy*, *width=0.001*, *length_includes_head=False*, *head_width=None*, *head_length=None*, *shape='full'*, *overhang=0*, *head_starts_at_zero=False*, ***kwargs*)

Bases: *matplotlib.patches.Polygon*

Like Arrow, but lets you set head width and head height independently.

Parameters

width: float, default: 0.001

Width of full arrow tail.

length_includes_head: bool, default: False

True if head is to be counted in calculating the length.

head_width: float or None, default: 3*width

Total width of the full arrow head.

head_length: float or None, default: 1.5*head_width

Length of arrow head.

shape: ['full', 'left', 'right'], default: 'full'

Draw the left-half, right-half, or full arrow.

overhang: float, default: 0

Fraction that the arrow is swept back (0 overhang means triangular shape). Can be negative or greater than one.

head_starts_at_zero: bool, default: False

If True, the head starts being drawn at coordinate 0 instead of ending at coordinate 0.

****kwargs**

Patch properties:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and retur
alpha	float or None
animated	bool
antialiased or aa	unknown
capstyle	{'butt', 'round', 'projecting'}
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color	color
contains	unknown
edgecolor or ec	color or None or 'auto'
facecolor or fc	color or None
figure	*Figure*
fill	bool
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float or None
path_effects	*AbstractPathEffect*
picker	None or bool or callable
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
visible	bool
zorder	float

__init__(*self*, *x*, *y*, *dx*, *dy*, *width=0.001*, *length_includes_head=False*, *head_width=None*, *head_length=None*, *shape='full'*, *overhang=0*, *head_starts_at_zero=False*, ***kwargs*)

Parameters

width: float, default: 0.001

Width of full arrow tail.

length_includes_head: bool, default: False

True if head is to be counted in calculating the length.

head_width: float or None, default: 3*width

Total width of the full arrow head.

head_length: float or None, default: 1.5*head_width

Length of arrow head.

shape: ['full', 'left', 'right'], default: 'full'

Draw the left-half, right-half, or full arrow.

overhang: float, default: 0

Fraction that the arrow is swept back (0 overhang means triangular shape). Can be negative or greater than one.

head_starts_at_zero: bool, default: False

If True, the head starts being drawn at coordinate 0 instead of ending at coordinate 0.

****kwargs**

Patch properties:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and retur
alpha	float or None
animated	bool
antialiased or aa	unknown
capstyle	{'butt', 'round', 'projecting'}
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color	color
contains	unknown
edgecolor or ec	color or None or 'auto'
facecolor or fc	color or None
figure	*Figure*

Table 158 – continued from previous page

Property	Description
fill	bool
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float or None
path_effects	*AbstractPathEffect*
picker	None or bool or callable
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
visible	bool
zorder	float

__module__ = 'matplotlib.patches'

__str__(*self*)
Return str(self).

Examples using matplotlib.patches.FancyArrow

matplotlib.patches.FancyArrowPatch

class matplotlib.patches.**FancyArrowPatch**(*posA=None*, *posB=None*, *path=None*, *arrowstyle='simple'*, *connectionstyle='arc3'*, *patchA=None*, *patchB=None*, *shrinkA=2*, *shrinkB=2*, *mutation_scale=1*, *mutation_aspect=None*, *dpi_cor=1*, ***kwargs*)

Bases: *matplotlib.patches.Patch*

A fancy arrow patch. It draws an arrow using the *ArrowStyle*.

The head and tail positions are fixed at the specified start and end points of the arrow, but the size and shape (in display coordinates) of the arrow does not change when the axis is moved or zoomed.

There are two ways for defining an arrow:

- If *posA* and *posB* are given, a path connecting two points is created according to *connectionstyle*. The path will be clipped with *patchA* and *patchB* and further shrunken by *shrinkA* and *shrinkB*. An arrow is drawn along this resulting path using the *arrowstyle* parameter.
- Alternatively if *path* is provided, an arrow is drawn along this path and *patchA*, *patchB*, *shrinkA*, and *shrinkB* are ignored.

Parameters

posA, posB

[(float, float), default: None] (x, y) coordinates of arrow tail and arrow head respectively.

path

[*Path*, default: None] If provided, an arrow is drawn along this path and *patchA*, *patchB*, *shrinkA*, and *shrinkB* are ignored.

arrowstyle

[str or *ArrowStyle*, default: 'simple']

The *ArrowStyle* with which the fancy arrow is drawn. If a string, it should be one of the available arrowstyle names, with optional comma-separated attributes. The optional attributes are meant to be scaled with the *mutation_scale*. The following arrow styles are available:

Class	Name	Attrs
Curve	`-`	None
CurveB	`->`	head_length=0.4, head_width=0.2
BracketB	`-[`	widthB=1.0, lengthB=0.2, angleB=None
CurveFilledB	`-|>`	head_length=0.4, head_width=0.2
CurveA	`<-`	head_length=0.4, head_width=0.2
CurveAB	`<->`	head_length=0.4, head_width=0.2
CurveFilledA	`<|-`	head_length=0.4, head_width=0.2
CurveFilledAB	`<|-|>`	head_length=0.4, head_width=0.2
BracketA	`]-`	widthA=1.0, lengthA=0.2, angleA=None
BracketAB	`]-[`	widthA=1.0, lengthA=0.2, angleA=None, widthB=1.0, lengthB=0.2, angleB=None
Fancy	`fancy`	head_length=0.4, head_width=0.4, tail_width=0.4
Simple	`simple`	head_length=0.5, head_width=0.5, tail_width=0.2
Wedge	`wedge`	tail_width=0.3, shrink_factor=0.5
BarAB	`|-|`	widthA=1.0, angleA=None, widthB=1.0, angleB=None

connectionstyle

[str or *ConnectionStyle* or None, optional, default: 'arc3']

The *ConnectionStyle* with which *posA* and *posB* are connected. If a string, it should be one of the available connectionstyle names, with optional comma-separated attributes. The following connection styles are available:

Class	Name	Attrs
Angle	`angle`	angleA=90, angleB=0, rad=0.0
Angle3	`angle3`	angleA=90, angleB=0
Arc	`arc`	angleA=0, angleB=0, armA=None, armB=None, rad=0.0
Arc3	`arc3`	rad=0.0
Bar	`bar`	armA=0.0, armB=0.0, fraction=0.3, angle=None

patchA, patchB

[*Patch*, default: None] Head and tail patches, respectively.

shrinkA, shrinkB

[float, default: 2] Shrinking factor of the tail and head of the arrow respectively.

mutation_scale

[float, default: 1] Value with which attributes of *arrowstyle* (e.g., *head_length*) will be scaled.

mutation_aspect

[None or float, default: None] The height of the rectangle will be squeezed by this value before the mutation and the mutated box will be stretched by the inverse of it.

dpi_cor

[float, default: 1] dpi_cor is currently used for linewidth-related things and shrink factor. Mutation scale is affected by this.

Other Parameters

****kwargs**

[*Patch* properties, optional] Here is a list of available *Patch* properties:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and retur
alpha	float or None
animated	bool
antialiased or aa	unknown
capstyle	{'butt', 'round', 'projecting'}
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color	color
contains	unknown
edgecolor or ec	color or None or 'auto'
facecolor or fc	color or None
figure	*Figure*
fill	bool
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object

Table 159 – continued from previous page

Property	Description
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float or None
path_effects	*AbstractPathEffect*
picker	None or bool or callable
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
visible	bool
zorder	float

In contrast to other patches, the default `capstyle` and `joinstyle` for *FancyArrowPatch* are set to `"round"`.

`__init__`(*self*, *posA=None*, *posB=None*, *path=None*, *arrowstyle='simple'*, *connectionstyle='arc3'*, *patchA=None*, *patchB=None*, *shrinkA=2*, *shrinkB=2*, *mutation_scale=1*, *mutation_aspect=None*, *dpi_cor=1*, ***kwargs*)

There are two ways for defining an arrow:

- If *posA* and *posB* are given, a path connecting two points is created according to *connectionstyle*. The path will be clipped with *patchA* and *patchB* and further shrunken by *shrinkA* and *shrinkB*. An arrow is drawn along this resulting path using the *arrowstyle* parameter.
- Alternatively if *path* is provided, an arrow is drawn along this path and *patchA*, *patchB*, *shrinkA*, and *shrinkB* are ignored.

Parameters

posA, posB

[(float, float), default: None] (x, y) coordinates of arrow tail and arrow head respectively.

path

[*Path*, default: None] If provided, an arrow is drawn along this path and *patchA*, *patchB*, *shrinkA*, and *shrinkB* are ignored.

arrowstyle

[str or *ArrowStyle*, default: 'simple']

The *ArrowStyle* with which the fancy arrow is drawn. If a string, it should be one of the available arrowstyle names, with optional comma-separated attributes. The optional at-

tributes are meant to be scaled with the *mutation_scale*. The following arrow styles are available:

Class	Name	Attrs
Curve	-	None
CurveB	->	head_length=0.4, head_width=0.2
BracketB	-[	widthB=1.0, lengthB=0.2, angleB=None
CurveFilledB	-|>	head_length=0.4, head_width=0.2
CurveA	<-	head_length=0.4, head_width=0.2
CurveAB	<->	head_length=0.4, head_width=0.2
CurveFilledA	<|-	head_length=0.4, head_width=0.2
CurveFilledAB	<|-|>	head_length=0.4, head_width=0.2
BracketA	]-	widthA=1.0, lengthA=0.2, angleA=None
BracketAB	]-[	widthA=1.0, lengthA=0.2, angleA=None, widthB=1.0, lengthB=0.2, angleB=None
Fancy	fancy	head_length=0.4, head_width=0.4, tail_width=0.4
Simple	simple	head_length=0.5, head_width=0.5, tail_width=0.2
Wedge	wedge	tail_width=0.3, shrink_factor=0.5
BarAB	|-|	widthA=1.0, angleA=None, widthB=1.0, angleB=None

connectionstyle

[str or *ConnectionStyle* or None, optional, default: 'arc3']

The *ConnectionStyle* with which *posA* and *posB* are connected. If a string, it should be one of the available connectionstyle names, with optional comma-separated attributes. The following connection styles are available:

Class	Name	Attrs
Angle	angle	angleA=90, angleB=0, rad=0.0
Angle3	angle3	angleA=90, angleB=0
Arc	arc	angleA=0, angleB=0, armA=None, armB=None, rad=0.0
Arc3	arc3	rad=0.0
Bar	bar	armA=0.0, armB=0.0, fraction=0.3, angle=None

patchA, patchB

[*Patch*, default: None] Head and tail patches, respectively.

shrinkA, shrinkB

[float, default: 2] Shrinking factor of the tail and head of the arrow respectively.

mutation_scale

[float, default: 1] Value with which attributes of *arrowstyle* (e.g., *head_length*) will be scaled.

mutation_aspect

[None or float, default: None] The height of the rectangle will be squeezed by this value before the mutation and the mutated box will be stretched by the inverse of it.

dpi_cor

[float, default: 1] dpi_cor is currently used for linewidth-related things and shrink factor. Mutation scale is affected by this.

Other Parameters

****kwargs**

[*Patch* properties, optional] Here is a list of available *Patch* properties:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and retur
alpha	float or None
animated	bool
antialiased or aa	unknown
capstyle	{'butt', 'round', 'projecting'}
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color	color
contains	unknown
edgecolor or ec	color or None or 'auto'
facecolor or fc	color or None
figure	*Figure*
fill	bool
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}

Table 160 – continued from previous page

Property	Description
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float or None
path_effects	*AbstractPathEffect*
picker	None or bool or callable
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
visible	bool
zorder	float

In contrast to other patches, the default `capstyle` and `joinstyle` for *FancyArrowPatch* are set to `"round"`.

`__module__ = 'matplotlib.patches'`

`__str__`(*self*)
Return str(self).

`draw`(*self*, *renderer*)
Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

Parameters

renderer

[*RendererBase* subclass.]

Notes

This method is overridden in the Artist subclasses.

`get_arrowstyle`(*self*)
Return the arrowstyle object.

`get_connectionstyle`(*self*)
Return the *ConnectionStyle* used.

`get_dpi_cor`(*self*)
dpi_cor is currently used for linewidth-related things and shrink factor. Mutation scale is affected by this.

Returns

scalar

`get_mutation_aspect`(*self*)
Return the aspect ratio of the bbox mutation.

`get_mutation_scale`(*self*)
Return the mutation scale.

Returns

scalar

`get_path`(*self*)
Return the path of the arrow in the data coordinates. Use get_path_in_displaycoord() method to retrieve the arrow path in display coordinates.

`get_path_in_displaycoord`(*self*)
Return the mutated path of the arrow in display coordinates.

`set_arrowstyle`(*self*, *arrowstyle=None*, ***kw*)
Set the arrow style. Old attributes are forgotten. Without arguments (or with `arrowstyle=None`) returns available box styles as a list of strings.

Parameters

arrowstyle

[None or ArrowStyle or str, default: None] Can be a string with arrowstyle name with optional comma-separated attributes, e.g.:

```
set_arrowstyle("Fancy,head_length=0.2")
```

Alternatively attributes can be provided as keywords, e.g.:

```
set_arrowstyle("fancy", head_length=0.2)
```

`set_connectionstyle`(*self*, *connectionstyle*, ***kw*)
Set the connection style. Old attributes are forgotten.

Parameters

connectionstyle

[str or *ConnectionStyle* or None, optional] Can be a string with connectionstyle name with optional comma-separated attributes, e.g.:

```
set_connectionstyle("arc,angleA=0,armA=30,rad=10")
```

Alternatively, the attributes can be provided as keywords, e.g.:

```
set_connectionstyle("arc", angleA=0,armA=30,rad=10)
```

Without any arguments (or with `connectionstyle=None`), return available styles as a list of strings.

`set_dpi_cor`(*self*, *dpi_cor*)

dpi_cor is currently used for linewidth-related things and shrink factor. Mutation scale is affected by this.

Parameters

dpi_cor

[float]

`set_mutation_aspect`(*self*, *aspect*)

Set the aspect ratio of the bbox mutation.

Parameters

aspect

[float]

`set_mutation_scale`(*self*, *scale*)

Set the mutation scale.

Parameters

scale

[float]

`set_patchA`(*self*, *patchA*)

Set the tail patch.

Parameters

patchA

[*patches.Patch*]

`set_patchB`(*self*, *patchB*)

Set the head patch.

Parameters

patchB

[*patches.Patch*]

set_positions(*self*, *posA*, *posB*)

Set the begin and end positions of the connecting path.

Parameters

posA, posB

[None, tuple] (x, y) coordinates of arrow tail and arrow head respectively. If None use current value.

Examples using matplotlib.patches.FancyArrowPatch

- sphx_glr_gallery_pie_and_polar_charts_bar_of_pie.py
- sphx_glr_gallery_shapes_and_collections_arrow_guide.py
- sphx_glr_gallery_userdemo_connect_simple01.py
- sphx_glr_gallery_userdemo_connectionstyle_demo.py

matplotlib.patches.FancyBboxPatch

class matplotlib.patches.**FancyBboxPatch**(*xy*, *width*, *height*, *boxstyle='round'*, *bbox_transmuter=None*, *mutation_scale=1.0*, *mutation_aspect=None*, ***kwargs*)

Bases: *matplotlib.patches.Patch*

A fancy box around a rectangle with lower left at *xy* = (*x*, *y*) with specified width and height.

FancyBboxPatch is similar to *Rectangle*, but it draws a fancy box around the rectangle. The transformation of the rectangle box to the fancy box is delegated to the style classes defined in *BoxStyle*.

Parameters

xy

[float, float] The lower left corner of the box.

width

[float] The width of the box.

height

[float] The height of the box.

boxstyle

[str or *matplotlib.patches.BoxStyle*]

The style of the fancy box. This can either be a *BoxStyle* instance or a string of the style name and optionally comma seprarated attributes (e.g. "Round, pad=0.2"). This string is passed to *BoxStyle* to construct a *BoxStyle* object. See there for a full documentation.

The following box styles are available:

Class	Name	Attrs
Circle	`circle`	pad=0.3
DArrow	`darrow`	pad=0.3
LArrow	`larrow`	pad=0.3
RArrow	`rarrow`	pad=0.3
Round	`round`	pad=0.3, rounding_size=None
Round4	`round4`	pad=0.3, rounding_size=None
Roundtooth	`roundtooth`	pad=0.3, tooth_size=None
Sawtooth	`sawtooth`	pad=0.3, tooth_size=None
Square	`square`	pad=0.3

mutation_scale

[float, default: 1] Scaling factor applied to the attributes of the box style (e.g. pad or rounding_size).

mutation_aspect

[float, optional] The height of the rectangle will be squeezed by this value before the mutation and the mutated box will be stretched by the inverse of it. For example, this allows different horizontal and vertical padding.

Other Parameters

****kwargs**

[*Patch* properties]

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and retur
alpha	float or None
animated	bool
antialiased or aa	unknown
capstyle	{'butt', 'round', 'projecting'}
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color	color
contains	unknown

Table 161 – continued from previous page

Property	Description
edgecolor or ec	color or None or 'auto'
facecolor or fc	color or None
figure	*Figure*
fill	bool
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float or None
path_effects	*AbstractPathEffect*
picker	None or bool or callable
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
visible	bool
zorder	float

`__init__`(*self*, *xy*, *width*, *height*, *boxstyle='round'*, *bbox_transmuter=None*, *mutation_scale=1.0*, *mutation_aspect=None*, ***kwargs*)

Parameters

xy

[float, float] The lower left corner of the box.

width

[float] The width of the box.

height

[float] The height of the box.

boxstyle

[str or *matplotlib.patches.BoxStyle*]

The style of the fancy box. This can either be a *BoxStyle* instance or a string of the style name and optionally comma seprarated attributes (e.g. "Round, pad=0.2"). This string is passed to *BoxStyle* to construct a *BoxStyle* object. See there for a full documentation.

The following box styles are available:

Class	Name	Attrs
Circle	`circle`	pad=0.3
DArrow	`darrow`	pad=0.3
LArrow	`larrow`	pad=0.3
RArrow	`rarrow`	pad=0.3
Round	`round`	pad=0.3, rounding_size=None
Round4	`round4`	pad=0.3, rounding_size=None
Roundtooth	`roundtooth`	pad=0.3, tooth_size=None
Sawtooth	`sawtooth`	pad=0.3, tooth_size=None
Square	`square`	pad=0.3

mutation_scale

[float, default: 1] Scaling factor applied to the attributes of the box style (e.g. pad or rounding_size).

mutation_aspect

[float, optional] The height of the rectangle will be squeezed by this value before the mutation and the mutated box will be stretched by the inverse of it. For example, this allows different horizontal and vertical padding.

Other Parameters

****kwargs**

[*Patch* properties]

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and retur
alpha	float or None
animated	bool
antialiased or aa	unknown
capstyle	{'butt', 'round', 'projecting'}
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color	color
contains	unknown
edgecolor or ec	color or None or 'auto'
facecolor or fc	color or None
figure	*Figure*
fill	bool
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool

Table 162 – continued from previous page

Property	Description
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float or None
path_effects	*AbstractPathEffect*
picker	None or bool or callable
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
visible	bool
zorder	float

`__module__ = 'matplotlib.patches'`

`__str__`(*self*)
Return str(self).

`get_bbox`(*self*)
Return the *Bbox*.

`get_boxstyle`(*self*)
Return the boxstyle object.

`get_height`(*self*)
Return the height of the rectangle.

`get_mutation_aspect`(*self*)
Return the aspect ratio of the bbox mutation.

`get_mutation_scale`(*self*)
Return the mutation scale.

`get_path`(*self*)
Return the mutated path of the rectangle.

`get_width`(*self*)
Return the width of the rectangle.

`get_x`(*self*)
Return the left coord of the rectangle.

`get_y`(*self*)
Return the bottom coord of the rectangle.

`set_bounds`(*self*, **args*)
Set the bounds of the rectangle.

Call signatures:

```
set_bounds(left, bottom, width, height)
set_bounds((left, bottom, width, height))
```

Parameters

left, bottom

[float] The coordinates of the bottom left corner of the rectangle.

width, height

[float] The width/height of the rectangle.

`set_boxstyle`(*self, boxstyle=None, **kwargs*)

Set the box style.

Most box styles can be further configured using attributes. Attributes from the previous box style are not reused.

Without argument (or with `boxstyle=None`), the available box styles are returned as a human-readable string.

Parameters

boxstyle

[str]

The name of the box style. Optionally, followed by a comma and a comma-separated list of attributes. The attributes may alternatively be passed separately as keyword arguments.

The following box styles are available:

Class	Name	Attrs
Circle	`circle`	pad=0.3
DArrow	`darrow`	pad=0.3
LArrow	`larrow`	pad=0.3
RArrow	`rarrow`	pad=0.3
Round	`round`	pad=0.3, rounding_size=None
Round4	`round4`	pad=0.3, rounding_size=None
Roundtooth	`roundtooth`	pad=0.3, tooth_size=None
Sawtooth	`sawtooth`	pad=0.3, tooth_size=None
Square	`square`	pad=0.3

****kwargs**

Additional attributes for the box style. See the table above for supported parameters.

Examples

```
set_boxstyle("round,pad=0.2")
set_boxstyle("round", pad=0.2)
```

`set_height`(*self*, *h*)
Set the rectangle height.

Parameters

h

[float]

`set_mutation_aspect`(*self*, *aspect*)
Set the aspect ratio of the bbox mutation.

Parameters

aspect

[float]

`set_mutation_scale`(*self*, *scale*)
Set the mutation scale.

Parameters

scale

[float]

`set_width`(*self*, *w*)
Set the rectangle width.

Parameters

w

[float]

`set_x`(*self*, *x*)
Set the left coord of the rectangle.

Parameters

x

[float]

set_y(*self*, *y*)

Set the bottom coord of the rectangle.

Parameters

y

[float]

Examples using matplotlib.patches.FancyBboxPatch

- sphx_glr_gallery_shapes_and_collections_artist_reference.py
- sphx_glr_gallery_shapes_and_collections_fancybox_demo.py
- sphx_glr_gallery_userdemo_annotate_text_arrow.py

matplotlib.patches.Patch

class matplotlib.patches.Patch(*edgecolor=None*, *facecolor=None*, *color=None*, *linewidth=None*, *linestyle=None*, *antialiased=None*, *hatch=None*, *fill=True*, *capstyle=None*, *joinstyle=None*, ***kwargs*)

Bases: *matplotlib.artist.Artist*

A patch is a 2D artist with a face color and an edge color.

If any of *edgecolor*, *facecolor*, *linewidth*, or *antialiased* are *None*, they default to their rc params setting.

The following kwarg properties are supported

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and retur
alpha	float or None
animated	bool
antialiased or aa	unknown
capstyle	{'butt', 'round', 'projecting'}
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color	color
contains	unknown
edgecolor or ec	color or None or 'auto'
facecolor or fc	color or None
figure	*Figure*
fill	bool

Table 163 – continued from previous page

Property	Description
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float or None
path_effects	*AbstractPathEffect*
picker	None or bool or callable
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
visible	bool
zorder	float

`__init__`(*self*, *edgecolor=None*, *facecolor=None*, *color=None*, *linewidth=None*, *linestyle=None*, *antialiased=None*, *hatch=None*, *fill=True*, *capstyle=None*, *joinstyle=None*, ***kwargs*)

The following kwarg properties are supported

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and retur
alpha	float or None
animated	bool
antialiased or aa	unknown
capstyle	{'butt', 'round', 'projecting'}
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color	color
contains	unknown
edgecolor or ec	color or None or 'auto'
facecolor or fc	color or None
figure	*Figure*
fill	bool
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object

Table 164 – continued from previous page

Property	Description
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float or None
path_effects	*AbstractPathEffect*
picker	None or bool or callable
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
visible	bool
zorder	float

`__module__ = 'matplotlib.patches'`

`contains`(*self*, *mouseevent*, *radius=None*)

Test whether the mouse event occurred in the patch.

Returns

(bool, empty dict)

`contains_point`(*self*, *point*, *radius=None*)

Return whether the given point is inside the patch.

Parameters

point

[(float, float)] The point (x, y) to check, in target coordinates of `self.get_transform()`. These are display coordinates for patches that are added to a figure or axes.

radius

[float, optional] Add an additional margin on the patch in target coordinates of `self.get_transform()`. See *Path.contains_point* for further details.

Returns

bool

Notes

The proper use of this method depends on the transform of the patch. Isolated patches do not have a transform. In this case, the patch creation coordinates and the point coordinates match. The following example checks that the center of a circle is within the circle

```
>>> center = 0, 0
>>> c = Circle(center, radius=1)
>>> c.contains_point(center)
True
```

The convention of checking against the transformed patch stems from the fact that this method is predominantly used to check if display coordinates (e.g. from mouse events) are within the patch. If you want to do the above check with data coordinates, you have to properly transform them first:

```
>>> center = 0, 0
>>> c = Circle(center, radius=1)
>>> plt.gca().add_patch(c)
>>> transformed_center = c.get_transform().transform(center)
>>> c.contains_point(transformed_center)
True
```

`contains_points`(*self*, *points*, *radius=None*)

Return whether the given points are inside the patch.

Parameters

points

[(N, 2) array] The points to check, in target coordinates of `self.get_transform()`. These are display coordinates for patches that are added to a figure or axes. Columns contain x and y values.

radius

[float, optional] Add an additional margin on the patch in target coordinates of `self.get_transform()`. See *Path.contains_point* for further details.

Returns

length-N bool array

Notes

The proper use of this method depends on the transform of the patch. See the notes on *Patch.contains_point*.

draw(*self*, *renderer*)
Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

Parameters

renderer

[*RendererBase* subclass.]

Notes

This method is overridden in the Artist subclasses.

property fill
Return whether the patch is filled.

get_aa(*self*)
Alias for *get_antialiased*.

get_antialiased(*self*)
Return whether antialiasing is used for drawing.

get_capstyle(*self*)
Return the capstyle.

get_data_transform(*self*)
Return the *Transform* mapping data coordinates to physical coordinates.

get_ec(*self*)
Alias for *get_edgecolor*.

get_edgecolor(*self*)
Return the edge color.

get_extents(*self*)
Return the *Patch*'s axis-aligned extents as a *Bbox*.

get_facecolor(*self*)
Return the face color.

get_fc(*self*)
Alias for *get_facecolor*.

get_fill(*self*)
Return whether the patch is filled.

`get_hatch`(*self*)
Return the hatching pattern.

`get_joinstyle`(*self*)
Return the joinstyle.

`get_linestyle`(*self*)
Return the linestyle.

`get_linewidth`(*self*)
Return the line width in points.

`get_ls`(*self*)
Alias for *get_linestyle*.

`get_lw`(*self*)
Alias for *get_linewidth*.

`get_patch_transform`(*self*)
Return the *Transform* instance mapping patch coordinates to data coordinates.

For example, one may define a patch of a circle which represents a radius of 5 by providing coordinates for a unit circle, and a transform which scales the coordinates (the patch coordinate) by 5.

`get_path`(*self*)
Return the path of this patch.

`get_transform`(*self*)
Return the *Transform* applied to the *Patch*.

`get_verts`(*self*)
Return a copy of the vertices used in this patch.

If the patch contains Bezier curves, the curves will be interpolated by line segments. To access the curves as curves, use *get_path*.

`get_window_extent`(*self*, *renderer=None*)
Get the axes bounding box in display space.

The bounding box' width and height are nonnegative.

Subclasses should override for inclusion in the bounding box "tight" calculation. Default is to return an empty bounding box at 0, 0.

Be careful when using this function, the results will not update if the artist window extent of the artist changes. The extent can change due to any changes in the transform stack, such as changing the axes limits, the figure size, or the canvas used (as is done when saving a figure). This can lead to unexpected behavior where interactive figures will look fine on the screen, but will save incorrectly.

`set_aa`(*self*, *aa*)
Alias for *set_antialiased*.

set_alpha(*self*, *alpha*)
Set the alpha value used for blending - not supported on all backends.

Parameters

alpha

[float or None]

set_antialiased(*self*, *aa*)
Set whether to use antialiased rendering.

Parameters

b

[bool or None]

set_capstyle(*self*, *s*)
Set the capstyle.

Parameters

s

[{'butt', 'round', 'projecting'}]

set_color(*self*, *c*)
Set both the edgecolor and the facecolor.

Parameters

c

[color]

See also:

Patch.set_facecolor, *Patch.set_edgecolor*

For setting the edge or face color individually.

set_ec(*self*, *color*)
Alias for *set_edgecolor*.

set_edgecolor(*self*, *color*)
Set the patch edge color.

Parameters

color

[color or None or 'auto']

set_facecolor(*self*, *color*)
Set the patch face color.

Parameters

color

[color or None]

set_fc(*self*, *color*)
Alias for *set_facecolor*.

set_fill(*self*, *b*)
Set whether to fill the patch.

Parameters

b

[bool]

set_hatch(*self*, *hatch*)
Set the hatching pattern.

hatch can be one of:

```
/   - diagonal hatching
\   - back diagonal
|   - vertical
-   - horizontal
+   - crossed
x   - crossed diagonal
o   - small circle
O   - large circle
.   - dots
*   - stars
```

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Parameters

hatch

[{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}]

set_joinstyle(*self*, *s*)
Set the joinstyle.

Parameters

s

[{'miter', 'round', 'bevel'}]

`set_linestyle`(*self*, *ls*)
Set the patch linestyle.

linestyle	description
`'-'` or `'solid'`	solid line
`'--'` or `'dashed'`	dashed line
`'-.'` or `'dashdot'`	dash-dotted line
`':'` or `'dotted'`	dotted line

Alternatively a dash tuple of the following form can be provided:

```
(offset, onoffseq)
```

where `onoffseq` is an even length tuple of on and off ink in points.

Parameters

ls

[{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}] The line style.

`set_linewidth`(*self*, *w*)
Set the patch linewidth in points.

Parameters

w

[float or None]

`set_ls`(*self*, *ls*)
Alias for *set_linestyle*.

`set_lw`(*self*, *w*)
Alias for *set_linewidth*.

`update_from`(*self*, *other*)
Copy properties from *other* to *self*.

```
validCap = ('butt', 'round', 'projecting')
```

```
validJoin = ('miter', 'round', 'bevel')
```

```
zorder = 1
```

Examples using `matplotlib.patches.Patch`

- sphx_glr_gallery_lines_bars_and_markers_curve_error_band.py
- sphx_glr_gallery_images_contours_and_fields_image_clip_path.py
- sphx_glr_gallery_images_contours_and_fields_image_demo.py
- sphx_glr_gallery_subplots_axes_and_figures_axes_box_aspect.py
- sphx_glr_gallery_subplots_axes_and_figures_axes_margins.py
- sphx_glr_gallery_subplots_axes_and_figures_axes_zoom_effect.py
- sphx_glr_gallery_statistics_boxplot_demo.py
- sphx_glr_gallery_statistics_confidence_ellipse.py
- sphx_glr_gallery_statistics_errorbars_and_boxes.py
- sphx_glr_gallery_pie_and_polar_charts_bar_of_pie.py
- sphx_glr_gallery_text_labels_and_annotations_annotation_demo.py
- sphx_glr_gallery_text_labels_and_annotations_custom_legends.py
- sphx_glr_gallery_text_labels_and_annotations_demo_annotation_box.py
- sphx_glr_gallery_text_labels_and_annotations_demo_text_path.py
- sphx_glr_gallery_text_labels_and_annotations_demo_text_rotation_mode.py
- sphx_glr_gallery_text_labels_and_annotations_fancyarrow_demo.py
- sphx_glr_gallery_text_labels_and_annotations_text_alignment.py
- sphx_glr_gallery_pyplots_text_layout.py
- sphx_glr_gallery_pyplots_whats_new_99_spines.py
- sphx_glr_gallery_color_named_colors.py
- sphx_glr_gallery_shapes_and_collections_arrow_guide.py
- sphx_glr_gallery_shapes_and_collections_artist_reference.py
- sphx_glr_gallery_shapes_and_collections_compound_path.py
- sphx_glr_gallery_shapes_and_collections_dolphin.py
- sphx_glr_gallery_shapes_and_collections_donut.py
- sphx_glr_gallery_shapes_and_collections_ellipse_demo.py
- sphx_glr_gallery_shapes_and_collections_fancybox_demo.py
- sphx_glr_gallery_shapes_and_collections_hatch_demo.py
- sphx_glr_gallery_shapes_and_collections_patch_collection.py
- sphx_glr_gallery_shapes_and_collections_path_patch.py

- sphx_glr_gallery_shapes_and_collections_quad_bezier.py
- sphx_glr_gallery_style_sheets_ggplot.py
- sphx_glr_gallery_style_sheets_grayscale.py
- sphx_glr_gallery_style_sheets_style_sheets_reference.py
- sphx_glr_gallery_axes_grid1_inset_locator_demo.py
- sphx_glr_gallery_showcase_anatomy.py
- sphx_glr_gallery_showcase_firefox.py
- sphx_glr_gallery_showcase_integral.py
- *Animated histogram*
- sphx_glr_gallery_event_handling_looking_glass.py
- sphx_glr_gallery_event_handling_path_editor.py
- sphx_glr_gallery_event_handling_pick_event_demo.py
- sphx_glr_gallery_event_handling_poly_editor.py
- sphx_glr_gallery_event_handling_trifinder_event_demo.py
- sphx_glr_gallery_event_handling_viewlims.py
- sphx_glr_gallery_misc_anchored_artists.py
- sphx_glr_gallery_misc_bbox_intersect.py
- sphx_glr_gallery_misc_custom_projection.py
- sphx_glr_gallery_misc_histogram_path.py
- sphx_glr_gallery_misc_logos2.py
- sphx_glr_gallery_misc_svg_filter_pie.py
- sphx_glr_gallery_mplot3d_pathpatch3d.py
- sphx_glr_gallery_specialty_plots_hinton_demo.py
- sphx_glr_gallery_specialty_plots_radar_chart.py
- sphx_glr_gallery_specialty_plots_skewt.py
- sphx_glr_gallery_units_artist_tests.py
- sphx_glr_gallery_units_ellipse_with_units.py
- sphx_glr_gallery_userdemo_anchored_box02.py
- sphx_glr_gallery_userdemo_anchored_box03.py
- sphx_glr_gallery_userdemo_anchored_box04.py
- sphx_glr_gallery_userdemo_annotate_explain.py
- sphx_glr_gallery_userdemo_connect_simple01.py

- sphx_glr_gallery_userdemo_connectionstyle_demo.py
- sphx_glr_gallery_userdemo_simple_annotate01.py
- sphx_glr_gallery_widgets_menu.py
- *Artist tutorial*
- *Legend guide*
- *Path Tutorial*
- *Transformations Tutorial*
- *Specifying Colors*
- *Text properties and layout*

matplotlib.patches.PathPatch

class matplotlib.patches.**PathPatch**(*path*, ***kwargs*)

Bases: *matplotlib.patches.Patch*

A general polycurve path patch.

path is a *Path* object.

Valid keyword arguments are:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and retur
alpha	float or None
animated	bool
antialiased or aa	unknown
capstyle	{'butt', 'round', 'projecting'}
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color	color
contains	unknown
edgecolor or ec	color or None or 'auto'
facecolor or fc	color or None
figure	*Figure*
fill	bool
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}

Table 165 – continued from previous page

Property	Description
linewidth or lw	float or None
path_effects	*AbstractPathEffect*
picker	None or bool or callable
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
visible	bool
zorder	float

`__init__`(*self*, *path*, ***kwargs*)

path is a *Path* object.

Valid keyword arguments are:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and retur
alpha	float or None
animated	bool
antialiased or aa	unknown
capstyle	{'butt', 'round', 'projecting'}
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color	color
contains	unknown
edgecolor or ec	color or None or 'auto'
facecolor or fc	color or None
figure	*Figure*
fill	bool
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float or None
path_effects	*AbstractPathEffect*
picker	None or bool or callable
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)

Table 166 – continued from previous page

Property	Description
snap	bool or None
transform	*Transform*
url	str
visible	bool
zorder	float

`__module__ = 'matplotlib.patches'`

`__str__`(*self*)
Return str(self).

`get_path`(*self*)
Return the path of this patch.

`set_path`(*self*, *path*)

Examples using `matplotlib.patches.PathPatch`

- sphx_glr_gallery_lines_bars_and_markers_curve_error_band.py
- sphx_glr_gallery_images_contours_and_fields_image_demo.py
- sphx_glr_gallery_statistics_boxplot_color.py
- sphx_glr_gallery_text_labels_and_annotations_demo_text_path.py
- sphx_glr_gallery_shapes_and_collections_artist_reference.py
- sphx_glr_gallery_shapes_and_collections_compound_path.py
- sphx_glr_gallery_shapes_and_collections_dolphin.py
- sphx_glr_gallery_shapes_and_collections_donut.py
- sphx_glr_gallery_shapes_and_collections_path_patch.py
- sphx_glr_gallery_shapes_and_collections_quad_bezier.py
- sphx_glr_gallery_showcase_firefox.py
- *Animated histogram*
- sphx_glr_gallery_event_handling_path_editor.py
- sphx_glr_gallery_misc_histogram_path.py
- sphx_glr_gallery_misc_logos2.py
- sphx_glr_gallery_mplot3d_pathpatch3d.py
- *Path Tutorial*

matplotlib.patches.Polygon

class matplotlib.patches.Polygon(*xy*, *closed=True*, ***kwargs*)

Bases: *matplotlib.patches.Patch*

A general polygon patch.

xy is a numpy array with shape Nx2.

If *closed* is *True*, the polygon will be closed so the starting and ending points are the same.

Valid keyword arguments are:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and retur
alpha	float or None
animated	bool
antialiased or aa	unknown
capstyle	{'butt', 'round', 'projecting'}
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color	color
contains	unknown
edgecolor or ec	color or None or 'auto'
facecolor or fc	color or None
figure	*Figure*
fill	bool
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float or None
path_effects	*AbstractPathEffect*
picker	None or bool or callable
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
visible	bool
zorder	float

__init__(*self*, *xy*, *closed=True*, ***kwargs*)
xy is a numpy array with shape Nx2.

If *closed* is *True*, the polygon will be closed so the starting and ending points are the same.

Valid keyword arguments are:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and retur
alpha	float or None
animated	bool
antialiased or aa	unknown
capstyle	{'butt', 'round', 'projecting'}
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color	color
contains	unknown
edgecolor or ec	color or None or 'auto'
facecolor or fc	color or None
figure	*Figure*
fill	bool
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float or None
path_effects	*AbstractPathEffect*
picker	None or bool or callable
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
visible	bool
zorder	float

__module__ = 'matplotlib.patches'

__str__(*self*)
Return str(self).

get_closed(*self*)
Return whether the polygon is closed.

`get_path`(*self*)
Get the *Path* of the polygon.

`get_xy`(*self*)
Get the vertices of the path.

Returns

(N, 2) numpy array

The coordinates of the vertices.

`set_closed`(*self*, *closed*)
Set whether the polygon is closed.

Parameters

closed

[bool] True if the polygon is closed

`set_xy`(*self*, *xy*)
Set the vertices of the polygon.

Parameters

xy

[(N, 2) array-like] The coordinates of the vertices.

Notes

Unlike *Path*, we do not ignore the last input vertex. If the polygon is meant to be closed, and the last point of the polygon is not equal to the first, we assume that the user has not explicitly passed a `CLOSEPOLY` vertex, and add it ourselves.

`property xy`
The vertices of the path as (N, 2) numpy array.

Examples using `matplotlib.patches.Polygon`

- sphx_glr_gallery_subplots_axes_and_figures_axes_margins.py
- sphx_glr_gallery_statistics_boxplot_demo.py
- sphx_glr_gallery_shapes_and_collections_hatch_demo.py
- sphx_glr_gallery_shapes_and_collections_patch_collection.py
- sphx_glr_gallery_axisartist_demo_floating_axes.py

- sphx_glr_gallery_showcase_integral.py
- sphx_glr_gallery_event_handling_poly_editor.py
- sphx_glr_gallery_event_handling_trifinder_event_demo.py
- *Annotations*

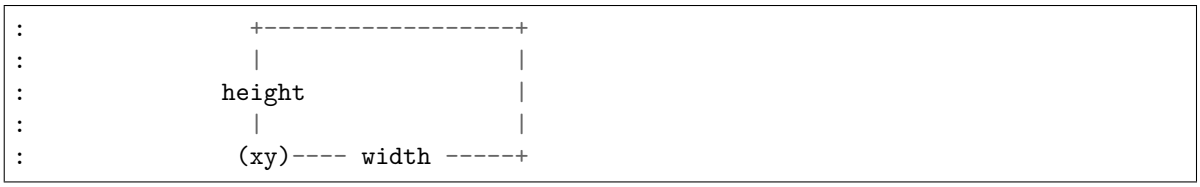
matplotlib.patches.Rectangle

class matplotlib.patches.**Rectangle**(*xy*, *width*, *height*, *angle=0.0*, ***kwargs*)

Bases: *matplotlib.patches.Patch*

A rectangle defined via an anchor point *xy* and its *width* and *height*.

The rectangle extends from `xy[0]` to `xy[0] + width` in x-direction and from `xy[1]` to `xy[1] + height` in y-direction.



One may picture *xy* as the bottom left corner, but which corner *xy* is actually depends on the the direction of the axis and the sign of *width* and *height*; e.g. *xy* would be the bottom right corner if the x-axis was inverted or if *width* was negative.

Parameters

xy

[(float, float)] The anchor point.

width

[float] Rectangle width.

height

[float] Rectangle height.

angle

[float, default: 0] Rotation in degrees anti-clockwise about *xy*.

Other Parameters

****kwargs**

[*Patch* properties]

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and retur
alpha	float or None
animated	bool
antialiased or aa	unknown
capstyle	{'butt', 'round', 'projecting'}
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color	color
contains	unknown
edgecolor or ec	color or None or 'auto'
facecolor or fc	color or None
figure	*Figure*
fill	bool
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float or None
path_effects	*AbstractPathEffect*
picker	None or bool or callable
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
visible	bool
zorder	float

`__init__`(*self*, *xy*, *width*, *height*, *angle=0.0*, ***kwargs*)

Parameters

xy

[(float, float)] The anchor point.

width

[float] Rectangle width.

height

[float] Rectangle height.

angle

[float, default: 0] Rotation in degrees anti-clockwise about *xy*.

Other Parameters

****kwargs**

[*Patch* properties]

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and retur
alpha	float or None
animated	bool
antialiased or aa	unknown
capstyle	{'butt', 'round', 'projecting'}
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color	color
contains	unknown
edgecolor or ec	color or None or 'auto'
facecolor or fc	color or None
figure	*Figure*
fill	bool
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float or None
path_effects	*AbstractPathEffect*
picker	None or bool or callable
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
visible	bool
zorder	float

`__module__ = 'matplotlib.patches'`

`__str__`(*self*)
Return str(self).

`get_bbox`(*self*)
Return the *Bbox*.

`get_height`(*self*)
Return the height of the rectangle.

`get_patch_transform`(*self*)
Return the *Transform* instance mapping patch coordinates to data coordinates.

For example, one may define a patch of a circle which represents a radius of 5 by providing coordinates for a unit circle, and a transform which scales the coordinates (the patch coordinate) by 5.

`get_path`(*self*)
Return the vertices of the rectangle.

`get_width`(*self*)
Return the width of the rectangle.

`get_x`(*self*)
Return the left coordinate of the rectangle.

`get_xy`(*self*)
Return the left and bottom coords of the rectangle as a tuple.

`get_y`(*self*)
Return the bottom coordinate of the rectangle.

`set_bounds`(*self*, **args*)
Set the bounds of the rectangle as *left*, *bottom*, *width*, *height*.

The values may be passed as separate parameters or as a tuple:

```
set_bounds(left, bottom, width, height)
set_bounds((left, bottom, width, height))
```

`set_height`(*self*, *h*)
Set the height of the rectangle.

`set_width`(*self*, *w*)
Set the width of the rectangle.

`set_x`(*self*, *x*)
Set the left coordinate of the rectangle.

`set_xy`(*self*, *xy*)
Set the left and bottom coordinates of the rectangle.

Parameters

xy

[(float, float)]

`set_y`(*self*, *y*)
Set the bottom coordinate of the rectangle.

property xy
: Return the left and bottom coords of the rectangle as a tuple.

Examples using `matplotlib.patches.Rectangle`

- sphx_glr_gallery_statistics_errorbars_and_boxes.py
- sphx_glr_gallery_statistics_hist.py
- sphx_glr_gallery_text_labels_and_annotations_demo_text_rotation_mode.py
- sphx_glr_gallery_text_labels_and_annotations_text_alignment.py
- sphx_glr_gallery_pyplots_fig_axes_customize_simple.py
- sphx_glr_gallery_pyplots_text_layout.py
- sphx_glr_gallery_color_named_colors.py
- sphx_glr_gallery_shapes_and_collections_artist_reference.py
- sphx_glr_gallery_shapes_and_collections_hatch_demo.py
- sphx_glr_gallery_axes_grid1_inset_locator_demo.py
- sphx_glr_gallery_event_handling_pick_event_demo.py
- sphx_glr_gallery_event_handling_viewlims.py
- sphx_glr_gallery_misc_bbox_intersect.py
- sphx_glr_gallery_misc_logos2.py
- sphx_glr_gallery_specialty_plots_hinton_demo.py
- sphx_glr_gallery_units_artist_tests.py
- sphx_glr_gallery_widgets_menu.py
- *Artist tutorial*
- *Legend guide*
- *Transformations Tutorial*
- *Specifying Colors*
- *Text properties and layout*

matplotlib.patches.RegularPolygon

class matplotlib.patches.RegularPolygon(*xy*, *numVertices*, *radius=5*, *orientation=0*, ***kwargs*)

Bases: *matplotlib.patches.Patch*

A regular polygon patch.

Parameters

xy

[(float, float)] The center position.

numVertices

[int] The number of vertices.

radius

[float] The distance from the center to each of the vertices.

orientation

[float] The polygon rotation angle (in radians).

****kwargs**

Patch properties:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and retur
alpha	float or None
animated	bool
antialiased or aa	unknown
capstyle	{'butt', 'round', 'projecting'}
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color	color
contains	unknown
edgecolor or ec	color or None or 'auto'
facecolor or fc	color or None
figure	*Figure*
fill	bool
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}

Table 171 – continued from previous page

Property	Description
linewidth or lw	float or None
path_effects	*AbstractPathEffect*
picker	None or bool or callable
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
visible	bool
zorder	float

`__init__`(*self*, *xy*, *numVertices*, *radius=5*, *orientation=0*, ***kwargs*)

Parameters

xy

[(float, float)] The center position.

numVertices

[int] The number of vertices.

radius

[float] The distance from the center to each of the vertices.

orientation

[float] The polygon rotation angle (in radians).

****kwargs**

Patch properties:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and retur
alpha	float or None
animated	bool
antialiased or aa	unknown
capstyle	{'butt', 'round', 'projecting'}
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color	color
contains	unknown
edgecolor or ec	color or None or 'auto'
facecolor or fc	color or None

Table 172 – continued from previous page

Property	Description
figure	*Figure*
fill	bool
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float or None
path_effects	*AbstractPathEffect*
picker	None or bool or callable
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
visible	bool
zorder	float

__module__ = 'matplotlib.patches'

__str__(*self*)
Return str(self).

get_patch_transform(*self*)
Return the *Transform* instance mapping patch coordinates to data coordinates.

For example, one may define a patch of a circle which represents a radius of 5 by providing coordinates for a unit circle, and a transform which scales the coordinates (the patch coordinate) by 5.

get_path(*self*)
Return the path of this patch.

property numvertices

property orientation

property radius

property xy

Examples using `matplotlib.patches.RegularPolygon`

- sphx_glr_gallery_shapes_and_collections_artist_reference.py
- sphx_glr_gallery_specialty_plots_radar_chart.py

matplotlib.patches.Shadow

class matplotlib.patches.**Shadow**(*patch*, *ox*, *oy*, *props=<deprecated parameter>*, ***kwargs*)

Bases: *matplotlib.patches.Patch*

Create a shadow of the given *patch*.

By default, the shadow will have the same face color as the *patch*, but darkened.

Parameters

patch

[*Patch*] The patch to create the shadow for.

ox, oy

[float] The shift of the shadow in data coordinates, scaled by a factor of dpi/72.

props

[dict] *deprecated (use kwargs instead)* Properties of the shadow patch.

****kwargs**

Properties of the shadow patch. Supported keys are:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and retur
alpha	float or None
animated	bool
antialiased or aa	unknown
capstyle	{'butt', 'round', 'projecting'}
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color	color
contains	unknown
edgecolor or ec	color or None or 'auto'
facecolor or fc	color or None
figure	*Figure*

Table 173 – continued from previous page

Property	Description
fill	bool
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float or None
path_effects	*AbstractPathEffect*
picker	None or bool or callable
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
visible	bool
zorder	float

`__init__`(*self*, *patch*, *ox*, *oy*, *props=<deprecated parameter>*, ***kwargs*)

Create a shadow of the given *patch*.

By default, the shadow will have the same face color as the *patch*, but darkened.

Parameters

patch

[*Patch*] The patch to create the shadow for.

ox, oy

[float] The shift of the shadow in data coordinates, scaled by a factor of dpi/72.

props

[dict] *deprecated (use kwargs instead)* Properties of the shadow patch.

****kwargs**

Properties of the shadow patch. Supported keys are:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and retur
alpha	float or None
animated	bool

Table 174 – continued from previous page

Property	Description
antialiased or aa	unknown
capstyle	{'butt', 'round', 'projecting'}
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color	color
contains	unknown
edgecolor or ec	color or None or 'auto'
facecolor or fc	color or None
figure	*Figure*
fill	bool
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float or None
path_effects	*AbstractPathEffect*
picker	None or bool or callable
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
visible	bool
zorder	float

__module__ = 'matplotlib.patches'

__str__(*self*)
Return str(self).

draw(*self*, *renderer*)
Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

Parameters

renderer

[*RendererBase* subclass.]

Notes

This method is overridden in the Artist subclasses.

`get_patch_transform`(*self*)
Return the *Transform* instance mapping patch coordinates to data coordinates.

For example, one may define a patch of a circle which represents a radius of 5 by providing coordinates for a unit circle, and a transform which scales the coordinates (the patch coordinate) by 5.

`get_path`(*self*)
Return the path of this patch.

`property props`

Examples using `matplotlib.patches.Shadow`

- sphx_glr_gallery_text_labels_and_annotations_demo_text_path.py
- sphx_glr_gallery_misc_svg_filter_pie.py

matplotlib.patches.Wedge

`class matplotlib.patches.Wedge`(*center*, *r*, *theta1*, *theta2*, *width=None*, ***kwargs*)
Bases: *matplotlib.patches.Patch*

Wedge shaped patch.

A wedge centered at *x*, *y* center with radius *r* that sweeps *theta1* to *theta2* (in degrees). If *width* is given, then a partial wedge is drawn from inner radius *r* - *width* to outer radius *r*.

Valid keyword arguments are:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and retur
alpha	float or None
animated	bool
antialiased or aa	unknown
capstyle	{'butt', 'round', 'projecting'}
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color	color
contains	unknown

Table 175 – continued from previous page

Property	Description
edgecolor or ec	color or None or 'auto'
facecolor or fc	color or None
figure	*Figure*
fill	bool
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float or None
path_effects	*AbstractPathEffect*
picker	None or bool or callable
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
visible	bool
zorder	float

__init__(*self*, *center*, *r*, *theta1*, *theta2*, *width=None*, ***kwargs*)

A wedge centered at *x*, *y* center with radius *r* that sweeps *theta1* to *theta2* (in degrees). If *width* is given, then a partial wedge is drawn from inner radius *r* - *width* to outer radius *r*.

Valid keyword arguments are:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and retur
alpha	float or None
animated	bool
antialiased or aa	unknown
capstyle	{'butt', 'round', 'projecting'}
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color	color
contains	unknown
edgecolor or ec	color or None or 'auto'
facecolor or fc	color or None
figure	*Figure*

Table 176 – continued from previous page

Property	Description
fill	bool
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float or None
path_effects	*AbstractPathEffect*
picker	None or bool or callable
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
visible	bool
zorder	float

__module__ = 'matplotlib.patches'

__str__(*self*)
Return str(self).

get_path(*self*)
Return the path of this patch.

set_center(*self*, *center*)

set_radius(*self*, *radius*)

set_theta1(*self*, *theta1*)

set_theta2(*self*, *theta2*)

set_width(*self*, *width*)

Examples using `matplotlib.patches.Wedge`

- sphx_glr_gallery_pie_and_polar_charts_pie_and_donut_labels.py
- sphx_glr_gallery_shapes_and_collections_artist_reference.py
- sphx_glr_gallery_shapes_and_collections_patch_collection.py
- sphx_glr_gallery_misc_svg_filter_pie.py

18.35.2 Functions

bbox_artist(artist, renderer[, props, fill])	A debug function to draw a rectangle around the bounding box returned by an artist's *Artist.get_window_extent* to test whether the artist is returning the correct bbox.
draw_bbox(bbox, renderer[, color, trans])	A debug function to draw a rectangle around the bounding box returned by an artist's *Artist.get_window_extent* to test whether the artist is returning the correct bbox.

matplotlib.patches.bbox_artist

matplotlib.patches.**bbox_artist**(*artist*, *renderer*, *props=None*, *fill=True*)

A debug function to draw a rectangle around the bounding box returned by an artist's *Artist.get_window_extent* to test whether the artist is returning the correct bbox.

props is a dict of rectangle props with the additional property 'pad' that sets the padding around the bbox in points.

Examples using `matplotlib.patches.bbox_artist`

matplotlib.patches.draw_bbox

matplotlib.patches.**draw_bbox**(*bbox*, *renderer*, *color='k'*, *trans=None*)

A debug function to draw a rectangle around the bounding box returned by an artist's *Artist.get_window_extent* to test whether the artist is returning the correct bbox.

Examples using `matplotlib.patches.draw_bbox`

18.36 matplotlib.path

A module for dealing with the polylines used throughout Matplotlib.

The primary class for polyline handling in Matplotlib is *Path*. Almost all vector drawing makes use of *Path*s somewhere in the drawing pipeline.

Whilst a *Path* instance itself cannot be drawn, some *Artist* subclasses, such as *PathPatch* and *PathCollection*, can be used for convenient *Path* visualisation.

class matplotlib.path.Path(*vertices*, *codes=None*, *_interpolation_steps=1*, *closed=False*, *readonly=False*)

Bases: object

A series of possibly disconnected, possibly closed, line and curve segments.

The underlying storage is made up of two parallel numpy arrays:

- *vertices*: an Nx2 float array of vertices
- *codes*: an N-length uint8 array of vertex types, or None

These two arrays always have the same length in the first dimension. For example, to represent a cubic curve, you must provide three vertices as well as three codes `CURVE3`.

The code types are:

- `STOP`

  [1 vertex (ignored)] A marker for the end of the entire path (currently not required and ignored)

- `MOVETO`

  [1 vertex] Pick up the pen and move to the given vertex.

- `LINETO`

  [1 vertex] Draw a line from the current position to the given vertex.

- `CURVE3`

  [1 control point, 1 endpoint] Draw a quadratic Bezier curve from the current position, with the given control point, to the given end point.

- `CURVE4`

  [2 control points, 1 endpoint] Draw a cubic Bezier curve from the current position, with the given control points, to the given end point.

- `CLOSEPOLY`

  [1 vertex (ignored)] Draw a line segment to the start point of the current polyline.

If *codes* is None, it is interpreted as a `MOVETO` followed by a series of `LINETO`.

Users of Path objects should not access the vertices and codes arrays directly. Instead, they should use *iter_segments* or *cleaned* to get the vertex/code pairs. This helps, in particular, to consistently handle the case of *codes* being None.

Some behavior of Path objects can be controlled by rcParams. See the rcParams whose keys start with 'path.'.

Note: The vertices and codes arrays should be treated as immutable -- there are a number of optimizations and assumptions made up front in the constructor

that will not change when the data changes.

Create a new path with the given vertices and codes.

Parameters

vertices

[array-like] The `(N, 2)` float array, masked array or sequence of pairs representing the vertices of the path.

If *vertices* contains masked values, they will be converted to NaNs which are then handled correctly by the Agg PathIterator and other consumers of path data, such as *iter_segments()*.

codes

[array-like or None, optional] n-length array integers representing the codes of the path. If not None, codes must be the same length as vertices. If None, *vertices* will be treated as a series of line segments.

_interpolation_steps

[int, optional] Used as a hint to certain projections, such as Polar, that this path should be linearly interpolated immediately before drawing. This attribute is primarily an implementation detail and is not intended for public use.

closed

[bool, optional] If *codes* is None and closed is True, vertices will be treated as line segments of a closed polygon. Note that the last vertex will then be ignored (as the corresponding code will be set to CLOSEPOLY).

readonly

[bool, optional] Makes the path behave in an immutable way and sets the vertices and codes as read-only arrays.

`CLOSEPOLY = 79`

`CURVE3 = 3`

`CURVE4 = 4`

`LINETO = 2`

`MOVETO = 1`

`NUM_VERTICES_FOR_CODE = {0: 1, 1: 1, 2: 1, 3: 2, 4: 3, 79: 1}`
A dictionary mapping Path codes to the number of vertices that the code expects.

`STOP = 0`

`classmethod arc(`*theta1*, *theta2*, *n=None*, *is_wedge=False*`)`
Return the unit circle arc from angles *theta1* to *theta2* (in degrees).

theta2 is unwrapped to produce the shortest arc within 360 degrees. That is, if *theta2* > *theta1* + 360, the arc will be from *theta1* to *theta2* - 360 and not a full circle plus some extra overlap.

If *n* is provided, it is the number of spline segments to make. If *n* is not provided, the number of spline segments is determined based on the delta between *theta1* and *theta2*.

Masionobe, L. 2003. Drawing an elliptical arc using polylines, quadratic or cubic Bezier curves.

`classmethod circle(`*center=0.0, 0.0*, *radius=1.0*, *readonly=False*`)`
Return a *Path* representing a circle of a given radius and center.

Parameters

center

[(float, float), default: (0, 0)] The center of the circle.

radius

[float, default: 1] The radius of the circle.

readonly

[bool] Whether the created path should have the "readonly" argument set when creating the Path instance.

Notes

The circle is approximated using 8 cubic Bezier curves, as described in

Lancaster, Don. Approximating a Circle or an Ellipse Using Four Bezier Cubic Splines.

`cleaned(`*self*, *transform=None*, *remove_nans=False*, *clip=None*, *quantize=<deprecated parameter>*, *simplify=False*, *curves=False*, *stroke_width=1.0*, *snap=False*, *sketch=None*`)`
Return a new Path with vertices and codes cleaned according to the parameters.

See also:

Path.iter_segments

for details of the keyword arguments.

`clip_to_bbox(`*self*, *bbox*, *inside=True*`)`
Clip the path to the given bounding box.

The path must be made up of one or more closed polygons. This algorithm will not behave correctly for unclosed paths.

If *inside* is `True`, clip to the inside of the box, otherwise to the outside of the box.

`code_type`
alias of `numpy.uint8`

`property codes`
The list of codes in the *Path* as a 1-D numpy array. Each code is one of *STOP*, *MOVETO*, *LINETO*, *CURVE3*, *CURVE4* or *CLOSEPOLY*. For codes that correspond to more than one vertex (*CURVE3* and *CURVE4*), that code will be repeated so that the length of `self.vertices` and `self.codes` is always the same.

`contains_path`(*self*, *path*, *transform=None*)
Return whether this (closed) path completely contains the given path.

If *transform* is not `None`, the path will be transformed before checking for containment.

`contains_point`(*self*, *point*, *transform=None*, *radius=0.0*)
Return whether the (closed) path contains the given point.

Parameters

point

[(float, float)] The point (x, y) to check.

transform

[*matplotlib.transforms.Transform*, optional] If not `None`, *point* will be compared to `self` transformed by *transform*; i.e. for a correct check, *transform* should transform the path into the coordinate system of *point*.

radius

[float, default: 0] Add an additional margin on the path in coordinates of *point*. The path is extended tangentially by *radius/2*; i.e. if you would draw the path with a linewidth of *radius*, all points on the line would still be considered to be contained in the area. Conversely, negative values shrink the area: Points on the imaginary line will be considered outside the area.

Returns

bool

`contains_points`(*self*, *points*, *transform=None*, *radius=0.0*)
Return whether the (closed) path contains the given point.

Parameters

points

[(N, 2) array] The points to check. Columns contain x and y values.

transform

[*matplotlib.transforms.Transform*, optional] If not `None`, *points* will be compared to `self` transformed by *transform*; i.e. for a correct check, *transform* should transform the path into the coordinate system of *points*.

radius

[float, default: 0.] Add an additional margin on the path in coordinates of *points*. The path is extended tangentially by *radius/2*; i.e. if you would draw the path with a linewidth of *radius*, all points on the line would still be considered to be contained in the area. Conversely, negative values shrink the area: Points on the imaginary line will be considered outside the area.

Returns

length-N bool array

`copy`(*self*)
Return a shallow copy of the *Path*, which will share the vertices and codes with the source *Path*.

`deepcopy`(*self*, *memo=None*)
Return a deepcopy of the *Path*. The *Path* will not be readonly, even if the source *Path* is.

`get_extents`(*self*, *transform=None*, ***kwargs*)
Get Bbox of the path.

Parameters

transform

[matplotlib.transforms.Transform, optional] Transform to apply to path before computing extents, if any.

****kwargs**

Forwarded to *iter_bezier*.

Returns

matplotlib.transforms.Bbox

The extents of the path Bbox([[xmin, ymin], [xmax, ymax]])

`static hatch`(*hatchpattern*, *density=6*)
Given a hatch specifier, *hatchpattern*, generates a Path that can be used in a repeated hatching pattern. *density* is the number of lines per unit square.

`interpolated`(*self*, *steps*)
Return a new path resampled to length N x steps.

Codes other than LINETO are not handled correctly.

`intersects_bbox`(*self*, *bbox*, *filled=True*)
Return whether this path intersects a given *Bbox*.

If *filled* is True, then this also returns True if the path completely encloses the *Bbox* (i.e., the path is treated as filled).

The bounding box is always considered filled.

`intersects_path`(*self*, *other*, *filled=True*)
Return whether if this path intersects another given path.

If *filled* is True, then this also returns True if one path completely encloses the other (i.e., the paths are treated as filled).

`iter_bezier`(*self*, ***kwargs*)
Iterate over each bezier curve (lines included) in a Path.

Parameters

****kwargs**

Forwarded to *iter_segments*.

Yields

B

[matplotlib.bezier.BezierSegment] The bezier curves that make up the current path. Note in particular that freestanding points are bezier curves of order 0, and lines are bezier curves of order 1 (with two control points).

code

[Path.code_type] The code describing what kind of curve is being returned. Path.MOVETO, Path.LINETO, Path.CURVE3, Path.CURVE4 correspond to bezier curves with 1, 2, 3, and 4 control points (respectively). Path.CLOSEPOLY is a Path.LINETO with the control points correctly chosen based on the start/end points of the current stroke.

`iter_segments`(*self*, *transform=None*, *remove_nans=True*, *clip=None*, *snap=False*, *stroke_width=1.0*, *simplify=None*, *curves=True*, *sketch=None*)
Iterate over all curve segments in the path.

Each iteration returns a pair `(vertices, code)`, where `vertices` is a sequence of 1-3 coordinate pairs, and `code` is a *Path* code.

Additionally, this method can provide a number of standard cleanups and conversions to the path.

Parameters

transform

[None or *Transform*] If not None, the given affine transformation will be applied to the path.

remove_nans

[bool, optional] Whether to remove all NaNs from the path and skip over them using MOVETO commands.

clip

[None or (float, float, float, float), optional] If not None, must be a four-tuple (x1, y1, x2, y2) defining a rectangle in which to clip the path.

snap

[None or bool, optional] If True, snap all nodes to pixels; if False, don't snap them. If None, snap if the path contains only segments parallel to the x or y axes, and no more than 1024 of them.

stroke_width

[float, optional] The width of the stroke being drawn (used for path snapping).

simplify

[None or bool, optional] Whether to simplify the path by removing vertices that do not affect its appearance. If None, use the *should_simplify* attribute. See also `rcParams["path.simplify"]` (default: `True`) and `rcParams["path.simplify_threshold"]` (default: `0.111111111111`).

curves

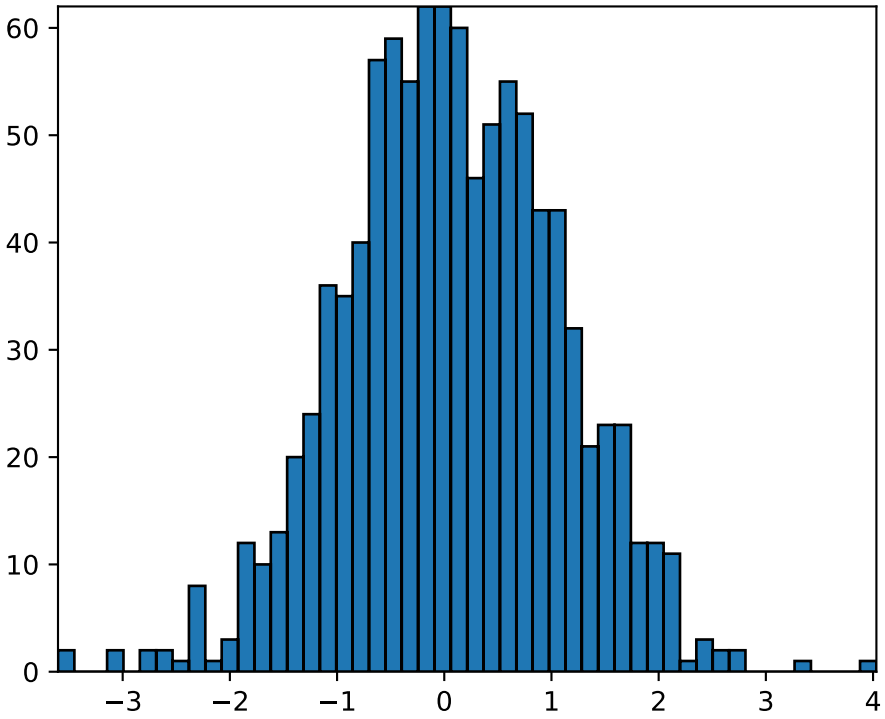
[bool, optional] If True, curve segments will be returned as curve segments. If False, all curves will be converted to line segments.

sketch

[None or sequence, optional] If not None, must be a 3-tuple of the form (scale, length, randomness), representing the sketch parameters.

classmethod **make_compound_path**(**args*)
: Make a compound path from a list of Path objects. Blindly removes all Path.STOP control points.

classmethod **make_compound_path_from_polys**(*XY*)
: Make a compound path object to draw a number of polygons with equal numbers of sides XY is a (numpolys x numsides x 2) numpy array of vertices. Return object is a *Path*



property **readonly**
: `True` if the *Path* is read-only.

property **should_simplify**
: `True` if the vertices array should be simplified.

property **simplify_threshold**
: The fraction of a pixel difference below which vertices will be simplified out.

to_polygons(*self*, *transform=None*, *width=0*, *height=0*, *closed_only=True*)
: Convert this path to a list of polygons or polylines. Each polygon/polyline is an Nx2 array of vertices. In other words, each polygon has no `MOVETO` instructions or curves. This is useful for displaying in backends that do not support compound paths or Bezier curves.

  If *width* and *height* are both non-zero then the lines will be simplified so that vertices outside of (0, 0), (width, height) will be clipped.

If *closed_only* is `True` (default), only closed polygons, with the last point being the same as the first point, will be returned. Any unclosed polylines in the path will be explicitly closed. If *closed_only* is `False`, any unclosed polygons in the path will be returned as unclosed polygons, and the closed polygons will be returned explicitly closed by setting the last point to the same as the first point.

`transformed`(*self*, *transform*)
Return a transformed copy of the path.

See also:

matplotlib.transforms.TransformedPath

A specialized path class that will cache the transformed result and automatically update when the transform changes.

`classmethod unit_circle`()
Return the readonly *Path* of the unit circle.

For most cases, *Path.circle()* will be what you want.

`classmethod unit_circle_righthalf`()
Return a *Path* of the right half of a unit circle.

See *Path.circle* for the reference on the approximation used.

`classmethod unit_rectangle`()
Return a *Path* instance of the unit rectangle from (0, 0) to (1, 1).

`classmethod unit_regular_asterisk`(*numVertices*)
Return a *Path* for a unit regular asterisk with the given numVertices and radius of 1.0, centered at (0, 0).

`classmethod unit_regular_polygon`(*numVertices*)
Return a *Path* instance for a unit regular polygon with the given *numVertices* such that the circumscribing circle has radius 1.0, centered at (0, 0).

`classmethod unit_regular_star`(*numVertices*, *innerCircle=0.5*)
Return a *Path* for a unit regular star with the given numVertices and radius of 1.0, centered at (0, 0).

`property vertices`
The list of vertices in the *Path* as an Nx2 numpy array.

`classmethod wedge`(*theta1*, *theta2*, *n=None*)
Return the unit circle wedge from angles *theta1* to *theta2* (in degrees).

theta2 is unwrapped to produce the shortest wedge within 360 degrees. That is, if *theta2* > *theta1* + 360, the wedge will be from *theta1* to *theta2* - 360 and not a full circle plus some extra overlap.

If *n* is provided, it is the number of spline segments to make. If *n* is not provided, the number of spline segments is determined based on the delta between *theta1* and *theta2*.

See *Path.arc* for the reference on the approximation used.

matplotlib.path.**get_path_collection_extents**(*master_transform*, *paths*, *transforms*, *offsets*, *offset_transform*)

Given a sequence of *Path*s, *Transform*s objects, and offsets, as found in a *PathCollection*, returns the bounding box that encapsulates all of them.

Parameters

master_transform

[*Transform*] Global transformation applied to all paths.

paths

[list of *Path*]

transforms

[list of *Affine2D*]

offsets

[(N, 2) array-like]

offset_transform

[*Affine2D*] Transform applied to the offsets before offsetting the path.

Notes

The way that *paths*, *transforms* and *offsets* are combined follows the same method as for collections: Each is iterated over independently, so if you have 3 paths, 2 transforms and 1 offset, their combinations are as follows:

(A, A, A), (B, B, A), (C, A, A)

18.37 matplotlib.patheffects

Defines classes for path effects. The path effects are supported in *Text*, *Line2D* and *Patch*.

See also:

Path effects guide

class matplotlib.patheffects.**AbstractPathEffect**(*offset=0.0, 0.0*)

Bases: object

A base class for path effects.

Subclasses should override the `draw_path` method to add effect functionality.

Parameters

offset

[pair of floats] The offset to apply to the path, measured in points.

`draw_path`(*self*, *renderer*, *gc*, *tpath*, *affine*, *rgbFace=None*)
Derived should override this method. The arguments are the same as *matplotlib.backend_bases.RendererBase.draw_path()* except the first argument is a renderer.

`class matplotlib.patheffects.Normal`(*offset=0.0, 0.0*)
Bases: *matplotlib.patheffects.AbstractPathEffect*

The "identity" PathEffect.

The Normal PathEffect's sole purpose is to draw the original artist with no special path effect.

Parameters

offset

[pair of floats] The offset to apply to the path, measured in points.

`class matplotlib.patheffects.PathEffectRenderer`(*path_effects*, *renderer*)
Bases: *matplotlib.backend_bases.RendererBase*

Implements a Renderer which contains another renderer.

This proxy then intercepts draw calls, calling the appropriate *AbstractPathEffect* draw method.

Note: Not all methods have been overridden on this RendererBase subclass. It may be necessary to add further methods to extend the PathEffects capabilities further.

Parameters

path_effects

[iterable of *AbstractPathEffect*] The path effects which this renderer represents.

renderer

[*matplotlib.backend_bases.RendererBase* subclass]

`copy_with_path_effect`(*self*, *path_effects*)

draw_markers(*self*, *gc*, *marker_path*, *marker_trans*, *path*, **args*, ***kwargs*)
Draw a marker at each of the vertices in path.

This includes all vertices, including control points on curves. To avoid that behavior, those vertices should be removed before calling this function.

This provides a fallback implementation of draw_markers that makes multiple calls to *draw_path()*. Some backends may want to override this method in order to draw the marker only once and reuse it multiple times.

Parameters

gc

[*GraphicsContextBase*] The graphics context.

marker_trans

[*matplotlib.transforms.Transform*] An affine transform applied to the marker.

trans

[*matplotlib.transforms.Transform*] An affine transform applied to the path.

draw_path(*self*, *gc*, *tpath*, *affine*, *rgbFace=None*)
Draw a *Path* instance using the given affine transform.

draw_path_collection(*self*, *gc*, *master_transform*, *paths*, **args*, ***kwargs*)
Draw a collection of paths selecting drawing properties from the lists *facecolors*, *edgecolors*, *linewidths*, *linestyles* and *antialiaseds*. *offsets* is a list of offsets to apply to each of the paths. The offsets in *offsets* are first transformed by *offsetTrans* before being applied.

offset_position may be either "screen" or "data" depending on the space that the offsets are in; "data" is deprecated.

This provides a fallback implementation of *draw_path_collection()* that makes multiple calls to *draw_path()*. Some backends may want to override this in order to render each set of path data only once, and then reference that path multiple times with the different offsets, colors, styles etc. The generator methods `_iter_collection_raw_paths()` and `_iter_collection()` are provided to help with (and standardize) the implementation across backends. It is highly recommended to use those generators, so that changes to the behavior of *draw_path_collection()* can be made globally.

class matplotlib.patheffects.PathPatchEffect(*offset=0, 0*, ***kwargs*)
Bases: *matplotlib.patheffects.AbstractPathEffect*

Draws a *PathPatch* instance whose Path comes from the original PathEffect artist.

Parameters

offset

[pair of floats] The offset to apply to the path, in points.

****kwargs**

All keyword arguments are passed through to the *PathPatch* constructor. The properties which cannot be overridden are "path", "clip_box" "transform" and "clip_path".

`draw_path`(*self*, *renderer*, *gc*, *tpath*, *affine*, *rgbFace*)

Derived should override this method. The arguments are the same as *matplotlib.backend_bases.RendererBase.draw_path()* except the first argument is a renderer.

`class matplotlib.patheffects.SimpleLineShadow`(*offset=2, - 2*, *shadow_color='k'*, *alpha=0.3*, *rho=0.3*, ***kwargs*)

Bases: *matplotlib.patheffects.AbstractPathEffect*

A simple shadow via a line.

Parameters

offset

[pair of floats] The offset to apply to the path, in points.

shadow_color

[color, default: 'black'] The shadow color. A value of `None` takes the original artist's color with a scale factor of *rho*.

alpha

[float, default: 0.3] The alpha transparency of the created shadow patch.

rho

[float, default: 0.3] A scale factor to apply to the rgbFace color if `shadow_rgbFace` is `None`.

****kwargs**

Extra keywords are stored and passed through to `AbstractPathEffect._update_gc()`.

`draw_path`(*self*, *renderer*, *gc*, *tpath*, *affine*, *rgbFace*)

Overrides the standard draw_path to add the shadow offset and necessary color changes for the shadow.

`class matplotlib.patheffects.SimplePatchShadow`(*offset=2, - 2*, *shadow_rgbFace=None*, *alpha=None*, *rho=0.3*, ***kwargs*)

Bases: *matplotlib.patheffects.AbstractPathEffect*

A simple shadow via a filled patch.

Parameters

offset

[pair of floats] The offset of the shadow in points.

shadow_rgbFace

[color] The shadow color.

alpha

[float, default: 0.3] The alpha transparency of the created shadow patch. http://matplotlib.1069221.n5.nabble.com/path-effects-question-td27630.html

rho

[float, default: 0.3] A scale factor to apply to the rgbFace color if `shadow_rgbFace` is not specified.

****kwargs**

Extra keywords are stored and passed through to `AbstractPathEffect._update_gc()`.

`draw_path`(*self*, *renderer*, *gc*, *tpath*, *affine*, *rgbFace*)
Overrides the standard draw_path to add the shadow offset and necessary color changes for the shadow.

`class matplotlib.patheffects.Stroke`(*offset=0, 0*, ***kwargs*)
Bases: *matplotlib.patheffects.AbstractPathEffect*

A line based PathEffect which re-draws a stroke.

The path will be stroked with its gc updated with the given keyword arguments, i.e., the keyword arguments should be valid gc parameter values.

`draw_path`(*self*, *renderer*, *gc*, *tpath*, *affine*, *rgbFace*)
Draw the path with updated gc.

`class matplotlib.patheffects.withSimplePatchShadow`(*offset=2, -2*, *shadow_rgbFace=None*, *alpha=None*, *rho=0.3*, ***kwargs*)
Bases: *matplotlib.patheffects.SimplePatchShadow*

A shortcut PathEffect for applying *SimplePatchShadow* and then drawing the original Artist.

With this class you can use

```
artist.set_path_effects([path_effects.withSimplePatchShadow()])
```

as a shortcut for

```
artist.set_path_effects([path_effects.SimplePatchShadow(),
                         path_effects.Normal()])
```

Parameters

offset

[pair of floats] The offset of the shadow in points.

shadow_rgbFace

[color] The shadow color.

alpha

[float, default: 0.3] The alpha transparency of the created shadow patch. http://matplotlib.1069221.n5.nabble.com/path-effects-question-td27630.html

rho

[float, default: 0.3] A scale factor to apply to the rgbFace color if `shadow_rgbFace` is not specified.

****kwargs**

Extra keywords are stored and passed through to `AbstractPathEffect._update_gc()`.

`draw_path`(*self*, *renderer*, *gc*, *tpath*, *affine*, *rgbFace*)
Overrides the standard draw_path to add the shadow offset and necessary color changes for the shadow.

`class matplotlib.patheffects.withStroke`(*offset=0, 0*, ***kwargs*)
Bases: *matplotlib.patheffects.Stroke*

A shortcut PathEffect for applying *Stroke* and then drawing the original Artist.

With this class you can use

```
artist.set_path_effects([path_effects.withStroke()])
```

as a shortcut for

```
artist.set_path_effects([path_effects.Stroke(),
                         path_effects.Normal()])
```

The path will be stroked with its gc updated with the given keyword arguments, i.e., the keyword arguments should be valid gc parameter values.

`draw_path`(*self*, *renderer*, *gc*, *tpath*, *affine*, *rgbFace*)
Draw the path with updated gc.

18.38 matplotlib.pyplot

18.38.1 Pyplot function overview

pyplot	*matplotlib.pyplot* is a state-based interface to matplotlib.

matplotlib.pyplot

matplotlib.pyplot is a state-based interface to matplotlib. It provides a MATLAB-like way of plotting.

pyplot is mainly intended for interactive plots and simple cases of programmatic plot generation:

```
import numpy as np
import matplotlib.pyplot as plt

x = np.arange(0, 5, 0.1)
y = np.sin(x)
plt.plot(x, y)
```

The object-oriented API is recommended for more complex plots.

Functions

acorr(x, *[, data])	Plot the autocorrelation of *x*.
angle_spectrum(x[, Fs, Fc, window, pad_to, ...])	Plot the angle spectrum.
annotate(text, xy, *args, **kwargs)	Annotate the point *xy* with text *text*.
arrow(x, y, dx, dy, **kwargs)	Add an arrow to the axes.
autoscale([enable, axis, tight])	Autoscale the axis view to the data (toggle).
autumn()	Set the colormap to "autumn".
axes([arg])	Add an axes to the current figure and make it the current axes.
axhline([y, xmin, xmax])	Add a horizontal line across the axis.
axhspan(ymin, ymax[, xmin, xmax])	Add a horizontal span (rectangle) across the axis.
axis(*args[, emit])	Convenience method to get or set some axis properties.
axline(xy1[, xy2, slope])	Add an infinitely long straight line.
axvline([x, ymin, ymax])	Add a vertical line across the axes.

continues on next page

Table 179 – continued from previous page

axvspan(xmin, xmax[, ymin, ymax])	Add a vertical span (rectangle) across the axes.
bar(x, height[, width, bottom, align, data])	Make a bar plot.
barbs(*args[, data])	Plot a 2D field of barbs.
barh(y, width[, height, left, align])	Make a horizontal bar plot.
bone()	Set the colormap to "bone".
box([on])	Turn the axes box on or off on the current axes.
boxplot(x[, notch, sym, vert, whis, ...])	Make a box and whisker plot.
broken_barh(xranges, yrange, *[, data])	Plot a horizontal sequence of rectangles.
cla()	Clear the current axes.
clabel(CS[, levels])	Label a contour plot.
clf()	Clear the current figure.
clim([vmin, vmax])	Set the color limits of the current image.
close([fig])	Close a figure window.
cohere(x, y[, NFFT, Fs, Fc, detrend, ...])	Plot the coherence between *x* and *y*.
colorbar([mappable, cax, ax])	Add a colorbar to a plot.
connect(s, func)	Bind function *func* to event *s*.
contour(*args[, data])	Plot contours.
contourf(*args[, data])	Plot contours.
cool()	Set the colormap to "cool".
copper()	Set the colormap to "copper".
csd(x, y[, NFFT, Fs, Fc, detrend, window, ...])	Plot the cross-spectral density.
delaxes([ax])	Remove an *Axes* (defaulting to the current axes) from its figure.
disconnect(cid)	Disconnect the callback with id *cid*.
draw()	Redraw the current figure.
draw_if_interactive()	
errorbar(x, y[, yerr, xerr, fmt, ecolor, ...])	Plot y versus x as lines and/or markers with attached errorbars.
eventplot(positions[, orientation, ...])	Plot identical parallel lines at the given positions.
figimage(X[, xo, yo, alpha, norm, cmap, ...])	Add a non-resampled image to the figure.
figlegend(*args, **kwargs)	Place a legend on the figure.
fignum_exists(num)	Return whether the figure with the given id exists.
figtext(x, y, s[, fontdict])	Add text to figure.
figure([num, figsize, dpi, facecolor, ...])	Create a new figure, or activate an existing figure.
fill(*args[, data])	Plot filled polygons.
fill_between(x, y1[, y2, where, ...])	Fill the area between two horizontal curves.

continues on next page

Table 179 – continued from previous page

fill_betweenx(y, x1[, x2, where, step, ...])	Fill the area between two vertical curves.
findobj([o, match, include_self])	Find artist objects.
flag()	Set the colormap to "flag".
gca(**kwargs)	Get the current axes, creating one if necessary.
gcf()	Get the current figure.
gci()	Get the current colorable artist.
get(obj, *args, **kwargs)	Return the value of an object's *property*, or print all of them.
get_current_fig_manager()	Return the figure manager of the current figure.
get_figlabels()	Return a list of existing figure labels.
get_fignums()	Return a list of existing figure numbers.
get_plot_commands()	Get a sorted list of all of the plotting commands.
getp(obj, *args, **kwargs)	Return the value of an object's *property*, or print all of them.
ginput([n, timeout, show_clicks, mouse_add, ...])	Blocking call to interact with a figure.
gray()	Set the colormap to "gray".
grid([b, which, axis])	Configure the grid lines.
hexbin(x, y[, C, gridsize, bins, xscale, ...])	Make a 2D hexagonal binning plot of points *x*, *y*.
hist(x[, bins, range, density, weights, ...])	Plot a histogram.
hist2d(x, y[, bins, range, density, ...])	Make a 2D histogram plot.
hlines(y, xmin, xmax[, colors, linestyles, ...])	Plot horizontal lines at each *y* from *xmin* to *xmax*.
hot()	Set the colormap to "hot".
hsv()	Set the colormap to "hsv".
imread(fname[, format])	Read an image from a file into an array.
imsave(fname, arr, **kwargs)	Save an array as an image file.
imshow(X[, cmap, norm, aspect, ...])	Display data as an image, i.e., on a 2D regular raster.
inferno()	Set the colormap to "inferno".
install_repl_displayhook()	Install a repl display hook so that any stale figure are automatically redrawn when control is returned to the repl.
ioff()	Turn the interactive mode off.
ion()	Turn the interactive mode on.
isinteractive()	Return if pyplot is in "interactive mode" or not.
jet()	Set the colormap to "jet".
legend(*args, **kwargs)	Place a legend on the axes.

continues on next page

Table 179 – continued from previous page

locator_params([axis, tight])	Control behavior of major tick locators.
loglog(*args, **kwargs)	Make a plot with log scaling on both the x and y axis.
magma()	Set the colormap to "magma".
magnitude_spectrum(x[, Fs, Fc, window, ...])	Plot the magnitude spectrum.
margins(*margins[, x, y, tight])	Set or retrieve autoscaling margins.
matshow(A[, fignum])	Display an array as a matrix in a new figure window.
minorticks_off()	Remove minor ticks from the axes.
minorticks_on()	Display minor ticks on the axes.
new_figure_manager(num, *args, **kwargs)	Create a new figure manager instance.
nipy_spectral()	Set the colormap to "nipy_spectral".
pause(interval)	Run the GUI event loop for *interval* seconds.
pcolor(*args[, shading, alpha, norm, cmap, ...])	Create a pseudocolor plot with a non-regular rectangular grid.
pcolormesh(*args[, alpha, norm, cmap, vmin, ...])	Create a pseudocolor plot with a non-regular rectangular grid.
phase_spectrum(x[, Fs, Fc, window, pad_to, ...])	Plot the phase spectrum.
pie(x[, explode, labels, colors, autopct, ...])	Plot a pie chart.
pink()	Set the colormap to "pink".
plasma()	Set the colormap to "plasma".
plot(*args[, scalex, scaley, data])	Plot y versus x as lines and/or markers.
plot_date(x, y[, fmt, tz, xdate, ydate, data])	Plot data that contains dates.
polar(*args, **kwargs)	Make a polar plot.
prism()	Set the colormap to "prism".
psd(x[, NFFT, Fs, Fc, detrend, window, ...])	Plot the power spectral density.
quiver(*args[, data])	Plot a 2D field of arrows.
quiverkey(Q, X, Y, U, label, **kw)	Add a key to a quiver plot.
rc(group, **kwargs)	Set the current *rcParams*.
rc_context([rc, fname])	Return a context manager for temporarily changing rcParams.
rcdefaults()	Restore the *rcParams* from Matplotlib's internal default style.
rgrids([radii, labels, angle, fmt])	Get or set the radial gridlines on the current polar plot.
savefig(*args, **kwargs)	Save the current figure.
sca(ax)	Set the current Axes to *ax* and the current Figure to the parent of *ax*.

continues on next page

Table 179 – continued from previous page

scatter(x, y[, s, c, marker, cmap, norm, ...])	A scatter plot of *y* vs.
sci(im)	Set the current image.
semilogx(*args, **kwargs)	Make a plot with log scaling on the x axis.
semilogy(*args, **kwargs)	Make a plot with log scaling on the y axis.
set_cmap(cmap)	Set the default colormap, and applies it to the current image if any.
setp(obj, *args, **kwargs)	Set a property on an artist object.
show(*[, block])	Display all open figures.
specgram(x[, NFFT, Fs, Fc, detrend, window, ...])	Plot a spectrogram.
spring()	Set the colormap to "spring".
spy(Z[, precision, marker, markersize, ...])	Plot the sparsity pattern of a 2D array.
stackplot(x, *args[, labels, colors, ...])	Draw a stacked area plot.
stem(*args[, linefmt, markerfmt, basefmt, ...])	Create a stem plot.
step(x, y, *args[, where, data])	Make a step plot.
streamplot(x, y, u, v[, density, linewidth, ...])	Draw streamlines of a vector flow.
subplot(*args, **kwargs)	Add a subplot to the current figure.
subplot2grid(shape, loc[, rowspan, colspan, fig])	Create a subplot at a specific location inside a regular grid.
subplot_mosaic(layout, *[, subplot_kw, ...])	Build a layout of Axes based on ASCII art or nested lists.
subplot_tool([targetfig])	Launch a subplot tool window for a figure.
subplots([nrows, ncols, sharex, sharey, ...])	Create a figure and a set of subplots.
subplots_adjust([left, bottom, right, top, ...])	Adjust the subplot layout parameters.
summer()	Set the colormap to "summer".
suptitle(t, **kwargs)	Add a centered title to the figure.
switch_backend(newbackend)	Close all open figures and set the Matplotlib backend.
table([cellText, cellColours, cellLoc, ...])	Add a table to an *Axes*.
text(x, y, s[, fontdict])	Add text to the axes.
thetagrids([angles, labels, fmt])	Get or set the theta gridlines on the current polar plot.
tick_params([axis])	Change the appearance of ticks, tick labels, and gridlines.
ticklabel_format(*[, axis, style, ...])	Configure the *ScalarFormatter* used by default for linear axes.

continues on next page

Table 179 – continued from previous page

tight_layout(*[, pad, h_pad, w_pad, rect])	Adjust the padding between and around subplots.
title(label[, fontdict, loc, pad, y])	Set a title for the axes.
tricontour(*args, **kwargs)	Draw contour lines on an unstructured triangular grid.
tricontourf(*args, **kwargs)	Draw contour regions on an unstructured triangular grid.
tripcolor(*args[, alpha, norm, cmap, vmin, ...])	Create a pseudocolor plot of an unstructured triangular grid.
triplot(*args, **kwargs)	Draw a unstructured triangular grid as lines and/or markers.
twinx([ax])	Make and return a second axes that shares the *x*-axis.
twiny([ax])	Make and return a second axes that shares the *y*-axis.
uninstall_repl_displayhook()	Uninstall the matplotlib display hook.
violinplot(dataset[, positions, vert, ...])	Make a violin plot.
viridis()	Set the colormap to "viridis".
vlines(x, ymin, ymax[, colors, linestyles, ...])	Plot vertical lines.
waitforbuttonpress([timeout])	Blocking call to interact with the figure.
winter()	Set the colormap to "winter".
xcorr(x, y[, normed, detrend, usevlines, ...])	Plot the cross correlation between *x* and *y*.
xkcd([scale, length, randomness])	Turn on xkcd sketch-style drawing mode.This will only have effect on things drawn after this function is called..
xlabel(xlabel[, fontdict, labelpad, loc])	Set the label for the x-axis.
xlim(*args, **kwargs)	Get or set the x limits of the current axes.
xscale(value, **kwargs)	Set the x-axis scale.
xticks([ticks, labels])	Get or set the current tick locations and labels of the x-axis.
ylabel(ylabel[, fontdict, labelpad, loc])	Set the label for the y-axis.
ylim(*args, **kwargs)	Get or set the y-limits of the current axes.
yscale(value, **kwargs)	Set the y-axis scale.
yticks([ticks, labels])	Get or set the current tick locations and labels of the y-axis.

matplotlib.pyplot.acorr

matplotlib.pyplot.acorr(*x*, *, *data=None*, ***kwargs*)

Plot the autocorrelation of *x*.

Parameters

x

[array-like]

detrend

[callable, default: *mlab.detrend_none* (no detrending)] A detrending function applied to *x*. It must have the signature

```
detrend(x: np.ndarray) -> np.ndarray
```

normed

[bool, default: True] If `True`, input vectors are normalised to unit length.

usevlines

[bool, default: True] Determines the plot style.

If `True`, vertical lines are plotted from 0 to the acorr value using *Axes.vlines*. Additionally, a horizontal line is plotted at y=0 using *Axes.axhline*.

If `False`, markers are plotted at the acorr values using *Axes.plot*.

maxlags

[int, default: 10] Number of lags to show. If `None`, will return all `2 * len(x) - 1` lags.

Returns

lags

[array (length `2*maxlags+1`)] The lag vector.

c

[array (length `2*maxlags+1`)] The auto correlation vector.

line

[*LineCollection* or *Line2D*] *Artist* added to the axes of the correlation:

- *LineCollection* if *usevlines* is True.
- *Line2D* if *usevlines* is False.

b

[*Line2D* or None] Horizontal line at 0 if *usevlines* is True None *usevlines* is False.

Other Parameters

linestyle

[*Line2D* property, optional] The linestyle for plotting the data points. Only used if *usevlines* is `False`.

marker

[str, default: 'o'] The marker for plotting the data points. Only used if *usevlines* is `False`.

****kwargs**

Additional parameters are passed to *Axes.vlines* and *Axes.axhline* if *usevlines* is `True`; otherwise they are passed to *Axes.plot*.

Notes

The cross correlation is performed with `numpy.correlate` with `mode = "full"`.

Note: In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, the following arguments can also be string `s`, which is interpreted as `data[s]` (unless this raises an exception): *x*.

Objects passed as **data** must support item access (`data[s]`) and membership test (`s in data`).

Examples using `matplotlib.pyplot.acorr`

- sphx_glr_gallery_lines_bars_and_markers_xcorr_acorr_demo.py

matplotlib.pyplot.angle_spectrum

matplotlib.pyplot.**angle_spectrum**(*x*, *Fs=None*, *Fc=None*, *window=None*, *pad_to=None*, *sides=None*, *, *data=None*, ***kwargs*)

Plot the angle spectrum.

Compute the angle spectrum (wrapped phase spectrum) of *x*. Data is padded to a length of *pad_to* and the windowing function *window* is applied to the signal.

Parameters

x

[1-D array or sequence] Array or sequence containing the data.

Fs

[float, default: 2] The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, *freqs*, in cycles per time unit.

window

[callable or ndarray, default: *window_hanning*] A function or a vector of length *NFFT*. To create window vectors see *window_hanning*, *window_none*, numpy.blackman, numpy.hamming, numpy.bartlett, scipy.signal, scipy.signal.get_window, etc. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

sides

[{'default', 'onesided', 'twosided'}, optional] Which sides of the spectrum to return. 'default' is one-sided for real data and two-sided for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

pad_to

[int, optional] The number of points to which the data segment is padded when performing the FFT. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to fft(). The default is None, which sets *pad_to* equal to the length of the input signal (i.e. no padding).

Fc

[int, default: 0] The center frequency of *x*, which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

Returns

spectrum

[1-D array] The values for the angle spectrum in radians (real valued).

freqs

[1-D array] The frequencies corresponding to the elements in *spectrum*.

line

[*Line2D*] The line created by this function.

Other Parameters

****kwargs**

Keyword arguments control the *Line2D* properties:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value
alpha	float or None
animated	bool
antialiased or aa	bool
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color or c	color
contains	unknown
dash_capstyle	{'butt', 'round', 'projecting'}
dash_joinstyle	{'miter', 'round', 'bevel'}
dashes	sequence of floats (on/off ink in points) or (None, None)
data	(2, N) array or two 1D arrays
drawstyle or ds	{'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'd
figure	*Figure*
fillstyle	{'full', 'left', 'right', 'bottom', 'top', 'none'}
gid	str
in_layout	bool
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float
marker	marker style string, *Path* or *MarkerStyle*
markeredgecolor or mec	color
markeredgewidth or mew	float
markerfacecolor or mfc	color

Table 180 – continued from previous page

Property	Description
markerfacecoloralt or mfcalt	color
markersize or ms	float
markevery	None or int or (int, int) or slice or List[int] or float or (float, float)
path_effects	*AbstractPathEffect*
picker	unknown
pickradius	float
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
solid_capstyle	{'butt', 'round', 'projecting'}
solid_joinstyle	{'miter', 'round', 'bevel'}
transform	*matplotlib.transforms.Transform*
url	str
visible	bool
xdata	1D array
ydata	1D array
zorder	float

See also:

magnitude_spectrum

Plots the magnitudes of the corresponding frequencies.

phase_spectrum

Plots the unwrapped version of this function.

specgram

Can plot the angle spectrum of segments within the signal in a colormap.

Notes

Note: In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, the following arguments can also be string `s`, which is interpreted as `data[s]` (unless this raises an exception): *x*.

Objects passed as **data** must support item access (`data[s]`) and membership test (`s in data`).

Examples using `matplotlib.pyplot.angle_spectrum`

matplotlib.pyplot.annotate

`matplotlib.pyplot.`**`annotate`**(*text*, *xy*, **args*, ***kwargs*)

Annotate the point *xy* with text *text*.

In the simplest form, the text is placed at *xy*.

Optionally, the text can be displayed in another position *xytext*. An arrow pointing from the text to the annotated point *xy* can then be added by defining *arrowprops*.

Parameters

text

[str] The text of the annotation. *s* is a deprecated synonym for this parameter.

xy

[(float, float)] The point *(x, y)* to annotate. The coordinate system is determined by *xycoords*.

xytext

[(float, float), default: *xy*] The position *(x, y)* to place the text at. The coordinate system is determined by *textcoords*.

xycoords

[str or *Artist* or *Transform* or callable or (float, float), default: 'data'] The coordinate system that *xy* is given in. The following types of values are supported:

- One of the following strings:

Value	Description
'figure points'	Points from the lower left of the figure
'figure pixels'	Pixels from the lower left of the figure
'figure fraction'	Fraction of figure from lower left
'axes points'	Points from lower left corner of axes
'axes pixels'	Pixels from lower left corner of axes
'axes fraction'	Fraction of axes from lower left
'data'	Use the coordinate system of the object being annotated (default)
'polar'	*(theta, r)* if not native 'data' coordinates

- An *Artist*: *xy* is interpreted as a fraction of the artist's *Bbox*. E.g. *(0, 0)* would be the lower left corner of the bounding box and *(0.5, 1)* would be the center top of the bounding box.
- A *Transform* to transform *xy* to screen coordinates.
- A function with one of the following signatures:

```
def transform(renderer) -> Bbox
def transform(renderer) -> Transform
```

  where *renderer* is a *RendererBase* subclass.

  The result of the function is interpreted like the *Artist* and *Transform* cases above.

- A tuple *(xcoords, ycoords)* specifying separate coordinate systems for *x* and *y*. *xcoords* and *ycoords* must each be of one of the above described types.

See *Advanced Annotations* for more details.

textcoords

[str or *Artist* or *Transform* or callable or (float, float), default: value of *xycoords*] The coordinate system that *xytext* is given in.

All *xycoords* values are valid as well as the following strings:

Value	Description
'offset points'	Offset (in points) from the *xy* value
'offset pixels'	Offset (in pixels) from the *xy* value

arrowprops

[dict, optional] The properties used to draw a *FancyArrowPatch* arrow between the positions *xy* and *xytext*.

If *arrowprops* does not contain the key 'arrowstyle' the allowed keys are:

Key	Description
width	The width of the arrow in points
headwidth	The width of the base of the arrow head in points
headlength	The length of the arrow head in points
shrink	Fraction of total length to shrink from both ends
?	Any key to *matplotlib.patches.FancyArrowPatch*

If *arrowprops* contains the key 'arrowstyle' the above keys are forbidden. The allowed values of `'arrowstyle'` are:

Name	Attrs
`'-'`	None
`'->'`	head_length=0.4,head_width=0.2
`'-['`	widthB=1.0,lengthB=0.2,angleB=None
`'|-|'`	widthA=1.0,widthB=1.0
`'-|>'`	head_length=0.4,head_width=0.2
`'<-'`	head_length=0.4,head_width=0.2
`'<->'`	head_length=0.4,head_width=0.2
`'<|-'`	head_length=0.4,head_width=0.2
`'<|-|>'`	head_length=0.4,head_width=0.2
`'fancy'`	head_length=0.4,head_width=0.4,tail_width=0.4
`'simple'`	head_length=0.5,head_width=0.5,tail_width=0.2
`'wedge'`	tail_width=0.3,shrink_factor=0.5

Valid keys for *FancyArrowPatch* are:

Key	Description
arrowstyle	the arrow style
connectionstyle	the connection style
relpos	default is (0.5, 0.5)
patchA	default is bounding box of the text
patchB	default is None
shrinkA	default is 2 points
shrinkB	default is 2 points
mutation_scale	default is text size (in points)
mutation_aspect	default is 1.
?	any key for *matplotlib.patches.PathPatch*

Defaults to None, i.e. no arrow is drawn.

annotation_clip

[bool or None, default: None] Whether to draw the annotation when the annotation point *xy* is outside the axes area.

- If *True*, the annotation will only be drawn when *xy* is within the axes.
- If *False*, the annotation will always be drawn.
- If *None*, the annotation will only be drawn when *xy* is within the axes and *xycoords* is 'data'.

****kwargs**

Additional kwargs are passed to *Text*.

Returns

Annotation

See also:

Advanced Annotations

Examples using `matplotlib.pyplot.annotate`

- sphx_glr_gallery_lines_bars_and_markers_barchart.py
- sphx_glr_gallery_statistics_barchart_demo.py
- sphx_glr_gallery_text_labels_and_annotations_mathtext_examples.py
- sphx_glr_gallery_pyplots_annotate_transform.py
- sphx_glr_gallery_pyplots_annotation_basic.py
- sphx_glr_gallery_pyplots_annotation_polar.py
- *Pyplot tutorial*
- *Annotations*

matplotlib.pyplot.arrow

`matplotlib.pyplot.arrow`(*x*, *y*, *dx*, *dy*, ***kwargs*)

Add an arrow to the axes.

This draws an arrow from `(x, y)` to `(x+dx, y+dy)`.

Parameters

x, y

[float] The x and y coordinates of the arrow base.

dx, dy

[float] The length of the arrow along x and y direction.

width: float, default: 0.001

Width of full arrow tail.

length_includes_head: bool, default: False

True if head is to be counted in calculating the length.

head_width: float or None, default: 3*width

Total width of the full arrow head.

head_length: float or None, default: 1.5*head_width

Length of arrow head.

shape: ['full', 'left', 'right'], default: 'full'

Draw the left-half, right-half, or full arrow.

overhang: float, default: 0

Fraction that the arrow is swept back (0 overhang means triangular shape). Can be negative or greater than one.

head_starts_at_zero: bool, default: False

If True, the head starts being drawn at coordinate 0 instead of ending at coordinate 0.

****kwargs**

Patch properties:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and retur
alpha	float or None
animated	bool
antialiased or aa	unknown
capstyle	{'butt', 'round', 'projecting'}
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color	color
contains	unknown
edgecolor or ec	color or None or 'auto'
facecolor or fc	color or None

Table 181 – continued from previous page

Property	Description
figure	*Figure*
fill	bool
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float or None
path_effects	*AbstractPathEffect*
picker	None or bool or callable
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
visible	bool
zorder	float

Returns

FancyArrow

The created *FancyArrow* object.

Notes

The resulting arrow is affected by the axes aspect ratio and limits. This may produce an arrow whose head is not square with its stem. To create an arrow whose head is square with its stem, use *annotate()* for example:

```
>>> ax.annotate("", xy=(0.5, 0.5), xytext=(0, 0),
...             arrowprops=dict(arrowstyle="->"))
```

Examples using `matplotlib.pyplot.arrow`

- sphx_glr_gallery_text_labels_and_annotations_arrow_demo.py

matplotlib.pyplot.autoscale

matplotlib.pyplot.**autoscale**(*enable=True*, *axis='both'*, *tight=None*)

Autoscale the axis view to the data (toggle).

Convenience method for simple axis view autoscaling. It turns autoscaling on or off, and then, if autoscaling for either axis is on, it performs the autoscaling on the specified axis or axes.

Parameters

enable

[bool or None, default: True] True turns autoscaling on, False turns it off. None leaves the autoscaling state unchanged.

axis

[{'both', 'x', 'y'}, default: 'both'] Which axis to operate on.

tight

[bool or None, default: None] If True, first set the margins to zero. Then, this argument is forwarded to `autoscale_view` (regardless of its value); see the description of its behavior there.

Examples using `matplotlib.pyplot.autoscale`

matplotlib.pyplot.autumn

matplotlib.pyplot.**autumn**()

Set the colormap to "autumn".

This changes the default colormap as well as the colormap of the current image if there is one. See `help(colormaps)` for more information.

Examples using `matplotlib.pyplot.autumn`

matplotlib.pyplot.axes

matplotlib.pyplot.**axes**(*arg=None*, ***kwargs*)

Add an axes to the current figure and make it the current axes.

Call signatures:

```
plt.axes()
plt.axes(rect, projection=None, polar=False, **kwargs)
plt.axes(ax)
```

Parameters

arg

[None or 4-tuple] The exact behavior of this function depends on the type:

- *None*: A new full window axes is added using `subplot(111, **kwargs)`.
- 4-tuple of floats *rect* = `[left, bottom, width, height]`. A new axes is added with dimensions *rect* in normalized (0, 1) units using *add_axes* on the current figure.

projection

[{None, 'aitoff', 'hammer', 'lambert', 'mollweide', 'polar', 'rectilinear', str}, optional] The projection type of the *Axes*. *str* is the name of a custom projection, see *projections*. The default None results in a 'rectilinear' projection.

polar

[bool, default: False] If True, equivalent to projection='polar'.

sharex, sharey

[*Axes*, optional] Share the x or y *axis* with sharex and/or sharey. The axis will have the same limits, ticks, and scale as the axis of the shared axes.

label

[str] A label for the returned axes.

Returns

Axes, or a subclass of *Axes*

The returned axes class depends on the projection used. It is *Axes* if rectilinear projection is used and *projections.polar.PolarAxes* if polar projection is used.

Other Parameters

**kwargs

This method also takes the keyword arguments for the returned axes class. The keyword arguments for the rectilinear axes class *Axes* can be found in the following table but there might also be other keyword arguments if another projection is used, see the actual axes class.

Property	Description
adjustable	{'box', 'datalim'}
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and r
alpha	float or None
anchor	2-tuple of floats or {'C', 'SW', 'S', 'SE', ...}
animated	bool
aspect	{'auto'} or num
autoscale_on	bool
autoscalex_on	bool
autoscaley_on	bool
axes_locator	Callable[[Axes, Renderer], Bbox]
axisbelow	bool or 'line'
box_aspect	None, or a number
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
contains	unknown
facecolor or fc	color
figure	*Figure*
frame_on	bool
gid	str
in_layout	bool
label	object
navigate	bool
navigate_mode	unknown
path_effects	*AbstractPathEffect*
picker	None or bool or callable
position	[left, bottom, width, height] or *Bbox*
prop_cycle	unknown
rasterization_zorder	float or None
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
title	str
transform	*Transform*
url	str
visible	bool
xbound	unknown
xlabel	str
xlim	(bottom: float, top: float)
xmargin	float greater than -0.5
xscale	{"linear", "log", "symlog", "logit", ...}
xticklabels	unknown
xticks	unknown
ybound	unknown

Table 182 – continued from previous page

Property	Description
ylabel	str
ylim	(bottom: float, top: float)
ymargin	float greater than -0.5
yscale	{"linear", "log", "symlog", "logit", ...}
yticklabels	unknown
yticks	unknown
zorder	float

See also:

Figure.add_axes

pyplot.subplot

Figure.add_subplot

Figure.subplots

pyplot.subplots

Notes

If the figure already has a axes with key (*args*, *kwargs*) then it will simply make that axes current and return it. This behavior is deprecated. Meanwhile, if you do not want this behavior (i.e., you want to force the creation of a new axes), you must use a unique set of args and kwargs. The axes *label* attribute has been exposed for this purpose: if you want two axes that are otherwise identical to be added to the figure, make sure you give them unique labels.

Examples

```
# Creating a new full window axes
plt.axes()

# Creating a new axes with specified dimensions and some kwargs
plt.axes((left, bottom, width, height), facecolor='w')
```

Examples using `matplotlib.pyplot.axes`

- sphx_glr_gallery_subplots_axes_and_figures_subplots_adjust.py
- sphx_glr_gallery_text_labels_and_annotations_arrow_simple_demo.py
- sphx_glr_gallery_text_labels_and_annotations_mathtext_examples.py
- sphx_glr_gallery_axes_grid1_make_room_for_ylabel_using_axesgrid.py
- sphx_glr_gallery_event_handling_lasso_demo.py
- sphx_glr_gallery_widgets_buttons.py
- sphx_glr_gallery_widgets_check_buttons.py
- sphx_glr_gallery_widgets_radio_buttons.py
- sphx_glr_gallery_widgets_slider_demo.py

matplotlib.pyplot.axhline

matplotlib.pyplot.**axhline**(*y=0*, *xmin=0*, *xmax=1*, ***kwargs*)

Add a horizontal line across the axis.

Parameters

y

[float, default: 0] y position in data coordinates of the horizontal line.

xmin

[float, default: 0] Should be between 0 and 1, 0 being the far left of the plot, 1 the far right of the plot.

xmax

[float, default: 1] Should be between 0 and 1, 0 being the far left of the plot, 1 the far right of the plot.

Returns

Line2D

Other Parameters

****kwargs**

Valid keyword arguments are *Line2D* properties, with the exception of 'transform':

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value
alpha	float or None
animated	bool
antialiased or aa	bool
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color or c	color
contains	unknown
dash_capstyle	{'butt', 'round', 'projecting'}
dash_joinstyle	{'miter', 'round', 'bevel'}
dashes	sequence of floats (on/off ink in points) or (None, None)
data	(2, N) array or two 1D arrays
drawstyle or ds	{'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'd
figure	*Figure*
fillstyle	{'full', 'left', 'right', 'bottom', 'top', 'none'}
gid	str
in_layout	bool
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float
marker	marker style string, *Path* or *MarkerStyle*
markeredgecolor or mec	color
markeredgewidth or mew	float
markerfacecolor or mfc	color
markerfacecoloralt or mfcalt	color
markersize or ms	float
markevery	None or int or (int, int) or slice or List[int] or float or (float, float)
path_effects	*AbstractPathEffect*
picker	unknown
pickradius	float
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
solid_capstyle	{'butt', 'round', 'projecting'}
solid_joinstyle	{'miter', 'round', 'bevel'}
transform	*matplotlib.transforms.Transform*
url	str
visible	bool
xdata	1D array
ydata	1D array
zorder	float

See also:

hlines

Add horizontal lines in data coordinates.

axhspan

Add a horizontal span (rectangle) across the axis.

axline

Add a line with an arbitrary slope.

Examples

- draw a thick red hline at 'y' = 0 that spans the xrange:

```
>>> axhline(linewidth=4, color='r')
```

- draw a default hline at 'y' = 1 that spans the xrange:

```
>>> axhline(y=1)
```

- draw a default hline at 'y' = .5 that spans the middle half of the xrange:

```
>>> axhline(y=.5, xmin=0.25, xmax=0.75)
```

Examples using `matplotlib.pyplot.axhline`

- sphx_glr_gallery_text_labels_and_annotations_multiline.py
- sphx_glr_gallery_pyplots_axline.py
- sphx_glr_gallery_color_color_cycle_default.py
- sphx_glr_gallery_misc_zorder_demo.py

matplotlib.pyplot.axhspan

matplotlib.pyplot.**axhspan**(*ymin*, *ymax*, *xmin=0*, *xmax=1*, ***kwargs*)

Add a horizontal span (rectangle) across the axis.

The rectangle spans from *ymin* to *ymax* vertically, and, by default, the whole x-axis horizontally. The x-span can be set using *xmin* (default: 0) and *xmax* (default: 1) which are in axis units; e.g. `xmin = 0.5` always refers to the middle of the x-axis regardless of the limits set by *set_xlim*.

Parameters

ymin

[float] Lower y-coordinate of the span, in data units.

ymax

[float] Upper y-coordinate of the span, in data units.

xmin

[float, default: 0] Lower x-coordinate of the span, in x-axis (0-1) units.

xmax

[float, default: 1] Upper x-coordinate of the span, in x-axis (0-1) units.

Returns

Polygon

Horizontal span (rectangle) from (xmin, ymin) to (xmax, ymax).

Other Parameters

****kwargs**

[*Polygon* properties]

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and retur
alpha	float or None
animated	bool
antialiased or aa	unknown
capstyle	{'butt', 'round', 'projecting'}
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color	color
contains	unknown
edgecolor or ec	color or None or 'auto'
facecolor or fc	color or None
figure	*Figure*
fill	bool
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float or None
path_effects	*AbstractPathEffect*

Table 184 – continued from previous page

Property	Description
picker	None or bool or callable
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
visible	bool
zorder	float

See also:

axvspan

Add a vertical span across the axes.

Examples using `matplotlib.pyplot.axhspan`

- sphx_glr_gallery_subplots_axes_and_figures_axhspan_demo.py

matplotlib.pyplot.axis

matplotlib.pyplot.**axis**(**args*, *emit=True*, ***kwargs*)

Convenience method to get or set some axis properties.

Call signatures:

```
xmin, xmax, ymin, ymax = axis()
xmin, xmax, ymin, ymax = axis([xmin, xmax, ymin, ymax])
xmin, xmax, ymin, ymax = axis(option)
xmin, xmax, ymin, ymax = axis(**kwargs)
```

Parameters

xmin, xmax, ymin, ymax

[float, optional] The axis limits to be set. This can also be achieved using

```
ax.set(xlim=(xmin, xmax), ylim=(ymin, ymax))
```

option

[bool or str] If a bool, turns axis lines and labels on or off. If a string, possible values are:

Value	Description
'on'	Turn on axis lines and labels. Same as `True`.
'off'	Turn off axis lines and labels. Same as `False`.
'equal'	Set equal scaling (i.e., make circles circular) by changing axis limits. This is the same as `ax.set_aspect('equal', adjustable='datalim')`. Explicit data limits may not be respected in this case.
'scaled'	Set equal scaling (i.e., make circles circular) by changing dimensions of the plot box. This is the same as `ax.set_aspect('equal', adjustable='box', anchor='C')`. Additionally, further autoscaling will be disabled.
'tight'	Set limits just large enough to show all data, then disable further autoscaling.
'auto'	Automatic scaling (fill plot box with data).
'image'	'scaled' with axis limits equal to data limits.
'square'	Square plot; similar to 'scaled', but initially forcing `xmax-xmin == ymax-ymin`.

emit

[bool, default: True] Whether observers are notified of the axis limit change. This option is passed on to *set_xlim* and *set_ylim*.

Returns

xmin, xmax, ymin, ymax

[float] The axis limits.

See also:

matplotlib.axes.Axes.set_xlim

matplotlib.axes.Axes.set_ylim

Examples using `matplotlib.pyplot.axis`

- sphx_glr_gallery_lines_bars_and_markers_fill.py
- sphx_glr_gallery_text_labels_and_annotations_autowrap.py
- sphx_glr_gallery_text_labels_and_annotations_text_alignment.py
- sphx_glr_gallery_shapes_and_collections_artist_reference.py
- *Pyplot tutorial*

matplotlib.pyplot.axline

matplotlib.pyplot.**axline**(*xy1*, *xy2=None*, *, *slope=None*, ***kwargs*)

Add an infinitely long straight line.

The line can be defined either by two points *xy1* and *xy2*, or by one point *xy1* and a *slope*.

This draws a straight line "on the screen", regardless of the x and y scales, and is thus also suitable for drawing exponential decays in semilog plots, power laws in loglog plots, etc. However, *slope* should only be used with linear scales; It has no clear meaning for all other scales, and thus the behavior is undefined. Please specify the line using the points *xy1*, *xy2* for non-linear scales.

Parameters

xy1, xy2

[(float, float)] Points for the line to pass through. Either *xy2* or *slope* has to be given.

slope

[float, optional] The slope of the line. Either *xy2* or *slope* has to be given.

Returns

Line2D

Other Parameters

****kwargs**

Valid kwargs are *Line2D* properties, with the exception of 'transform':

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value
alpha	float or None
animated	bool
antialiased or aa	bool
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color or c	color
contains	unknown
dash_capstyle	{'butt', 'round', 'projecting'}
dash_joinstyle	{'miter', 'round', 'bevel'}

Table 185 – continued from previous page

Property	Description
dashes	sequence of floats (on/off ink in points) or (None, None)
data	(2, N) array or two 1D arrays
drawstyle or ds	{'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'd
figure	*Figure*
fillstyle	{'full', 'left', 'right', 'bottom', 'top', 'none'}
gid	str
in_layout	bool
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float
marker	marker style string, *Path* or *MarkerStyle*
markeredgecolor or mec	color
markeredgewidth or mew	float
markerfacecolor or mfc	color
markerfacecoloralt or mfcalt	color
markersize or ms	float
markevery	None or int or (int, int) or slice or List[int] or float or (float, float)
path_effects	*AbstractPathEffect*
picker	unknown
pickradius	float
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
solid_capstyle	{'butt', 'round', 'projecting'}
solid_joinstyle	{'miter', 'round', 'bevel'}
transform	*matplotlib.transforms.Transform*
url	str
visible	bool
xdata	1D array
ydata	1D array
zorder	float

See also:

axhline

for horizontal lines

axvline

for vertical lines

Examples

Draw a thick red line passing through (0, 0) and (1, 1):

```
>>> axline((0, 0), (1, 1), linewidth=4, color='r')
```

Examples using `matplotlib.pyplot.axline`

- sphx_glr_gallery_pyplots_axline.py

matplotlib.pyplot.axvline

`matplotlib.pyplot.axvline`(*x=0*, *ymin=0*, *ymax=1*, ***kwargs*)

Add a vertical line across the axes.

Parameters

x

[float, default: 0] x position in data coordinates of the vertical line.

ymin

[float, default: 0] Should be between 0 and 1, 0 being the bottom of the plot, 1 the top of the plot.

ymax

[float, default: 1] Should be between 0 and 1, 0 being the bottom of the plot, 1 the top of the plot.

Returns

Line2D

Other Parameters

****kwargs**

Valid keyword arguments are *Line2D* properties, with the exception of 'transform':

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value
alpha	float or None
animated	bool

Table 186 – continued from previous page

Property	Description
antialiased or aa	bool
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color or c	color
contains	unknown
dash_capstyle	{'butt', 'round', 'projecting'}
dash_joinstyle	{'miter', 'round', 'bevel'}
dashes	sequence of floats (on/off ink in points) or (None, None)
data	(2, N) array or two 1D arrays
drawstyle or ds	{'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'd
figure	*Figure*
fillstyle	{'full', 'left', 'right', 'bottom', 'top', 'none'}
gid	str
in_layout	bool
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float
marker	marker style string, *Path* or *MarkerStyle*
markeredgecolor or mec	color
markeredgewidth or mew	float
markerfacecolor or mfc	color
markerfacecoloralt or mfcalt	color
markersize or ms	float
markevery	None or int or (int, int) or slice or List[int] or float or (float, float)
path_effects	*AbstractPathEffect*
picker	unknown
pickradius	float
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
solid_capstyle	{'butt', 'round', 'projecting'}
solid_joinstyle	{'miter', 'round', 'bevel'}
transform	*matplotlib.transforms.Transform*
url	str
visible	bool
xdata	1D array
ydata	1D array
zorder	float

See also:

vlines

Add vertical lines in data coordinates.

axvspan

Add a vertical span (rectangle) across the axis.

axline

Add a line with an arbitrary slope.

Examples

- draw a thick red vline at $x = 0$ that spans the yrange:

```
>>> axvline(linewidth=4, color='r')
```

- draw a default vline at $x = 1$ that spans the yrange:

```
>>> axvline(x=1)
```

- draw a default vline at $x = .5$ that spans the middle half of the yrange:

```
>>> axvline(x=.5, ymin=0.25, ymax=0.75)
```

Examples using `matplotlib.pyplot.axvline`

- sphx_glr_gallery_pyplots_axline.py
- sphx_glr_gallery_color_color_cycle_default.py

matplotlib.pyplot.axvspan

`matplotlib.pyplot.`**`axvspan`**(*xmin*, *xmax*, *ymin=0*, *ymax=1*, ***kwargs*)

Add a vertical span (rectangle) across the axes.

The rectangle spans from *xmin* to *xmax* horizontally, and, by default, the whole y-axis vertically. The y-span can be set using *ymin* (default: 0) and *ymax* (default: 1) which are in axis units; e.g. `ymin = 0.5` always refers to the middle of the y-axis regardless of the limits set by *set_ylim*.

Parameters

xmin

[float] Lower x-coordinate of the span, in data units.

xmax

[float] Upper x-coordinate of the span, in data units.

ymin

[float, default: 0] Lower y-coordinate of the span, in y-axis units (0-1).

ymax

[float, default: 1] Upper y-coordinate of the span, in y-axis units (0-1).

Returns

Polygon

Vertical span (rectangle) from (xmin, ymin) to (xmax, ymax).

Other Parameters

****kwargs**

[*Polygon* properties]

%(Polygon)s

See also:

axhspan

Add a horizontal span across the axes.

Examples

Draw a vertical, green, translucent rectangle from x = 1.25 to x = 1.55 that spans the yrange of the axes.

```
>>> axvspan(1.25, 1.55, facecolor='g', alpha=0.5)
```

Examples using `matplotlib.pyplot.axvspan`

- sphx_glr_gallery_subplots_axes_and_figures_axhspan_demo.py

matplotlib.pyplot.bar

matplotlib.pyplot.**bar**(*x*, *height*, *width=0.8*, *bottom=None*, ***, *align='center'*, *data=None*, ***kwargs*)

Make a bar plot.

The bars are positioned at *x* with the given *align*ment. Their dimensions are given by *height* and *width*. The vertical baseline is *bottom* (default 0).

Many parameters can take either a single value applying to all bars or a sequence of values, one for each bar.

Parameters

x

[float or array-like] The x coordinates of the bars. See also *align* for the alignment of the bars to the coordinates.

height

[float or array-like] The height(s) of the bars.

width

[float or array-like, default: 0.8] The width(s) of the bars.

bottom

[float or array-like, default: 0] The y coordinate(s) of the bars bases.

align

[{'center', 'edge'}, default: 'center'] Alignment of the bars to the *x* coordinates:

- 'center': Center the base on the *x* positions.
- 'edge': Align the left edges of the bars with the *x* positions.

To align the bars on the right edge pass a negative *width* and `align='edge'`.

Returns

BarContainer

Container with all the bars and optionally errorbars.

Other Parameters

color

[color or list of color, optional] The colors of the bar faces.

edgecolor

[color or list of color, optional] The colors of the bar edges.

linewidth

[float or array-like, optional] Width of the bar edge(s). If 0, don't draw edges.

tick_label

[str or list of str, optional] The tick labels of the bars. Default: None (Use default numeric labels.)

xerr, yerr

[float or array-like of shape(N,) or shape(2, N), optional] If not *None*, add horizontal / vertical errorbars to the bar tips. The values are +/- sizes relative to the data:

- scalar: symmetric +/- values for all bars
- shape(N,): symmetric +/- values for each bar
- shape(2, N): Separate - and + values for each bar. First row contains the lower errors, the second row contains the upper errors.
- *None*: No errorbar. (Default)

See /gallery/statistics/errorbar_features for an example on the usage of `xerr` and `yerr`.

ecolor

[color or list of color, default: 'black'] The line color of the errorbars.

capsize

[float, default: `rcParams["errorbar.capsize"]` (default: `0.0`)] The length of the error bar caps in points.

error_kw

[dict, optional] Dictionary of kwargs to be passed to the *errorbar* method. Values of *ecolor* or *capsize* defined here take precedence over the independent kwargs.

log

[bool, default: False] If *True*, set the y-axis to be log scale.

****kwargs**

[*Rectangle* properties]

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and retur
alpha	float or None
animated	bool
antialiased or aa	unknown
capstyle	{'butt', 'round', 'projecting'}
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color	color
contains	unknown
edgecolor or ec	color or None or 'auto'
facecolor or fc	color or None
figure	*Figure*
fill	bool
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float or None
path_effects	*AbstractPathEffect*
picker	None or bool or callable
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
visible	bool
zorder	float

See also:

barh

Plot a horizontal bar plot.

Notes

Stacked bars can be achieved by passing individual *bottom* values per bar. See /gallery/lines_bars_and_markers/bar_stacked.

Note: In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, every other argument can also be string `s`, which is interpreted as `data[s]` (unless this raises an exception).

Objects passed as **data** must support item access (`data[s]`) and membership test (`s in data`).

Examples using `matplotlib.pyplot.bar`

- sphx_glr_gallery_lines_bars_and_markers_barchart.py
- sphx_glr_gallery_statistics_barchart_demo.py
- sphx_glr_gallery_pie_and_polar_charts_nested_pie.py
- sphx_glr_gallery_pie_and_polar_charts_polar_bar.py
- sphx_glr_gallery_shapes_and_collections_hatch_demo.py
- sphx_glr_gallery_misc_table_demo.py
- *Pyplot tutorial*
- *Sample plots in Matplotlib*

matplotlib.pyplot.barbs

matplotlib.pyplot.**barbs**(**args*, *data=None*, ***kw*)

Plot a 2D field of barbs.

Call signature:

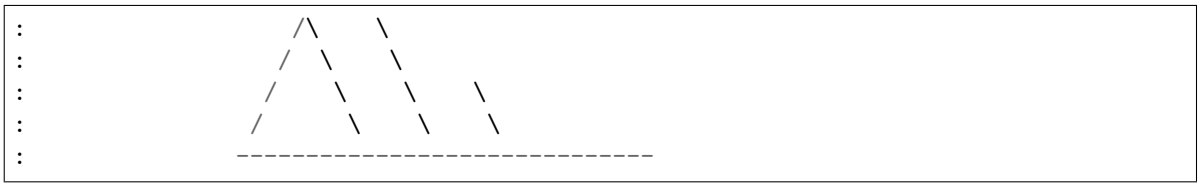
```
barbs([X, Y], U, V, [C], **kw)
```

Where X, Y define the barb locations, U, V define the barb directions, and C optionally sets the color.

All arguments may be 1D or 2D. U, V, C may be masked arrays, but masked X, Y are not supported at present.

Barbs are traditionally used in meteorology as a way to plot the speed and direction of wind observations, but can technically be used to plot any two dimensional vector quantity. As opposed to arrows, which give vector magnitude by

the length of the arrow, the barbs give more quantitative information about the vector magnitude by putting slanted lines or a triangle for various increments in magnitude, as show schematically below:



The largest increment is given by a triangle (or "flag"). After those come full lines (barbs). The smallest increment is a half line. There is only, of course, ever at most 1 half line. If the magnitude is small and only needs a single half-line and no full lines or triangles, the half-line is offset from the end of the barb so that it can be easily distinguished from barbs with a single full line. The magnitude for the barb shown above would nominally be 65, using the standard increments of 50, 10, and 5.

See also https://en.wikipedia.org/wiki/Wind_barb.

Parameters

X, Y

[1D or 2D array-like, optional] The x and y coordinates of the barb locations. See *pivot* for how the barbs are drawn to the x, y positions.

If not given, they will be generated as a uniform integer meshgrid based on the dimensions of *U* and *V*.

If *X* and *Y* are 1D but *U*, *V* are 2D, *X*, *Y* are expanded to 2D using `X, Y = np.meshgrid(X, Y)`. In this case `len(X)` and `len(Y)` must match the column and row dimensions of *U* and *V*.

U, V

[1D or 2D array-like] The x and y components of the barb shaft.

C

[1D or 2D array-like, optional] Numeric data that defines the barb colors by colormapping via *norm* and *cmap*.

This does not support explicit colors. If you want to set colors directly, use *barbcolor* instead.

length

[float, default: 7] Length of the barb in points; the other parts of the barb are scaled against this.

pivot

[{'tip', 'middle'} or float, default: 'tip'] The part of the arrow that is anchored to the *X*, *Y* grid. The barb rotates about this point. This can also be a number, which shifts the start of the barb that many points away from grid point.

barbcolor

[color or color sequence] The color of all parts of the barb except for the flags. This parameter is analogous to the *edgecolor* parameter for polygons, which can be used instead. However this parameter will override facecolor.

flagcolor

[color or color sequence] The color of any flags on the barb. This parameter is analogous to the *facecolor* parameter for polygons, which can be used instead. However, this parameter will override facecolor. If this is not set (and *C* has not either) then *flagcolor* will be set to match *barbcolor* so that the barb has a uniform color. If *C* has been set, *flagcolor* has no effect.

sizes

[dict, optional] A dictionary of coefficients specifying the ratio of a given feature to the length of the barb. Only those values one wishes to override need to be included. These features include:

- 'spacing' - space between features (flags, full/half barbs)
- 'height' - height (distance from shaft to top) of a flag or full barb
- 'width' - width of a flag, twice the width of a full barb
- 'emptybarb' - radius of the circle used for low magnitudes

fill_empty

[bool, default: False] Whether the empty barbs (circles) that are drawn should be filled with the flag color. If they are not filled, the center is transparent.

rounding

[bool, default: True] Whether the vector magnitude should be rounded when allocating barb components. If True, the magnitude is rounded to the nearest multiple of the half-barb increment. If False, the magnitude is simply truncated to the next lowest multiple.

barb_increments

[dict, optional] A dictionary of increments specifying values to associate with different parts of the barb. Only those values one wishes to override need to be included.

- 'half' - half barbs (Default is 5)
- 'full' - full barbs (Default is 10)
- 'flag' - flags (default is 50)

flip_barb

[bool or array-like of bool, default: False] Whether the lines and flags should point opposite to normal. Normal behavior is for the barbs and lines to point right (comes from wind barbs having these features point towards low pressure in the Northern Hemisphere).

A single value is applied to all barbs. Individual barbs can be flipped by passing a bool array of the same size as *U* and *V*.

Returns

barbs

[*Barbs*]

Other Parameters

****kwargs**

The barbs can further be customized using *PolyCollection* keyword arguments:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a
alpha	float or None
animated	bool
antialiased or aa or antialiaseds	bool or list of bools
array	ndarray
capstyle	{'butt', 'round', 'projecting'}
clim	(vmin: float, vmax: float)
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
cmap	*Colormap* or str or None
color	color or list of rgba tuples
contains	unknown
edgecolor or ec or edgecolors	color or list of colors or 'face'
facecolor or facecolors or fc	color or list of colors
figure	*Figure*
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}

Table 188 – continued from previous page

Property	Description
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or dashes or linestyles or ls	str or tuple or list thereof
linewidth or linewidths or lw	float or list of floats
norm	*Normalize* or None
offset_position	unknown
offsets	array-like (N, 2) or (2,)
path_effects	*AbstractPathEffect*
picker	None or bool or callable
pickradius	unknown
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
urls	list of str or None
visible	bool
zorder	float

Notes

Note: In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, every other argument can also be string `s`, which is interpreted as `data[s]` (unless this raises an exception).

Objects passed as **data** must support item access (`data[s]`) and membership test (`s in data`).

Examples using `matplotlib.pyplot.barbs`

- sphx_glr_gallery_images_contours_and_fields_barb_demo.py

matplotlib.pyplot.barh

matplotlib.pyplot.**barh**(*y*, *width*, *height=0.8*, *left=None*, *, *align='center'*, ***kwargs*)

Make a horizontal bar plot.

The bars are positioned at *y* with the given *align*ment. Their dimensions are given by *width* and *height*. The horizontal baseline is *left* (default 0).

Many parameters can take either a single value applying to all bars or a sequence of values, one for each bar.

Parameters

y

[float or array-like] The y coordinates of the bars. See also *align* for the alignment of the bars to the coordinates.

width

[float or array-like] The width(s) of the bars.

height

[float or array-like, default: 0.8] The heights of the bars.

left

[float or array-like, default: 0] The x coordinates of the left sides of the bars.

align

[{'center', 'edge'}, default: 'center'] Alignment of the base to the *y* coordinates*:

- 'center': Center the bars on the *y* positions.
- 'edge': Align the bottom edges of the bars with the *y* positions.

To align the bars on the top edge pass a negative *height* and `align='edge'`.

Returns

BarContainer

Container with all the bars and optionally errorbars.

Other Parameters

color

[color or list of color, optional] The colors of the bar faces.

edgecolor

[color or list of color, optional] The colors of the bar edges.

linewidth

[float or array-like, optional] Width of the bar edge(s). If 0, don't draw edges.

tick_label

[str or list of str, optional] The tick labels of the bars. Default: None (Use default numeric labels.)

xerr, yerr

[float or array-like of shape(N,) or shape(2, N), optional] If not `None`, add horizontal / vertical errorbars to the bar tips. The values are +/- sizes relative to the data:

- scalar: symmetric +/- values for all bars
- shape(N,): symmetric +/- values for each bar
- shape(2, N): Separate - and + values for each bar. First row contains the lower errors, the second row contains the upper errors.
- *None*: No errorbar. (default)

See /gallery/statistics/errorbar_features for an example on the usage of `xerr` and `yerr`.

ecolor

[color or list of color, default: 'black'] The line color of the errorbars.

capsize

[float, default: `rcParams["errorbar.capsize"]` (default: `0.0`)] The length of the error bar caps in points.

error_kw

[dict, optional] Dictionary of kwargs to be passed to the *errorbar* method. Values of *ecolor* or *capsize* defined here take precedence over the independent kwargs.

log

[bool, default: False] If `True`, set the x-axis to be log scale.

****kwargs**

[*Rectangle* properties]

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and retur
alpha	float or None
animated	bool
antialiased or aa	unknown
capstyle	{'butt', 'round', 'projecting'}
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color	color
contains	unknown
edgecolor or ec	color or None or 'auto'
facecolor or fc	color or None
figure	*Figure*
fill	bool
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float or None
path_effects	*AbstractPathEffect*
picker	None or bool or callable
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
visible	bool
zorder	float

See also:

bar

Plot a vertical bar plot.

Notes

Stacked bars can be achieved by passing individual *left* values per bar. See /gallery/lines_bars_and_markers/horizontal_barchart_distribution .

Examples using `matplotlib.pyplot.barh`

- sphx_glr_gallery_lines_bars_and_markers_horizontal_barchart_distribution.py

matplotlib.pyplot.bone

`matplotlib.pyplot.bone()`
Set the colormap to "bone".

This changes the default colormap as well as the colormap of the current image if there is one. See `help(colormaps)` for more information.

Examples using `matplotlib.pyplot.bone`

matplotlib.pyplot.box

`matplotlib.pyplot.box(`*on=None*`)`
Turn the axes box on or off on the current axes.

Parameters

on

[bool or None] The new *Axes* box state. If `None`, toggle the state.

See also:

matplotlib.axes.Axes.set_frame_on()

matplotlib.axes.Axes.get_frame_on()

Examples using `matplotlib.pyplot.box`

matplotlib.pyplot.boxplot

matplotlib.pyplot.**boxplot**(*x*, *notch=None*, *sym=None*, *vert=None*, *whis=None*, *positions=None*, *widths=None*, *patch_artist=None*, *bootstrap=None*, *usermedians=None*, *conf_intervals=None*, *meanline=None*, *showmeans=None*, *showcaps=None*, *showbox=None*, *showfliers=None*, *boxprops=None*, *labels=None*, *flierprops=None*, *medianprops=None*, *meanprops=None*, *capprops=None*, *whiskerprops=None*, *manage_ticks=True*, *autorange=False*, *zorder=None*, ***, *data=None*)

Make a box and whisker plot.

Make a box and whisker plot for each column of *x* or each vector in sequence *x*. The box extends from the lower to upper quartile values of the data, with a line at the median. The whiskers extend from the box to show the range of the data. Flier points are those past the end of the whiskers.

Parameters

x

[Array or a sequence of vectors.] The input data.

notch

[bool, default: False] Whether to draw a noteched box plot (`True`), or a rectangular box plot (`False`). The notches represent the confidence interval (CI) around the median. The documentation for *bootstrap* describes how the locations of the notches are computed.

Note: In cases where the values of the CI are less than the lower quartile or greater than the upper quartile, the notches will extend beyond the box, giving it a distinctive "flipped" appearance. This is expected behavior and consistent with other statistical visualization packages.

---

sym

[str, optional] The default symbol for flier points. An empty string ('') hides the fliers. If `None`, then the fliers default to 'b+'. More control is provided by the *flierprops* parameter.

vert

[bool, default: True] If `True`, draws vertical boxes. If `False`, draw horizontal boxes.

whis

[float or (float, float), default: 1.5] The position of the whiskers.

If a float, the lower whisker is at the lowest datum above `Q1 - whis*(Q3-Q1)`, and the upper whisker at the highest datum below `Q3 + whis*(Q3-Q1)`, where Q1 and Q3 are the first and third quartiles. The default value of `whis = 1.5` corresponds to Tukey's original definition of boxplots.

If a pair of floats, they indicate the percentiles at which to draw the whiskers (e.g., (5, 95)). In particular, setting this to (0, 100) results in whiskers covering the whole range of the data. "range" is a deprecated synonym for (0, 100).

In the edge case where `Q1 == Q3`, *whis* is automatically set to (0, 100) (cover the whole range of the data) if *autorange* is True.

Beyond the whiskers, data are considered outliers and are plotted as individual points.

bootstrap

[int, optional] Specifies whether to bootstrap the confidence intervals around the median for notched boxplots. If *bootstrap* is None, no bootstrapping is performed, and notches are calculated using a Gaussian-based asymptotic approximation (see McGill, R., Tukey, J.W., and Larsen, W.A., 1978, and Kendall and Stuart, 1967). Otherwise, bootstrap specifies the number of times to bootstrap the median to determine its 95% confidence intervals. Values between 1000 and 10000 are recommended.

usermedians

[array-like, optional] A 1D array-like of length `len(x)`. Each entry that is not `None` forces the value of the median for the corresponding dataset. For entries that are `None`, the medians are computed by Matplotlib as normal.

conf_intervals

[array-like, optional] A 2D array-like of shape `(len(x), 2)`. Each entry that is not None forces the location of the corresponding notch (which is only drawn if *notch* is `True`). For entries that are `None`, the notches are computed by the method specified by the other parameters (e.g., *bootstrap*).

positions

[array-like, optional] Sets the positions of the boxes. The ticks and limits are automatically set to match the positions. Defaults to `range(1, N+1)` where N is the number of boxes to be drawn.

widths

[float or array-like] Sets the width of each box either with a scalar or a sequence. The default is 0.5, or `0.15*(distance between`

`extreme positions`), if that is smaller.

patch_artist

[bool, default: False] If `False` produces boxes with the Line2D artist. Otherwise, boxes and drawn with Patch artists.

labels

[sequence, optional] Labels for each dataset (one per dataset).

manage_ticks

[bool, default: True] If True, the tick locations and labels will be adjusted to match the boxplot positions.

autorange

[bool, default: False] When `True` and the data are distributed such that the 25th and 75th percentiles are equal, *whis* is set to (0, 100) such that the whisker ends are at the minimum and maximum of the data.

meanline

[bool, default: False] If `True` (and *showmeans* is `True`), will try to render the mean as a line spanning the full width of the box according to *meanprops* (see below). Not recommended if *shownotches* is also True. Otherwise, means will be shown as points.

zorder

[float, default: `Line2D.zorder = 2`] Sets the zorder of the boxplot.

Returns

dict

A dictionary mapping each component of the boxplot to a list of the *Line2D* instances created. That dictionary has the following keys (assuming vertical boxplots):

- `boxes`: the main body of the boxplot showing the quartiles and the median's confidence intervals if enabled.
- `medians`: horizontal lines at the median of each box.
- `whiskers`: the vertical lines extending to the most extreme, non-outlier data points.
- `caps`: the horizontal lines at the ends of the whiskers.
- `fliers`: points representing data that extend beyond the whiskers (fliers).
- `means`: points or lines representing the means.

Other Parameters

showcaps

[bool, default: True] Show the caps on the ends of whiskers.

showbox

[bool, default: True] Show the central box.

showfliers

[bool, default: True] Show the outliers beyond the caps.

showmeans

[bool, default: False] Show the arithmetic means.

capprops

[dict, default: None] The style of the caps.

boxprops

[dict, default: None] The style of the box.

whiskerprops

[dict, default: None] The style of the whiskers.

flierprops

[dict, default: None] The style of the fliers.

medianprops

[dict, default: None] The style of the median.

meanprops

[dict, default: None] The style of the mean.

Notes

Note: In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, every other argument can also be string `s`, which is interpreted as `data[s]` (unless this raises an exception).

Objects passed as **data** must support item access (`data[s]`) and membership test (`s in data`).

Examples using `matplotlib.pyplot.boxplot`

- sphx_glr_gallery_pyplots_boxplot_demo_pyplot.py

matplotlib.pyplot.broken_barh

matplotlib.pyplot.**broken_barh**(*xranges*, *yrange*, *, *data=None*, ***kwargs*)

Plot a horizontal sequence of rectangles.

A rectangle is drawn for each element of *xranges*. All rectangles have the same vertical position and size defined by *yrange*.

This is a convenience function for instantiating a *BrokenBarHCollection*, adding it to the axes and autoscaling the view.

Parameters

xranges

[sequence of tuples (*xmin*, *xwidth*)] The x-positions and extends of the rectangles. For each tuple (*xmin*, *xwidth*) a rectangle is drawn from *xmin* to *xmin* + *xwidth*.

yrange

[(*ymin*, *yheight*)] The y-position and extend for all the rectangles.

Returns

BrokenBarHCollection

Other Parameters

****kwargs**

[*BrokenBarHCollection* properties] Each *kwarg* can be either a single argument applying to all rectangles, e.g.:

```
facecolors='black'
```

or a sequence of arguments over which is cycled, e.g.:

```
facecolors=('black', 'blue')
```

would create interleaving black and blue rectangles.

Supported keywords:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a
alpha	float or None
animated	bool
antialiased or aa or antialiaseds	bool or list of bools
array	ndarray
capstyle	{'butt', 'round', 'projecting'}
clim	(vmin: float, vmax: float)
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
cmap	*Colormap* or str or None
color	color or list of rgba tuples
contains	unknown
edgecolor or ec or edgecolors	color or list of colors or 'face'
facecolor or facecolors or fc	color or list of colors
figure	*Figure*
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or dashes or linestyles or ls	str or tuple or list thereof
linewidth or linewidths or lw	float or list of floats
norm	*Normalize* or None
offset_position	unknown
offsets	array-like (N, 2) or (2,)
path_effects	*AbstractPathEffect*
picker	None or bool or callable
pickradius	unknown
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
urls	list of str or None
visible	bool
zorder	float

Notes

Note: In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, every other argument can also be string `s`, which is interpreted as `data[s]` (unless this raises an exception).

Objects passed as **data** must support item access (`data[s]`) and membership test (`s in data`).

Examples using `matplotlib.pyplot.broken_barh`

matplotlib.pyplot.cla

`matplotlib.pyplot.cla()`
Clear the current axes.

Examples using `matplotlib.pyplot.cla`

- *Pyplot tutorial*

matplotlib.pyplot.clabel

`matplotlib.pyplot.clabel(`*CS*, *levels=None*, ***kwargs*`)`
Label a contour plot.

Adds labels to line contours in given *ContourSet*.

Parameters

CS

[*ContourSet* instance] Line contours to label.

levels

[array-like, optional] A list of level values, that should be labeled. The list must be a subset of `CS.levels`. If not given, all levels are labeled.

****kwargs**

All other parameters are documented in *clabel*.

Examples using `matplotlib.pyplot.clabel`

- sphx_glr_gallery_event_handling_ginput_manual_clabel_sgskip.py

matplotlib.pyplot.clf

matplotlib.pyplot.**clf**()
Clear the current figure.

Examples using `matplotlib.pyplot.clf`

- sphx_glr_gallery_event_handling_ginput_manual_clabel_sgskip.py

matplotlib.pyplot.clim

matplotlib.pyplot.**clim**(*vmin=None*, *vmax=None*)
Set the color limits of the current image.

If either *vmin* or *vmax* is None, the image min/max respectively will be used for color scaling.

If you want to set the clim of multiple images, use *set_clim* on every image, for example:

```
for im in gca().get_images():
    im.set_clim(0, 0.5)
```

Examples using `matplotlib.pyplot.clim`

matplotlib.pyplot.close

matplotlib.pyplot.**close**(*fig=None*)
Close a figure window.

Parameters

fig

[None or int or str or *Figure*] The figure to close. There are a number of ways to specify this:

- *None*: the current figure
- *Figure*: the given *Figure* instance
- `int`: a figure number

- `str`: a figure name
- 'all': all figures

Examples using `matplotlib.pyplot.close`

- sphx_glr_gallery_misc_multiprocess_sgskip.py

matplotlib.pyplot.cohere

`matplotlib.pyplot.cohere`(*x*, *y*, *NFFT=256*, *Fs=2*, *Fc=0*, *detrend=<function detrend_none at 0x7f5434697820>*, *window=<function window_hanning at 0x7f5434697160>*, *noverlap=0*, *pad_to=None*, *sides='default'*, *scale_by_freq=None*, ***, *data=None*, ***kwargs*)

Plot the coherence between *x* and *y*.

Plot the coherence between *x* and *y*. Coherence is the normalized cross spectral density:

$$C_{xy} = \frac{|P_{xy}|^2}{P_{xx}P_{yy}}$$

Parameters

Fs

[float, default: 2] The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, *freqs*, in cycles per time unit.

window

[callable or ndarray, default: *window_hanning*] A function or a vector of length *NFFT*. To create window vectors see *window_hanning*, *window_none*, `numpy.blackman`, `numpy.hamming`, `numpy.bartlett`, `scipy.signal`, `scipy.signal.get_window`, etc. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

sides

[{'default', 'onesided', 'twosided'}, optional] Which sides of the spectrum to return. 'default' is one-sided for real data and two-sided for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

pad_to

[int, optional] The number of points to which the data segment is padded when performing the FFT. This can be different from

NFFT, which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to fft(). The default is None, which sets *pad_to* equal to *NFFT*

NFFT

[int, default: 256] The number of data points used in each block for the FFT. A power 2 is most efficient. This should *NOT* be used to get zero padding, or the scaling of the result will be incorrect; use *pad_to* for this instead.

detrend

[{'none', 'mean', 'linear'} or callable, default 'none'] The function applied to each segment before fft-ing, designed to remove the mean or linear trend. Unlike in MATLAB, where the *detrend* parameter is a vector, in Matplotlib is it a function. The *mlab* module defines *detrend_none*, *detrend_mean*, and *detrend_linear*, but you can use a custom function as well. You can also use a string to choose one of the functions: 'none' calls *detrend_none*. 'mean' calls *detrend_mean*. 'linear' calls *detrend_linear*.

scale_by_freq

[bool, default: True] Whether the resulting density values should be scaled by the scaling frequency, which gives density in units of Hz^-1. This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

noverlap

[int, default: 0 (no overlap)] The number of points of overlap between blocks.

Fc

[int, default: 0] The center frequency of *x*, which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

Returns

Cxy

[1-D array] The coherence vector.

freqs

[1-D array] The frequencies for the elements in *Cxy*.

Other Parameters

**kwargs

Keyword arguments control the *Line2D* properties:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value
alpha	float or None
animated	bool
antialiased or aa	bool
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color or c	color
contains	unknown
dash_capstyle	{'butt', 'round', 'projecting'}
dash_joinstyle	{'miter', 'round', 'bevel'}
dashes	sequence of floats (on/off ink in points) or (None, None)
data	(2, N) array or two 1D arrays
drawstyle or ds	{'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'd
figure	*Figure*
fillstyle	{'full', 'left', 'right', 'bottom', 'top', 'none'}
gid	str
in_layout	bool
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float
marker	marker style string, *Path* or *MarkerStyle*
markeredgecolor or mec	color
markeredgewidth or mew	float
markerfacecolor or mfc	color
markerfacecoloralt or mfcalt	color
markersize or ms	float
markevery	None or int or (int, int) or slice or List[int] or float or (float, float)
path_effects	*AbstractPathEffect*
picker	unknown
pickradius	float
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
solid_capstyle	{'butt', 'round', 'projecting'}
solid_joinstyle	{'miter', 'round', 'bevel'}
transform	*matplotlib.transforms.Transform*
url	str
visible	bool

Table 191 – continued from previous page

Property	Description
xdata	1D array
ydata	1D array
zorder	float

Notes

Note: In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, the following arguments can also be string `s`, which is interpreted as `data[s]` (unless this raises an exception): *x*, *y*.

Objects passed as **data** must support item access (`data[s]`) and membership test (`s in data`).

References

Bendat & Piersol -- Random Data: Analysis and Measurement Procedures, John Wiley & Sons (1986)

Examples using `matplotlib.pyplot.cohere`

matplotlib.pyplot.colorbar

matplotlib.pyplot.**colorbar**(*mappable=None*, *cax=None*, *ax=None*, ***kw*)

Add a colorbar to a plot.

Function signatures for the *pyplot* interface; all but the first are also method signatures for the *colorbar* method:

```
colorbar(**kwargs)
colorbar(mappable, **kwargs)
colorbar(mappable, cax=cax, **kwargs)
colorbar(mappable, ax=ax, **kwargs)
```

Parameters

mappable

The *matplotlib.cm.ScalarMappable* (i.e., *AxesImage*, *ContourSet*, etc.) described by this colorbar. This argument is mandatory for the *Figure.colorbar* method but optional for the *pyplot.colorbar* function, which sets the default to the current image.

Note that one can create a *ScalarMappable* "on-the-fly" to generate colorbars not attached to a previously drawn artist, e.g.

```
fig.colorbar(cm.ScalarMappable(norm=norm, cmap=cmap), ax=ax)
```

cax

[*Axes*, optional] Axes into which the colorbar will be drawn.

ax

[*Axes*, list of Axes, optional] Parent axes from which space for a new colorbar axes will be stolen. If a list of axes is given they will all be resized to make room for the colorbar axes.

use_gridspec

[bool, optional] If *cax* is `None`, a new *cax* is created as an instance of Axes. If *ax* is an instance of Subplot and *use_gridspec* is `True`, *cax* is created as an instance of Subplot using the *gridspec* module.

Returns

colorbar

[*Colorbar*] See also its base class, *ColorbarBase*.

Notes

Additional keyword arguments are of two kinds:

axes properties:

fraction

[float, default: 0.15] Fraction of original axes to use for colorbar.

shrink

[float, default: 1.0] Fraction by which to multiply the size of the colorbar.

aspect

[float, default: 20] Ratio of long to short dimensions.

pad

[float, default: 0.05 if vertical, 0.15 if horizontal] Fraction of original axes between colorbar and new image axes.

anchor

[(float, float), optional] The anchor point of the colorbar axes. Defaults to (0.0, 0.5) if vertical; (0.5, 1.0) if horizontal.

panchor

[(float, float), or *False*, optional] The anchor point of the colorbar parent axes. If *False*, the parent axes' anchor will be unchanged. Defaults to (1.0, 0.5) if vertical; (0.5, 0.0) if horizontal.

colorbar properties:

Property	Description
extend	{'neither', 'both', 'min', 'max'} If not 'neither', make pointed end(s) for out-of- range values. These are set for a given colormap using the colormap set_under and set_over methods.
extendfrac	{*None*, 'auto', length, lengths} If set to *None*, both the minimum and maximum triangular colorbar extensions with have a length of 5% of the interior colorbar length (this is the default setting). If set to 'auto', makes the triangular colorbar extensions the same lengths as the interior boxes (when *spacing* is set to 'uniform') or the same lengths as the respective adjacent interior boxes (when *spacing* is set to 'proportional'). If a scalar, indicates the length of both the minimum and maximum triangular colorbar extensions as a fraction of the interior colorbar length. A two-element sequence of fractions may also be given, indicating the lengths of the minimum and maximum colorbar extensions respectively as a fraction of the interior colorbar length.
extendrect	bool If *False* the minimum and maximum colorbar extensions will be triangular (the default). If *True* the extensions will be rectangular.
spacing	{'uniform', 'proportional'} Uniform spacing gives each discrete color the same space; proportional makes the space proportional to the data interval.
ticks	*None* or list of ticks or Locator If None, ticks are determined automatically from the input.
format	None or str or Formatter If None, *ScalarFormatter* is used. If a format string is given, e.g., '%.3f', that is used. An alternative *Formatter* may be given instead.
drawedges	bool Whether to draw lines at color boundaries.
label	str The label on the colorbar's long axis.

The following will probably be useful only in the context of indexed colors (that is, when the mappable has norm=NoNorm()), or other unusual circumstances.

Property	Description
boundaries	None or a sequence
values	None or a sequence which must be of length 1 less than the sequence of *boundaries*. For each region delimited by adjacent entries in *boundaries*, the color mapped to the corresponding value in values will be used.

If *mappable* is a *ContourSet*, its *extend* kwarg is included automatically.

The *shrink* kwarg provides a simple way to scale the colorbar with respect to the axes. Note that if *cax* is specified, it determines the size of the colorbar and *shrink* and *aspect* kwargs are ignored.

For more precise control, you can manually specify the positions of the axes objects in which the mappable and the colorbar are drawn. In this case, do not use any of the axes properties kwargs.

It is known that some vector graphics viewers (svg and pdf) renders white gaps between segments of the colorbar. This is due to bugs in the viewers, not Matplotlib. As a workaround, the colorbar can be rendered with overlapping segments:

```
cbar = colorbar()
cbar.solids.set_edgecolor("face")
draw()
```

However this has negative consequences in other circumstances, e.g. with semi-transparent images (alpha < 1) and colorbar extensions; therefore, this workaround is not used by default (see issue #1188).

Examples using `matplotlib.pyplot.colorbar`

- sphx_glr_gallery_images_contours_and_fields_contour_demo.py
- sphx_glr_gallery_images_contours_and_fields_contour_image.py
- sphx_glr_gallery_images_contours_and_fields_contourf_demo.py
- sphx_glr_gallery_images_contours_and_fields_contourf_hatching.py
- sphx_glr_gallery_images_contours_and_fields_contourf_log.py
- sphx_glr_gallery_images_contours_and_fields_image_annotated_heatmap.py
- sphx_glr_gallery_images_contours_and_fields_image_masked.py

- sphx_glr_gallery_images_contours_and_fields_multi_image.py
- sphx_glr_gallery_images_contours_and_fields_pcolor_demo.py
- sphx_glr_gallery_images_contours_and_fields_pcolormesh_levels.py
- sphx_glr_gallery_subplots_axes_and_figures_subplots_adjust.py
- sphx_glr_gallery_color_colorbar_basics.py
- sphx_glr_gallery_color_custom_cmap.py
- sphx_glr_gallery_shapes_and_collections_ellipse_collection.py
- sphx_glr_gallery_shapes_and_collections_line_collection.py
- sphx_glr_gallery_axes_grid1_demo_axes_divider.py
- sphx_glr_gallery_axes_grid1_simple_colorbar.py
- *Image tutorial*
- *Tight Layout guide*

matplotlib.pyplot.connect

matplotlib.pyplot.**connect**(*s*, *func*)

Bind function *func* to event *s*.

Parameters

s

[str] One of the following events ids:

- 'button_press_event'
- 'button_release_event'
- 'draw_event'
- 'key_press_event'
- 'key_release_event'
- 'motion_notify_event'
- 'pick_event'
- 'resize_event'
- 'scroll_event'
- 'figure_enter_event',
- 'figure_leave_event',
- 'axes_enter_event',

- 'axes_leave_event'
- 'close_event'.

func

[callable] The callback function to be executed, which must have the signature:

```
def func(event: Event) -> Any
```

For the location events (button and key press/release), if the mouse is over the axes, the `inaxes` attribute of the event will be set to the *Axes* the event occurs is over, and additionally, the variables `xdata` and `ydata` attributes will be set to the mouse location in data coordinates. See *KeyEvent* and *MouseEvent* for more info.

Returns

cid

A connection id that can be used with *FigureCanvasBase.mpl_disconnect*.

Examples

```
def on_press(event):
    print('you pressed', event.button, event.xdata, event.ydata)

cid = canvas.mpl_connect('button_press_event', on_press)
```

Examples using `matplotlib.pyplot.connect`

- sphx_glr_gallery_event_handling_coords_demo.py

matplotlib.pyplot.contour

matplotlib.pyplot.**contour**(**args*, *data=None*, ***kwargs*)

Plot contours.

Call signature:

```
contour([X, Y,] Z, [levels], **kwargs)
```

contour and *contourf* draw contour lines and filled contours, respectively. Except as noted, function signatures and return values are the same for both versions.

Parameters

X, Y

[array-like, optional] The coordinates of the values in *Z*.

X and *Y* must both be 2-D with the same shape as *Z* (e.g. created via `numpy.meshgrid`), or they must both be 1-D such that `len(X) == M` is the number of columns in *Z* and `len(Y) == N` is the number of rows in *Z*.

If not given, they are assumed to be integer indices, i.e. `X = range(M)`, `Y = range(N)`.

Z

[array-like(N, M)] The height values over which the contour is drawn.

levels

[int or array-like, optional] Determines the number and positions of the contour lines / regions.

If an int *n*, use *MaxNLocator*, which tries to automatically choose no more than *n+1* "nice" contour levels between *vmin* and *vmax*.

If array-like, draw contour lines at the specified levels. The values must be in increasing order.

Returns

QuadContourSet

Other Parameters

corner_mask

[bool, default: `rcParams["contour.corner_mask"]` (default: `True`)] Enable/disable corner masking, which only has an effect if *Z* is a masked array. If `False`, any quad touching a masked point is masked out. If `True`, only the triangular corners of quads nearest those points are always masked out, other triangular corners comprising three unmasked points are contoured as usual.

colors

[color string or sequence of colors, optional] The colors of the levels, i.e. the lines for *contour* and the areas for *contourf*.

The sequence is cycled for the levels in ascending order. If the sequence is shorter than the number of levels, it's repeated.

As a shortcut, single color strings may be used in place of one-element lists, i.e. `'red'` instead of `['red']` to color all levels with the same color. This shortcut does only work for color strings, not for other ways of specifying colors.

By default (value *None*), the colormap specified by *cmap* will be used.

alpha

[float, default: 1] The alpha blending value, between 0 (transparent) and 1 (opaque).

cmap

[str or *Colormap*, default: `rcParams["image.cmap"]` (default: `'viridis'`)] A *Colormap* instance or registered colormap name. The colormap maps the level values to colors.

If both *colors* and *cmap* are given, an error is raised.

norm

[*Normalize*, optional] If a colormap is used, the *Normalize* instance scales the level values to the canonical colormap range [0, 1] for mapping to colors. If not given, the default linear scaling is used.

vmin, vmax

[float, optional] If not *None*, either or both of these values will be supplied to the *Normalize* instance, overriding the default color scaling based on *levels*.

origin

[{*None*, 'upper', 'lower', 'image'}, default: None] Determines the orientation and exact position of *Z* by specifying the position of `Z[0, 0]`. This is only relevant, if *X*, *Y* are not given.

- *None*: `Z[0, 0]` is at X=0, Y=0 in the lower left corner.
- 'lower': `Z[0, 0]` is at X=0.5, Y=0.5 in the lower left corner.
- 'upper': `Z[0, 0]` is at X=N+0.5, Y=0.5 in the upper left corner.
- 'image': Use the value from `rcParams["image.origin"]` (default: `'upper'`).

extent

[(x0, x1, y0, y1), optional] If *origin* is not *None*, then *extent* is interpreted as in *imshow*: it gives the outer pixel boundaries. In this case, the position of Z[0, 0] is the center of the pixel, not a corner. If *origin* is *None*, then (*x0*, *y0*) is the position of Z[0, 0], and (*x1*, *y1*) is the position of Z[-1, -1].

This argument is ignored if *X* and *Y* are specified in the call to contour.

locator

[ticker.Locator subclass, optional] The locator is used to determine the contour levels if they are not given explicitly via *levels*. Defaults to *MaxNLocator*.

extend

[{'neither', 'both', 'min', 'max'}, default: 'neither'] Determines the `contourf`-coloring of values that are outside the *levels* range.

If 'neither', values outside the *levels* range are not colored. If 'min', 'max' or 'both', color the values below, above or below and above the *levels* range.

Values below `min(levels)` and above `max(levels)` are mapped to the under/over values of the *Colormap*. Note that most colormaps do not have dedicated colors for these by default, so that the over and under values are the edge values of the colormap. You may want to set these values explicitly using *Colormap.set_under* and *Colormap.set_over*.

Note: An existing *QuadContourSet* does not get notified if properties of its colormap are changed. Therefore, an explicit call `QuadContourSet.changed()` is needed after modifying the colormap. The explicit call can be left out, if a colorbar is assigned to the *QuadContourSet* because it internally calls `QuadContourSet.changed()`.

Example:

```
x = np.arange(1, 10)
y = x.reshape(-1, 1)
h = x * y

cs = plt.contourf(h, levels=[10, 30, 50],
    colors=['#808080', '#A0A0A0', '#C0C0C0'], extend='both')
cs.cmap.set_over('red')
cs.cmap.set_under('blue')
cs.changed()
```

xunits, yunits

[registered units, optional] Override axis units by specifying an instance of a *matplotlib.units.ConversionInterface*.

antialiased

[bool, optional] Enable antialiasing, overriding the defaults. For filled contours, the default is *True*. For line contours, it is taken from `rcParams["lines.antialiased"]` (default: `True`).

nchunk

[int >= 0, optional] If 0, no subdivision of the domain. Specify a positive integer to divide the domain into subdomains of *nchunk* by *nchunk* quads. Chunking reduces the maximum length of polygons generated by the contouring algorithm which reduces the rendering workload passed on to the backend and also requires slightly less RAM. It can however introduce rendering artifacts at chunk boundaries depending on the backend, the *antialiased* flag and value of *alpha*.

linewidths

[float or array-like, default: `rcParams["contour.linewidth"]` (default: `None`)] *Only applies to* *`contour`*.

The line width of the contour lines.

If a number, all levels will be plotted with this linewidth.

If a sequence, the levels in ascending order will be plotted with the linewidths in the order specified.

If None, this falls back to `rcParams["lines.linewidth"]` (default: `1.5`).

linestyles

[{*None*, 'solid', 'dashed', 'dashdot', 'dotted'}, optional] *Only applies to* *`contour`*.

If *linestyles* is *None*, the default is 'solid' unless the lines are monochrome. In that case, negative contours will take their linestyle from `rcParams["contour.negative_linestyle"]` (default: `'dashed'`) setting.

linestyles can also be an iterable of the above strings specifying a set of linestyles to be used. If this iterable is shorter than the number of contour levels it will be repeated as necessary.

hatches

[List[str], optional] *Only applies to* *`contourf`*.

A list of cross hatch patterns to use on the filled areas. If None, no hatching will be added to the contour. Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Notes

1. *contourf* differs from the MATLAB version in that it does not draw the polygon edges. To draw edges, add line contours with calls to *contour*.
2. *contourf* fills intervals that are closed at the top; that is, for boundaries *z1* and *z2*, the filled region is:

```
z1 < Z <= z2
```

   except for the lowest interval, which is closed on both sides (i.e. it includes the lowest value).

Examples using `matplotlib.pyplot.contour`

- sphx_glr_gallery_event_handling_ginput_manual_clabel_sgskip.py

matplotlib.pyplot.contourf

matplotlib.pyplot.**contourf**(**args*, *data=None*, ***kwargs*)

Plot contours.

Call signature:

```
contour([X, Y,] Z, [levels], **kwargs)
```

contour and *contourf* draw contour lines and filled contours, respectively. Except as noted, function signatures and return values are the same for both versions.

Parameters

X, Y

[array-like, optional] The coordinates of the values in *Z*.

X and *Y* must both be 2-D with the same shape as *Z* (e.g. created via `numpy.meshgrid`), or they must both be 1-D such that `len(X) == M` is the number of columns in *Z* and `len(Y) == N` is the number of rows in *Z*.

If not given, they are assumed to be integer indices, i.e. `X = range(M)`, `Y = range(N)`.

Z

[array-like(N, M)] The height values over which the contour is drawn.

levels

[int or array-like, optional] Determines the number and positions of the contour lines / regions.

If an int *n*, use *MaxNLocator*, which tries to automatically choose no more than *n+1* "nice" contour levels between *vmin* and *vmax*.

If array-like, draw contour lines at the specified levels. The values must be in increasing order.

Returns

QuadContourSet

Other Parameters

corner_mask

[bool, default: `rcParams["contour.corner_mask"]` (default: `True`)] Enable/disable corner masking, which only has an effect if *Z* is a masked array. If `False`, any quad touching a masked point is masked out. If `True`, only the triangular corners of quads nearest those points are always masked out, other triangular corners comprising three unmasked points are contoured as usual.

colors

[color string or sequence of colors, optional] The colors of the levels, i.e. the lines for *contour* and the areas for *contourf*.

The sequence is cycled for the levels in ascending order. If the sequence is shorter than the number of levels, it's repeated.

As a shortcut, single color strings may be used in place of one-element lists, i.e. `'red'` instead of `['red']` to color all levels with the same color. This shortcut does only work for color strings, not for other ways of specifying colors.

By default (value *None*), the colormap specified by *cmap* will be used.

alpha

[float, default: 1] The alpha blending value, between 0 (transparent) and 1 (opaque).

cmap

[str or *Colormap*, default: `rcParams["image.cmap"]` (default: `'viridis'`)] A *Colormap* instance or registered colormap name. The colormap maps the level values to colors.

If both *colors* and *cmap* are given, an error is raised.

norm

[*Normalize*, optional] If a colormap is used, the *Normalize* instance scales the level values to the canonical colormap range [0, 1] for mapping to colors. If not given, the default linear scaling is used.

vmin, vmax

[float, optional] If not *None*, either or both of these values will be supplied to the *Normalize* instance, overriding the default color scaling based on *levels*.

origin

[{*None*, 'upper', 'lower', 'image'}, default: None] Determines the orientation and exact position of *Z* by specifying the position of `Z[0, 0]`. This is only relevant, if *X*, *Y* are not given.

- *None*: `Z[0, 0]` is at X=0, Y=0 in the lower left corner.
- 'lower': `Z[0, 0]` is at X=0.5, Y=0.5 in the lower left corner.
- 'upper': `Z[0, 0]` is at X=N+0.5, Y=0.5 in the upper left corner.
- 'image': Use the value from `rcParams["image.origin"]` (default: `'upper'`).

extent

[(x0, x1, y0, y1), optional] If *origin* is not *None*, then *extent* is interpreted as in *imshow*: it gives the outer pixel boundaries. In this case, the position of Z[0, 0] is the center of the pixel, not a corner. If *origin* is *None*, then (*x0*, *y0*) is the position of Z[0, 0], and (*x1*, *y1*) is the position of Z[-1, -1].

This argument is ignored if *X* and *Y* are specified in the call to contour.

locator

[ticker.Locator subclass, optional] The locator is used to determine the contour levels if they are not given explicitly via *levels*. Defaults to *MaxNLocator*.

extend

[{'neither', 'both', 'min', 'max'}, default: 'neither'] Determines the `contourf`-coloring of values that are outside the *levels* range.

If 'neither', values outside the *levels* range are not colored. If 'min', 'max' or 'both', color the values below, above or below and above the *levels* range.

Values below `min(levels)` and above `max(levels)` are mapped to the under/over values of the *Colormap*. Note that most colormaps do not have dedicated colors for these by default, so that the over and under values are the edge values of the colormap. You may

want to set these values explicitly using *Colormap.set_under* and *Colormap.set_over*.

Note: An existing *QuadContourSet* does not get notified if properties of its colormap are changed. Therefore, an explicit call `QuadContourSet.changed()` is needed after modifying the colormap. The explicit call can be left out, if a colorbar is assigned to the *QuadContourSet* because it internally calls `QuadContourSet.changed()`.

Example:

```
x = np.arange(1, 10)
y = x.reshape(-1, 1)
h = x * y

cs = plt.contourf(h, levels=[10, 30, 50],
    colors=['#808080', '#A0A0A0', '#C0C0C0'], extend='both')
cs.cmap.set_over('red')
cs.cmap.set_under('blue')
cs.changed()
```

xunits, yunits

[registered units, optional] Override axis units by specifying an instance of a *matplotlib.units.ConversionInterface*.

antialiased

[bool, optional] Enable antialiasing, overriding the defaults. For filled contours, the default is *True*. For line contours, it is taken from `rcParams["lines.antialiased"]` (default: `True`).

nchunk

[int >= 0, optional] If 0, no subdivision of the domain. Specify a positive integer to divide the domain into subdomains of *nchunk* by *nchunk* quads. Chunking reduces the maximum length of polygons generated by the contouring algorithm which reduces the rendering workload passed on to the backend and also requires slightly less RAM. It can however introduce rendering artifacts at chunk boundaries depending on the backend, the *antialiased* flag and value of *alpha*.

linewidths

[float or array-like, default: `rcParams["contour.linewidth"]` (default: `None`)] *Only applies to* *contour*.

The line width of the contour lines.

If a number, all levels will be plotted with this linewidth.

If a sequence, the levels in ascending order will be plotted with the linewidths in the order specified.

If None, this falls back to `rcParams["lines.linewidth"]` (default: `1.5`).

linestyles

[{*None*, 'solid', 'dashed', 'dashdot', 'dotted'}, optional] *Only applies to* *contour*.

If *linestyles* is *None*, the default is 'solid' unless the lines are monochrome. In that case, negative contours will take their linestyle from `rcParams["contour.negative_linestyle"]` (default: `'dashed'`) setting.

linestyles can also be an iterable of the above strings specifying a set of linestyles to be used. If this iterable is shorter than the number of contour levels it will be repeated as necessary.

hatches

[List[str], optional] *Only applies to* *contourf*.

A list of cross hatch patterns to use on the filled areas. If None, no hatching will be added to the contour. Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Notes

1. *contourf* differs from the MATLAB version in that it does not draw the polygon edges. To draw edges, add line contours with calls to *contour*.
2. *contourf* fills intervals that are closed at the top; that is, for boundaries *z1* and *z2*, the filled region is:

```
z1 < Z <= z2
```

except for the lowest interval, which is closed on both sides (i.e. it includes the lowest value).

Examples using `matplotlib.pyplot.contourf`

- sphx_glr_gallery_images_contours_and_fields_contour_corner_mask.py
- sphx_glr_gallery_images_contours_and_fields_contourf_demo.py
- sphx_glr_gallery_images_contours_and_fields_contourf_hatching.py
- sphx_glr_gallery_images_contours_and_fields_contourf_log.py
- sphx_glr_gallery_images_contours_and_fields_irregulardatagrid.py

- sphx_glr_gallery_images_contours_and_fields_pcolormesh_levels.py
- sphx_glr_gallery_images_contours_and_fields_triinterp_demo.py

matplotlib.pyplot.cool

matplotlib.pyplot.cool()
Set the colormap to "cool".

This changes the default colormap as well as the colormap of the current image if there is one. See `help(colormaps)` for more information.

Examples using `matplotlib.pyplot.cool`

matplotlib.pyplot.copper

matplotlib.pyplot.copper()
Set the colormap to "copper".

This changes the default colormap as well as the colormap of the current image if there is one. See `help(colormaps)` for more information.

Examples using `matplotlib.pyplot.copper`

matplotlib.pyplot.csd

matplotlib.pyplot.csd(*x*, *y*, *NFFT=None*, *Fs=None*, *Fc=None*, *detrend=None*, *window=None*, *noverlap=None*, *pad_to=None*, *sides=None*, *scale_by_freq=None*, *return_line=None*, ***, *data=None*, ***kwargs*)
Plot the cross-spectral density.

The cross spectral density P_{xy} by Welch's average periodogram method. The vectors *x* and *y* are divided into *NFFT* length segments. Each segment is detrended by function *detrend* and windowed by function *window*. *noverlap* gives the length of the overlap between segments. The product of the direct FFTs of *x* and *y* are averaged over each segment to compute P_{xy}, with a scaling to correct for power loss due to windowing.

If len(*x*) < *NFFT* or len(*y*) < *NFFT*, they will be zero padded to *NFFT*.

Parameters

x, y

[1-D arrays or sequences] Arrays or sequences containing the data.

Fs

[float, default: 2] The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, *freqs*, in cycles per time unit.

window

[callable or ndarray, default: *window_hanning*] A function or a vector of length *NFFT*. To create window vectors see *window_hanning*, *window_none*, numpy.blackman, numpy.hamming, numpy.bartlett, scipy.signal, scipy.signal.get_window, etc. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

sides

[{'default', 'onesided', 'twosided'}, optional] Which sides of the spectrum to return. 'default' is one-sided for real data and two-sided for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

pad_to

[int, optional] The number of points to which the data segment is padded when performing the FFT. This can be different from *NFFT*, which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to fft(). The default is None, which sets *pad_to* equal to *NFFT*

NFFT

[int, default: 256] The number of data points used in each block for the FFT. A power 2 is most efficient. This should *NOT* be used to get zero padding, or the scaling of the result will be incorrect; use *pad_to* for this instead.

detrend

[{'none', 'mean', 'linear'} or callable, default 'none'] The function applied to each segment before fft-ing, designed to remove the mean or linear trend. Unlike in MATLAB, where the *detrend* parameter is a vector, in Matplotlib is it a function. The *mlab* module defines *detrend_none*, *detrend_mean*, and *detrend_linear*, but you can use a custom function as well. You can also use a string to choose one of the functions: 'none' calls *detrend_none*. 'mean' calls *detrend_mean*. 'linear' calls *detrend_linear*.

scale_by_freq

[bool, default: True] Whether the resulting density values should be scaled by the scaling frequency, which gives density in units of Hz^-1. This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

noverlap

[int, default: 0 (no overlap)] The number of points of overlap between segments.

Fc

[int, default: 0] The center frequency of *x*, which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

return_line

[bool, default: False] Whether to include the line object plotted in the returned values.

Returns

Pxy

[1-D array] The values for the cross spectrum P_{xy} before scaling (complex valued).

freqs

[1-D array] The frequencies corresponding to the elements in *Pxy*.

line

[*Line2D*] The line created by this function. Only returned if *return_line* is True.

Other Parameters

****kwargs**

Keyword arguments control the *Line2D* properties:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value
alpha	float or None
animated	bool
antialiased or aa	bool
clip_box	*Bbox*
clip_on	bool

Table 192 – continued from previous page

Property	Description
clip_path	Patch or (Path, Transform) or None
color or c	color
contains	unknown
dash_capstyle	{'butt', 'round', 'projecting'}
dash_joinstyle	{'miter', 'round', 'bevel'}
dashes	sequence of floats (on/off ink in points) or (None, None)
data	(2, N) array or two 1D arrays
drawstyle or ds	{'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'd
figure	*Figure*
fillstyle	{'full', 'left', 'right', 'bottom', 'top', 'none'}
gid	str
in_layout	bool
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float
marker	marker style string, *Path* or *MarkerStyle*
markeredgecolor or mec	color
markeredgewidth or mew	float
markerfacecolor or mfc	color
markerfacecoloralt or mfcalt	color
markersize or ms	float
markevery	None or int or (int, int) or slice or List[int] or float or (float, float)
path_effects	*AbstractPathEffect*
picker	unknown
pickradius	float
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
solid_capstyle	{'butt', 'round', 'projecting'}
solid_joinstyle	{'miter', 'round', 'bevel'}
transform	*matplotlib.transforms.Transform*
url	str
visible	bool
xdata	1D array
ydata	1D array
zorder	float

See also:

psd

is equivalent to setting `y = x`.

Notes

For plotting, the power is plotted as $10\log_{10}(P_{xy})$ for decibels, though P_{xy} itself is returned.

References

Bendat & Piersol -- Random Data: Analysis and Measurement Procedures, John Wiley & Sons (1986)

Note: In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, the following arguments can also be string `s`, which is interpreted as `data[s]` (unless this raises an exception): *x*, *y*.

Objects passed as **data** must support item access (`data[s]`) and membership test (`s in data`).

Examples using `matplotlib.pyplot.csd`

matplotlib.pyplot.delaxes

matplotlib.pyplot.**delaxes**(*ax=None*)
Remove an *Axes* (defaulting to the current axes) from its figure.

Examples using `matplotlib.pyplot.delaxes`

matplotlib.pyplot.disconnect

matplotlib.pyplot.**disconnect**(*cid*)
Disconnect the callback with id *cid*.

Examples

```
cid = canvas.mpl_connect('button_press_event', on_press)
# ... later
canvas.mpl_disconnect(cid)
```

Examples using `matplotlib.pyplot.disconnect`

- sphx_glr_gallery_event_handling_coords_demo.py

matplotlib.pyplot.draw

matplotlib.pyplot.**draw**()

Redraw the current figure.

This is used to update a figure that has been altered, but not automatically redrawn. If interactive mode is on (via *ion()*), this should be only rarely needed, but there may be ways to modify the state of a figure without marking it as "stale". Please report these cases as bugs.

This is equivalent to calling `fig.canvas.draw_idle()`, where `fig` is the current figure.

Examples using `matplotlib.pyplot.draw`

- sphx_glr_gallery_event_handling_ginput_manual_clabel_sgskip.py
- sphx_glr_gallery_mplot3d_rotate_axes3d_sgskip.py

matplotlib.pyplot.draw_if_interactive

matplotlib.pyplot.**draw_if_interactive**()

Examples using `matplotlib.pyplot.draw_if_interactive`

matplotlib.pyplot.errorbar

matplotlib.pyplot.**errorbar**(*x*, *y*, *yerr=None*, *xerr=None*, *fmt=''*, *ecolor=None*, *elinewidth=None*, *capsize=None*, *barsabove=False*, *lolims=False*, *uplims=False*, *xlolims=False*, *xuplims=False*, *errorevery=1*, *capthick=None*, ***, *data=None*, ***kwargs*)

Plot y versus x as lines and/or markers with attached errorbars.

x, *y* define the data locations, *xerr*, *yerr* define the errorbar sizes. By default, this draws the data markers/lines as well the errorbars. Use fmt='none' to draw errorbars without any data markers.

Parameters

x, y

[float or array-like] The data positions.

xerr, yerr

[float or array-like, shape(N,) or shape(2, N), optional] The errorbar sizes:

- scalar: Symmetric +/- values for all data points.
- shape(N,): Symmetric +/-values for each data point.
- shape(2, N): Separate - and + values for each bar. First row contains the lower errors, the second row contains the upper errors.
- *None*: No errorbar.

Note that all error arrays should have *positive* values.

See /gallery/statistics/errorbar_features for an example on the usage of `xerr` and `yerr`.

fmt

[str, default: ''] The format for the data points / data lines. See *plot* for details.

Use 'none' (case insensitive) to plot errorbars without any data markers.

ecolor

[color, default: None] The color of the errorbar lines. If None, use the color of the line connecting the markers.

elinewidth

[float, default: None] The linewidth of the errorbar lines. If None, the linewidth of the current style is used.

capsize

[float, default: `rcParams["errorbar.capsize"]` (default: `0.0`)] The length of the error bar caps in points.

capthick

[float, default: None] An alias to the keyword argument *markeredgewidth* (a.k.a. *mew*). This setting is a more sensible name for the property that controls the thickness of the error bar cap in points. For backwards compatibility, if *mew* or *markeredgewidth* are given, then they will over-ride *capthick*. This may change in future releases.

barsabove

[bool, default: False] If True, will plot the errorbars above the plot symbols. Default is below.

lolims, uplims, xlolims, xuplims

[bool, default: False] These arguments can be used to indicate that a value gives only upper/lower limits. In that case a caret symbol is used to indicate this. *lims*-arguments may be scalars, or array-likes of the same length as *xerr* and *yerr*. To use limits with inverted axes, *set_xlim* or *set_ylim* must be called before *errorbar()*. Note the tricky parameter names: setting e.g. *lolims* to True means that the y-value is a *lower* limit of the True value, so, only an *upward*-pointing arrow will be drawn!

errorevery

[int or (int, int), default: 1] draws error bars on a subset of the data. *errorevery* =N draws error bars on the points (x[::N], y[::N]). *errorevery* =(start, N) draws error bars on the points (x[start::N], y[start::N]). e.g. errorevery=(6, 3) adds error bars to the data at (x[6], x[9], x[12], x[15], ...). Used to avoid overlapping error bars when two series share x-axis values.

Returns

ErrorbarContainer

The container contains:

- plotline: *Line2D* instance of x, y plot markers and/or line.
- caplines: A tuple of *Line2D* instances of the error bar caps.
- barlinecols: A tuple of *LineCollection* with the horizontal and vertical error ranges.

Other Parameters

****kwargs**

All other keyword arguments are passed on to the *plot* call drawing the markers. For example, this code makes big red squares with thick green edges:

```
x, y, yerr = rand(3, 10)
errorbar(x, y, yerr, marker='s', mfc='red',
         mec='green', ms=20, mew=4)
```

where *mfc*, *mec*, *ms* and *mew* are aliases for the longer property names, *markerfacecolor*, *markeredgecolor*, *markersize* and *markeredgewidth*.

Valid kwargs for the marker properties are *Line2D* properties:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value
alpha	float or None
animated	bool
antialiased or aa	bool
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color or c	color
contains	unknown
dash_capstyle	{'butt', 'round', 'projecting'}
dash_joinstyle	{'miter', 'round', 'bevel'}
dashes	sequence of floats (on/off ink in points) or (None, None)
data	(2, N) array or two 1D arrays
drawstyle or ds	{'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'd
figure	*Figure*
fillstyle	{'full', 'left', 'right', 'bottom', 'top', 'none'}
gid	str
in_layout	bool
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float
marker	marker style string, *Path* or *MarkerStyle*
markeredgecolor or mec	color
markeredgewidth or mew	float
markerfacecolor or mfc	color
markerfacecoloralt or mfcalt	color
markersize or ms	float
markevery	None or int or (int, int) or slice or List[int] or float or (float, float)
path_effects	*AbstractPathEffect*
picker	unknown
pickradius	float
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
solid_capstyle	{'butt', 'round', 'projecting'}
solid_joinstyle	{'miter', 'round', 'bevel'}
transform	*matplotlib.transforms.Transform*
url	str
visible	bool
xdata	1D array
ydata	1D array
zorder	float

Notes

Note: In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, the following arguments can also be string `s`, which is interpreted as `data[s]` (unless this raises an exception): *x*, *y*, *xerr*, *yerr*.

Objects passed as **data** must support item access (`data[s]`) and membership test (`s in data`).

Examples using `matplotlib.pyplot.errorbar`

- sphx_glr_gallery_lines_bars_and_markers_errorbar_limits_simple.py

matplotlib.pyplot.eventplot

matplotlib.pyplot.**eventplot**(*positions*, *orientation='horizontal'*, *lineoffsets=1*, *linelengths=1*, *linewidths=None*, *colors=None*, *linestyles='solid'*, ***, *data=None*, ***kwargs*)

Plot identical parallel lines at the given positions.

This type of plot is commonly used in neuroscience for representing neural events, where it is usually called a spike raster, dot raster, or raster plot.

However, it is useful in any situation where you wish to show the timing or position of multiple sets of discrete events, such as the arrival times of people to a business on each day of the month or the date of hurricanes each year of the last century.

Parameters

positions

[array-like or list of array-like] A 1D array-like defines the positions of one sequence of events.

Multiple groups of events may be passed as a list of array-likes. Each group can be styled independently by passing lists of values to *lineoffsets*, *linelengths*, *linewidths*, *colors* and *linestyles*.

Note that *positions* can be a 2D array, but in practice different event groups usually have different counts so that one will use a list of different-length arrays rather than a 2D array.

orientation

[{'horizontal', 'vertical'}, default: 'horizontal'] The direction of the event sequence:

- 'horizontal': the events are arranged horizontally. The indicator lines are vertical.
- 'vertical': the events are arranged vertically. The indicator lines are horizontal.

lineoffsets

[float or array-like, default: 1] The offset of the center of the lines from the origin, in the direction orthogonal to *orientation*.

If *positions* is 2D, this can be a sequence with length matching the length of *positions*.

linelengths

[float or array-like, default: 1] The total height of the lines (i.e. the lines stretches from `lineoffset - linelength/2` to `lineoffset + linelength/2`).

If *positions* is 2D, this can be a sequence with length matching the length of *positions*.

linewidths

[float or array-like, default: `rcParams["lines.linewidth"]` (default: `1.5`)] The line width(s) of the event lines, in points.

If *positions* is 2D, this can be a sequence with length matching the length of *positions*.

colors

[color or list of colors, default: `rcParams["lines.color"]` (default: `'C0'`)] The color(s) of the event lines.

If *positions* is 2D, this can be a sequence with length matching the length of *positions*.

linestyles

[str or tuple or list of such values, default: 'solid'] Default is 'solid'. Valid strings are ['solid', 'dashed', 'dashdot', 'dotted', '-', '--', '-.', ':']. Dash tuples should be of the form:

```
(offset, onoffseq),
```

where *onoffseq* is an even length tuple of on and off ink in points.

If *positions* is 2D, this can be a sequence with length matching the length of *positions*.

****kwargs**

Other keyword arguments are line collection properties. See *LineCollection* for a list of the valid properties.

Returns

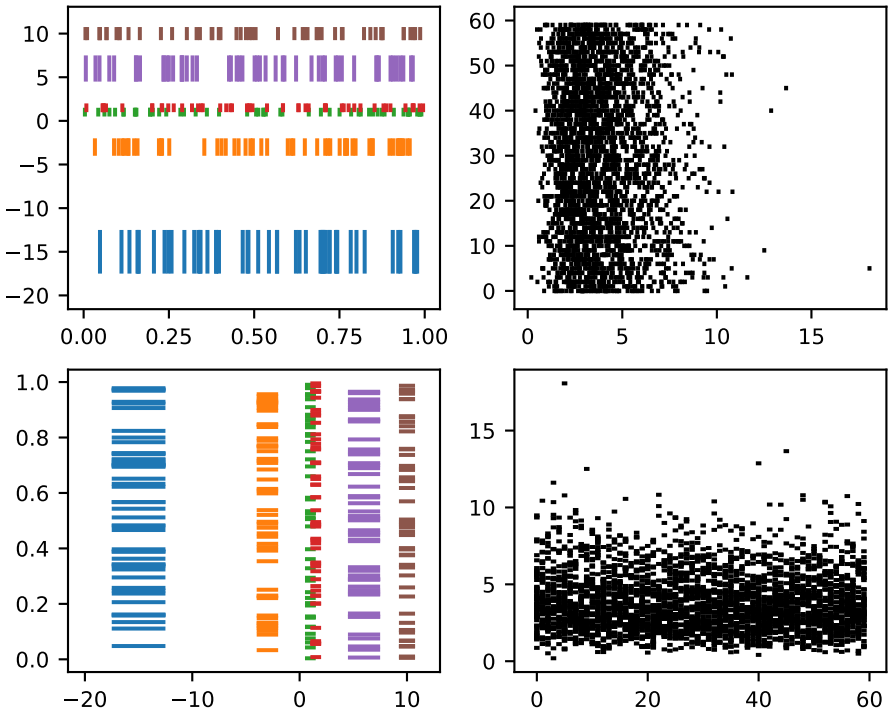
list of *EventCollection*

The *EventCollection* that were added.

Notes

For *linelengths*, *linewidths*, *colors*, and *linestyles*, if only a single value is given, that value is applied to all lines. If an array-like is given, it must have the same length as *positions*, and each value will be applied to the corresponding row of the array.

Examples



Note: In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, the following arguments can also be string `s`, which is interpreted as `data[s]` (unless this raises an exception): *positions*, *lineoffsets*, *linelengths*, *linewidths*, *colors*, *linestyles*.

Objects passed as **data** must support item access (`data[s]`) and membership test (`s in data`).

Examples using `matplotlib.pyplot.eventplot`

matplotlib.pyplot.figimage

matplotlib.pyplot.**figimage**(*X*, *xo=0*, *yo=0*, *alpha=None*, *norm=None*, *cmap=None*, *vmin=None*, *vmax=None*, *origin=None*, *resize=False*, ***kwargs*)

Add a non-resampled image to the figure.

The image is attached to the lower or upper left corner depending on *origin*.

Parameters

X

The image data. This is an array of one of the following shapes:

- MxN: luminance (grayscale) values
- MxNx3: RGB values
- MxNx4: RGBA values

xo, yo

[int] The *x*/*y* image offset in pixels.

alpha

[None or float] The alpha blending value.

norm

[*matplotlib.colors.Normalize*] A *Normalize* instance to map the luminance to the interval [0, 1].

cmap

[str or *matplotlib.colors.Colormap*, default: `rcParams["image.cmap"]` (default: `'viridis'`)] The colormap to use.

vmin, vmax

[float] If *norm* is not given, these values set the data limits for the colormap.

origin

[{'upper', 'lower'}, default: `rcParams["image.origin"]` (default: `'upper'`)] Indicates where the [0, 0] index of the array is in the upper left or lower left corner of the axes.

resize

[bool] If *True*, resize the figure to match the given image size.

Returns

matplotlib.image.FigureImage

Other Parameters

****kwargs**

Additional kwargs are *Artist* kwargs passed on to *FigureImage*.

Notes

figimage complements the axes image (*imshow*) which will be resampled to fit the current axes. If you want a resampled image to fill the entire figure, you can define an *Axes* with extent [0, 0, 1, 1].

Examples

```
f = plt.figure()
nx = int(f.get_figwidth() * f.dpi)
ny = int(f.get_figheight() * f.dpi)
data = np.random.random((ny, nx))
f.figimage(data)
plt.show()
```

Examples using `matplotlib.pyplot.figimage`

- sphx_glr_gallery_images_contours_and_fields_figimage_demo.py

matplotlib.pyplot.figlegend

matplotlib.pyplot.**figlegend**(**args*, ***kwargs*)

Place a legend on the figure.

To make a legend from existing artists on every axes:

```
figlegend()
```

To make a legend for a list of lines and labels:

```
figlegend(
    (line1, line2, line3),
    ('label1', 'label2', 'label3'),
    loc='upper right')
```

These can also be specified by keyword:

```
figlegend(
    handles=(line1, line2, line3),
    labels=('label1', 'label2', 'label3'),
    loc='upper right')
```

Parameters

handles

[list of *Artist*, optional] A list of Artists (lines, patches) to be added to the legend. Use this together with *labels*, if you need full control on what is shown in the legend and the automatic mechanism described above is not sufficient.

The length of handles and labels should be the same in this case. If they are not, they are truncated to the smaller length.

labels

[list of str, optional] A list of labels to show next to the artists. Use this together with *handles*, if you need full control on what is shown in the legend and the automatic mechanism described above is not sufficient.

Returns

Legend

Other Parameters

loc

[str or pair of floats, default: `rcParams["legend.loc"]` (default: `'best'`) ('best' for axes, 'upper right' for figures)] The location of the legend.

The strings `'upper left'`, `'upper right'`, `'lower left'`, `'lower right'` place the legend at the corresponding corner of the axes/figure.

The strings `'upper center'`, `'lower center'`, `'center left'`, `'center right'` place the legend at the center of the corresponding edge of the axes/figure.

The string `'center'` places the legend at the center of the axes/figure.

The string `'best'` places the legend at the location, among the nine locations defined so far, with the minimum overlap with other drawn artists. This option can be quite slow for plots with large amounts of data; your plotting speed may benefit from providing a specific location.

The location can also be a 2-tuple giving the coordinates of the lower-left corner of the legend in axes coordinates (in which case *bbox_to_anchor* will be ignored).

For back-compatibility, `'center right'` (but no other location) can also be spelled `'right'`, and each "string" locations can also be given as a numeric value:

Location String	Location Code
'best'	0
'upper right'	1
'upper left'	2
'lower left'	3
'lower right'	4
'right'	5
'center left'	6
'center right'	7
'lower center'	8
'upper center'	9
'center'	10

bbox_to_anchor

[*BboxBase*, 2-tuple, or 4-tuple of floats] Box that is used to position the legend in conjunction with *loc*. Defaults to `axes.bbox` (if called as a method to *Axes.legend*) or `figure.bbox` (if *Figure.legend*). This argument allows arbitrary placement of the legend.

Bbox coordinates are interpreted in the coordinate system given by *bbox_transform*, with the default transform Axes or Figure coordinates, depending on which `legend` is called.

If a 4-tuple or *BboxBase* is given, then it specifies the bbox `(x, y, width, height)` that the legend is placed in. To put the legend in the best location in the bottom right quadrant of the axes (or figure):

```
loc='best', bbox_to_anchor=(0.5, 0., 0.5, 0.5)
```

A 2-tuple `(x, y)` places the corner of the legend specified by *loc* at x, y. For example, to put the legend's upper right-hand corner

in the center of the axes (or figure) the following keywords can be used:

```
loc='upper right', bbox_to_anchor=(0.5, 0.5)
```

ncol

[int, default: 1] The number of columns that the legend has.

prop

[None or *matplotlib.font_manager.FontProperties* or dict] The font properties of the legend. If None (default), the current *matplotlib.rcParams* will be used.

fontsize

[int or {'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large'}] The font size of the legend. If the value is numeric the size will be the absolute font size in points. String values are relative to the current default font size. This argument is only used if *prop* is not specified.

labelcolor

[str or list] Sets the color of the text in the legend. Can be a valid color string (for example, 'red'), or a list of color strings. The labelcolor can also be made to match the color of the line or marker using 'linecolor', 'markerfacecolor' (or 'mfc'), or 'markeredgecolor' (or 'mec').

numpoints

[int, default: `rcParams["legend.numpoints"]` (default: `1`)] The number of marker points in the legend when creating a legend entry for a *Line2D* (line).

scatterpoints

[int, default: `rcParams["legend.scatterpoints"]` (default: `1`)] The number of marker points in the legend when creating a legend entry for a *PathCollection* (scatter plot).

scatteryoffsets

[iterable of floats, default: `[0.375, 0.5, 0.3125]`] The vertical offset (relative to the font size) for the markers created for a scatter plot legend entry. 0.0 is at the base the legend text, and 1.0 is at the top. To draw all markers at the same height, set to `[0.5]`.

markerscale

[float, default: `rcParams["legend.markerscale"]` (default: `1.0`)] The relative size of legend markers compared with the originally drawn ones.

markerfirst

[bool, default: True] If *True*, legend marker is placed to the left of the legend label. If *False*, legend marker is placed to the right of the legend label.

frameon

[bool, default: `rcParams["legend.frameon"]` (default: `True`)] Whether the legend should be drawn on a patch (frame).

fancybox

[bool, default: `rcParams["legend.fancybox"]` (default: `True`)] Whether round edges should be enabled around the *FancyBboxPatch* which makes up the legend's background.

shadow

[bool, default: `rcParams["legend.shadow"]` (default: `False`)] Whether to draw a shadow behind the legend.

framealpha

[float, default: `rcParams["legend.framealpha"]` (default: `0.8`)] The alpha transparency of the legend's background. If *shadow* is activated and *framealpha* is `None`, the default value is ignored.

facecolor

["inherit" or color, default: `rcParams["legend.facecolor"]` (default: `'inherit'`)] The legend's background color. If `"inherit"`, use `rcParams["axes.facecolor"]` (default: `'white'`).

edgecolor

["inherit" or color, default: `rcParams["legend.edgecolor"]` (default: `'0.8'`)] The legend's background patch edge color. If `"inherit"`, use take `rcParams["axes.edgecolor"]` (default: `'black'`).

mode

[{"expand", None}] If *mode* is set to `"expand"` the legend will be horizontally expanded to fill the axes area (or *bbox_to_anchor* if defines the legend's size).

bbox_transform

[None or *matplotlib.transforms.Transform*] The transform for the bounding box (*bbox_to_anchor*). For a value of `None` (default) the Axes' `transAxes` transform will be used.

title

[str or None] The legend's title. Default is no title (`None`).

title_fontsize

[int or {'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large'}, default: `rcParams["legend.title_fontsize"]` (default: `None`)] The font size of the legend's title.

borderpad

[float, default: `rcParams["legend.borderpad"]` (default: `0.4`)] The fractional whitespace inside the legend border, in font-size units.

labelspacing

[float, default: `rcParams["legend.labelspacing"]` (default: `0.5`)] The vertical space between the legend entries, in font-size units.

handlelength

[float, default: `rcParams["legend.handlelength"]` (default: `2.0`)] The length of the legend handles, in font-size units.

handletextpad

[float, default: `rcParams["legend.handletextpad"]` (default: `0.8`)] The pad between the legend handle and text, in font-size units.

borderaxespad

[float, default: `rcParams["legend.borderaxespad"]` (default: `0.5`)] The pad between the axes and legend border, in font-size units.

columnspacing

[float, default: `rcParams["legend.columnspacing"]` (default: `2.0`)] The spacing between columns, in font-size units.

handler_map

[dict or None] The custom dictionary mapping instances or types to a legend handler. This *handler_map* updates the default handler map found at *matplotlib.legend.Legend.get_legend_handler_map*.

Notes

Some artists are not supported by this function. See *Legend guide* for details.

Examples using `matplotlib.pyplot.figlegend`

matplotlib.pyplot.fignum_exists

matplotlib.pyplot.**fignum_exists**(*num*)

Return whether the figure with the given id exists.

Examples using `matplotlib.pyplot.fignum_exists`

matplotlib.pyplot.figtext

matplotlib.pyplot.**figtext**(*x*, *y*, *s*, *fontdict=None*, ***kwargs*)

Add text to figure.

Parameters

x, y

[float] The position to place the text. By default, this is in figure coordinates, floats in [0, 1]. The coordinate system can be changed using the *transform* keyword.

s

[str] The text string.

fontdict

[dict, optional] A dictionary to override the default text properties. If not given, the defaults are determined by `rcParams["font.*"]`. Properties passed as *kwargs* override the corresponding ones given in *fontdict*.

Returns

Text

Other Parameters

****kwargs**

[*Text* properties] Other miscellaneous text parameters.

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array an
alpha	float or None
animated	bool
backgroundcolor	color

Tabl

Property	Description
bbox	dict with properties for *patches.FancyBboxPatch*
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color or c	color
contains	unknown
figure	*Figure*
fontfamily or family	{FONTNAME, 'serif', 'sans-serif', 'cursive', 'fantasy', '
fontproperties or font or font_properties	*font_manager.FontProperties* or `str` or `pathlib.Path`
fontsize or size	float or {'xx-small', 'x-small', 'small', 'medium', 'large',
fontstretch or stretch	{a numeric value in range 0-1000, 'ultra-condensed',
fontstyle or style	{'normal', 'italic', 'oblique'}
fontvariant or variant	{'normal', 'small-caps'}
fontweight or weight	{a numeric value in range 0-1000, 'ultralight', 'light',
gid	str
horizontalalignment or ha	{'center', 'right', 'left'}
in_layout	bool
label	object
linespacing	float (multiple of font size)
multialignment or ma	{'left', 'right', 'center'}
path_effects	*AbstractPathEffect*
picker	None or bool or callable
position	(float, float)
rasterized	bool or None
rotation	float or {'vertical', 'horizontal'}
rotation_mode	{None, 'default', 'anchor'}
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
text	object
transform	*Transform*
url	str
usetex	bool or None
verticalalignment or va	{'center', 'top', 'bottom', 'baseline', 'center_baseline'}
visible	bool
wrap	bool
x	float
y	float
zorder	float

See also:

Axes.text

pyplot.text

Examples using `matplotlib.pyplot.figtext`

- sphx_glr_gallery_text_labels_and_annotations_fonts_demo.py
- sphx_glr_gallery_text_labels_and_annotations_fonts_demo_kw.py

matplotlib.pyplot.figure

matplotlib.pyplot.**figure**(*num=None*, *figsize=None*, *dpi=None*, *facecolor=None*, *edgecolor=None*, *frameon=True*, *FigureClass=<class 'matplotlib.figure.Figure'>*, *clear=False*, ***kwargs*)

Create a new figure, or activate an existing figure.

Parameters

num

[int or str, optional] A unique identifier for the figure.

If a figure with that identifier already exists, this figure is made active and returned. An integer refers to the `Figure.number` attribute, a string refers to the figure label.

If there is no figure with the identifier or *num* is not given, a new figure is created, made active and returned. If *num* is an int, it will be used for the `Figure.number` attribute, otherwise, an auto-generated integer value is used (starting at 1 and incremented for each new figure). If *num* is a string, the figure label and the window title is set to this value.

figsize

[(float, float), default: `rcParams["figure.figsize"]` (default: `[6.4, 4.8]`)] Width, height in inches.

dpi

[float, default: `rcParams["figure.dpi"]` (default: `100.0`)] The resolution of the figure in dots-per-inch.

facecolor

[color, default: `rcParams["figure.facecolor"]` (default: `'white'`)] The background color.

edgecolor

[color, default: `rcParams["figure.edgecolor"]` (default: `'white'`)] The border color.

frameon

[bool, default: True] If False, suppress drawing the figure frame.

FigureClass

[subclass of *Figure*] Optionally use a custom *Figure* instance.

clear

[bool, default: False] If True and the figure already exists, then it is cleared.

tight_layout

[bool or dict, default: `rcParams["figure.autolayout"]` (default: `False`)] If `False` use *subplotpars*. If `True` adjust subplot parameters using *tight_layout* with default padding. When providing a dict containing the keys `pad`, `w_pad`, `h_pad`, and `rect`, the default *tight_layout* paddings will be overridden.

constrained_layout

[bool, default: `rcParams["figure.constrained_layout.use"]` (default: `False`)] If `True` use constrained layout to adjust positioning of plot elements. Like `tight_layout`, but designed to be more flexible. See *Constrained Layout Guide* for examples. (Note: does not work with `add_subplot` or *subplot2grid*.)

****kwargs : optional**

See *Figure* for other possible arguments.

Returns

Figure

The *Figure* instance returned will also be passed to new_figure_manager in the backends, which allows to hook custom *Figure* classes into the pyplot interface. Additional kwargs will be passed to the *Figure* init function.

Notes

If you are creating many figures, make sure you explicitly call *pyplot.close* on the figures you are not using, because this will enable pyplot to properly clean up the memory.

rcParams defines the default values, which can be modified in the matplotlibrc file.

Examples using `matplotlib.pyplot.figure`

- *Frame grabbing*
- sphx_glr_gallery_misc_hyperlinks_sgskip.py
- sphx_glr_gallery_mplot3d_rotate_axes3d_sgskip.py
- sphx_glr_gallery_mplot3d_wire3d_animation_sgskip.py
- sphx_glr_gallery_user_interfaces_svg_histogram_sgskip.py
- sphx_glr_gallery_user_interfaces_toolmanager_sgskip.py
- sphx_glr_gallery_userdemo_pgf_preamble_sgskip.py

matplotlib.pyplot.fill

`matplotlib.pyplot.fill`(**args*, *data=None*, ***kwargs*)

Plot filled polygons.

Parameters

***args**

[sequence of x, y, [color]] Each polygon is defined by the lists of *x* and *y* positions of its nodes, optionally followed by a *color* specifier. See *matplotlib.colors* for supported color specifiers. The standard color cycle is used for polygons without a color specifier.

You can plot multiple polygons by providing multiple *x*, *y*, *[color]* groups.

For example, each of the following is legal:

```
ax.fill(x, y)                    # a polygon with default color
ax.fill(x, y, "b")               # a blue polygon
ax.fill(x, y, x2, y2)            # two polygons
ax.fill(x, y, "b", x2, y2, "r")  # a blue and a red polygon
```

data

[indexable object, optional] An object with labelled data. If given, provide the label names to plot in *x* and *y*, e.g.:

```
ax.fill("time", "signal",
        data={"time": [0, 1, 2], "signal": [0, 1, 0]})
```

Returns

list of *Polygon*

Other Parameters

****kwargs**

[*Polygon* properties]

Notes

Use *fill_between()* if you would like to fill the region between two curves.

Examples using `matplotlib.pyplot.fill`

- sphx_glr_gallery_event_handling_ginput_manual_clabel_sgskip.py

matplotlib.pyplot.fill_between

`matplotlib.pyplot.`**`fill_between`**(*x*, *y1*, *y2=0*, *where=None*, *interpolate=False*, *step=None*, ***, *data=None*, ***kwargs*)

Fill the area between two horizontal curves.

The curves are defined by the points (*x*, *y1*) and (*x*, *y2*). This creates one or multiple polygons describing the filled area.

You may exclude some horizontal sections from filling using *where*.

By default, the edges connect the given points directly. Use *step* if the filling should be a step function, i.e. constant in between *x*.

Parameters

x

[array (length N)] The x coordinates of the nodes defining the curves.

y1

[array (length N) or scalar] The y coordinates of the nodes defining the first curve.

y2

[array (length N) or scalar, default: 0] The y coordinates of the nodes defining the second curve.

where

[array of bool (length N), optional] Define *where* to exclude some horizontal regions from being filled. The filled regions are defined by the coordinates `x[where]`. More precisely, fill between

`x[i]` and `x[i+1]` if `where[i] and where[i+1]`. Note that this definition implies that an isolated *True* value between two *False* values in *where* will not result in filling. Both sides of the *True* position remain unfilled due to the adjacent *False* values.

interpolate

[bool, default: False] This option is only relevant if *where* is used and the two curves are crossing each other.

Semantically, *where* is often used for *y1 > y2* or similar. By default, the nodes of the polygon defining the filled region will only be placed at the positions in the *x* array. Such a polygon cannot describe the above semantics close to the intersection. The x-sections containing the intersection are simply clipped.

Setting *interpolate* to *True* will calculate the actual intersection point and extend the filled region up to this point.

step

[{'pre', 'post', 'mid'}, optional] Define *step* if the filling should be a step function, i.e. constant in between *x*. The value determines where the step will occur:

- 'pre': The y value is continued constantly to the left from every *x* position, i.e. the interval `(x[i-1], x[i]]` has the value `y[i]`.
- 'post': The y value is continued constantly to the right from every *x* position, i.e. the interval `[x[i], x[i+1])` has the value `y[i]`.
- 'mid': Steps occur half-way between the *x* positions.

Returns

PolyCollection

A *PolyCollection* containing the plotted polygons.

Other Parameters

****kwargs**

All other keyword arguments are passed on to *PolyCollection*. They control the *Polygon* properties:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a
alpha	float or None
animated	bool
antialiased or aa or antialiaseds	bool or list of bools

Table 195 – continued from previous page

Property	Description
array	ndarray
capstyle	{'butt', 'round', 'projecting'}
clim	(vmin: float, vmax: float)
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
cmap	*Colormap* or str or None
color	color or list of rgba tuples
contains	unknown
edgecolor or ec or edgecolors	color or list of colors or 'face'
facecolor or facecolors or fc	color or list of colors
figure	*Figure*
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or dashes or linestyles or ls	str or tuple or list thereof
linewidth or linewidths or lw	float or list of floats
norm	*Normalize* or None
offset_position	unknown
offsets	array-like (N, 2) or (2,)
path_effects	*AbstractPathEffect*
picker	None or bool or callable
pickradius	unknown
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
urls	list of str or None
visible	bool
zorder	float

See also:

fill_between

Fill between two sets of y-values.

fill_betweenx

Fill between two sets of x-values.

Notes

Note: In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, the following arguments can also be string `s`, which is interpreted as `data[s]` (unless this raises an exception): *x*, *y1*, *y2*, *where*.

Objects passed as **data** must support item access (`data[s]`) and membership test (`s in data`).

Examples using `matplotlib.pyplot.fill_between`

- sphx_glr_gallery_lines_bars_and_markers_fill_between_demo.py
- sphx_glr_gallery_text_labels_and_annotations_mathtext_examples.py

matplotlib.pyplot.fill_betweenx

matplotlib.pyplot.**fill_betweenx**(*y*, *x1*, *x2=0*, *where=None*, *step=None*, *interpolate=False*, ***, *data=None*, ***kwargs*)

Fill the area between two vertical curves.

The curves are defined by the points (*y*, *x1*) and (*y*, *x2*). This creates one or multiple polygons describing the filled area.

You may exclude some vertical sections from filling using *where*.

By default, the edges connect the given points directly. Use *step* if the filling should be a step function, i.e. constant in between *y*.

Parameters

y

[array (length N)] The y coordinates of the nodes defining the curves.

x1

[array (length N) or scalar] The x coordinates of the nodes defining the first curve.

x2

[array (length N) or scalar, default: 0] The x coordinates of the nodes defining the second curve.

where

[array of bool (length N), optional] Define *where* to exclude some vertical regions from being filled. The filled regions are defined by the coordinates `y[where]`. More precisely, fill between `y[i]` and `y[i+1]` if `where[i] and where[i+1]`. Note that this definition implies that an isolated *True* value between two *False* values in *where* will not result in filling. Both sides of the *True* position remain unfilled due to the adjacent *False* values.

interpolate

[bool, default: False] This option is only relevant if *where* is used and the two curves are crossing each other.

Semantically, *where* is often used for *x1 > x2* or similar. By default, the nodes of the polygon defining the filled region will only be placed at the positions in the *y* array. Such a polygon cannot describe the above semantics close to the intersection. The y-sections containing the intersection are simply clipped.

Setting *interpolate* to *True* will calculate the actual intersection point and extend the filled region up to this point.

step

[{'pre', 'post', 'mid'}, optional] Define *step* if the filling should be a step function, i.e. constant in between *y*. The value determines where the step will occur:

- 'pre': The y value is continued constantly to the left from every *x* position, i.e. the interval `(x[i-1], x[i]]` has the value `y[i]`.
- 'post': The y value is continued constantly to the right from every *x* position, i.e. the interval `[x[i], x[i+1])` has the value `y[i]`.
- 'mid': Steps occur half-way between the *x* positions.

Returns

PolyCollection

A *PolyCollection* containing the plotted polygons.

Other Parameters

****kwargs**

All other keyword arguments are passed on to *PolyCollection*. They control the *Polygon* properties:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a
alpha	float or None
animated	bool
antialiased or aa or antialiaseds	bool or list of bools
array	ndarray
capstyle	{'butt', 'round', 'projecting'}
clim	(vmin: float, vmax: float)
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
cmap	*Colormap* or str or None
color	color or list of rgba tuples
contains	unknown
edgecolor or ec or edgecolors	color or list of colors or 'face'
facecolor or facecolors or fc	color or list of colors
figure	*Figure*
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or dashes or linestyles or ls	str or tuple or list thereof
linewidth or linewidths or lw	float or list of floats
norm	*Normalize* or None
offset_position	unknown
offsets	array-like (N, 2) or (2,)
path_effects	*AbstractPathEffect*
picker	None or bool or callable
pickradius	unknown
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
urls	list of str or None
visible	bool
zorder	float

See also:

fill_between

Fill between two sets of y-values.

fill_betweenx

Fill between two sets of x-values.

Notes

Note: In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, the following arguments can also be string `s`, which is interpreted as `data[s]` (unless this raises an exception): *y*, *x1*, *x2*, *where*.

Objects passed as **data** must support item access (`data[s]`) and membership test (`s in data`).

Examples using `matplotlib.pyplot.fill_betweenx`

matplotlib.pyplot.findobj

`matplotlib.pyplot.`**`findobj`**(*o=None*, *match=None*, *include_self=True*)

Find artist objects.

Recursively find all *Artist* instances contained in the artist.

Parameters

match

A filter criterion for the matches. This can be

- *None*: Return all objects contained in artist.
- A function with signature `def match(artist: Artist) -> bool`. The result will only contain artists for which the function returns *True*.
- A class instance: e.g., *Line2D*. The result will only contain artists of this class or its subclasses (`isinstance` check).

include_self

[bool] Include *self* in the list to be checked for a match.

Returns

list of *Artist*

Examples using `matplotlib.pyplot.findobj`

matplotlib.pyplot.flag

matplotlib.pyplot.**flag**()

Set the colormap to "flag".

This changes the default colormap as well as the colormap of the current image if there is one. See `help(colormaps)` for more information.

Examples using `matplotlib.pyplot.flag`

matplotlib.pyplot.gca

matplotlib.pyplot.**gca**(***kwargs*)

Get the current axes, creating one if necessary.

The following kwargs are supported for ensuring the returned axes adheres to the given projection etc., and for axes creation if the active axes does not exist:

Property	Description
adjustable	{'box', 'datalim'}
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and r
alpha	float or None
anchor	2-tuple of floats or {'C', 'SW', 'S', 'SE', ...}
animated	bool
aspect	{'auto'} or num
autoscale_on	bool
autoscalex_on	bool
autoscaley_on	bool
axes_locator	Callable[[Axes, Renderer], Bbox]
axisbelow	bool or 'line'
box_aspect	None, or a number
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
contains	unknown
facecolor or fc	color
figure	*Figure*
frame_on	bool
gid	str
in_layout	bool
label	object
navigate	bool

Table 197 – continued from previous page

Property	Description
navigate_mode	unknown
path_effects	*AbstractPathEffect*
picker	None or bool or callable
position	[left, bottom, width, height] or *Bbox*
prop_cycle	unknown
rasterization_zorder	float or None
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
title	str
transform	*Transform*
url	str
visible	bool
xbound	unknown
xlabel	str
xlim	(bottom: float, top: float)
xmargin	float greater than -0.5
xscale	{"linear", "log", "symlog", "logit", ...}
xticklabels	unknown
xticks	unknown
ybound	unknown
ylabel	str
ylim	(bottom: float, top: float)
ymargin	float greater than -0.5
yscale	{"linear", "log", "symlog", "logit", ...}
yticklabels	unknown
yticks	unknown
zorder	float

Examples using `matplotlib.pyplot.gca`

- sphx_glr_gallery_event_handling_ginput_manual_clabel_sgskip.py

matplotlib.pyplot.gcf

matplotlib.pyplot.gcf()
Get the current figure.

If no current figure exists, a new one is created using *figure()*.

Examples using matplotlib.pyplot.gcf

- sphx_glr_gallery_text_labels_and_annotations_arrow_demo.py
- sphx_glr_gallery_event_handling_trifinder_event_demo.py
- sphx_glr_gallery_misc_agg_buffer.py
- *Tight Layout guide*

matplotlib.pyplot.gci

matplotlib.pyplot.gci()
Get the current colorable artist.

Specifically, returns the current *ScalarMappable* instance (Image created by *imshow* or *figimage*, *Collection* created by *pcolor* or *scatter*, etc.), or *None* if no such instance has been defined.

The current image is an attribute of the current axes, or the nearest earlier axes in the current figure that contains an image.

Notes

Historically, the only colorable artists were images; hence the name gci (get current image).

Examples using matplotlib.pyplot.gci

matplotlib.pyplot.get

matplotlib.pyplot.get(*obj*, **args*, ***kwargs*)
Return the value of an object's *property*, or print all of them.

Parameters

obj

[*Artist*] The queried artist; e.g., a *Line2D*, a *Text*, or an *Axes*.

property

[str or None, default: None] If *property* is 'somename', this function returns `obj.get_somename()`.

If is is None (or unset), it *prints* all gettable properties from *obj*. Many properties have aliases for shorter typing, e.g. 'lw' is an alias for 'linewidth'. In the output, aliases and full property names will be listed as:

property or alias = value

e.g.:

linewidth or lw = 2

Examples using `matplotlib.pyplot.get`

matplotlib.pyplot.get_current_fig_manager

matplotlib.pyplot.**get_current_fig_manager**()

Return the figure manager of the current figure.

The figure manager is a container for the actual backend-depended window that displays the figure on screen.

If if no current figure exists, a new one is created an its figure manager is returned.

Returns

FigureManagerBase **or backend-dependent subclass thereof**

Examples using `matplotlib.pyplot.get_current_fig_manager`

matplotlib.pyplot.get_figlabels

matplotlib.pyplot.**get_figlabels**()

Return a list of existing figure labels.

Examples using `matplotlib.pyplot.get_figlabels`

matplotlib.pyplot.get_fignums

`matplotlib.pyplot.get_fignums()`
Return a list of existing figure numbers.

Examples using `matplotlib.pyplot.get_fignums`

matplotlib.pyplot.get_plot_commands

`matplotlib.pyplot.get_plot_commands()`
Get a sorted list of all of the plotting commands.

Examples using `matplotlib.pyplot.get_plot_commands`

matplotlib.pyplot.getp

`matplotlib.pyplot.getp(obj, *args, **kwargs)`
Return the value of an object's *property*, or print all of them.

Parameters

obj

[*Artist*] The queried artist; e.g., a *Line2D*, a *Text*, or an *Axes*.

property

[str or None, default: None] If *property* is 'somename', this function returns `obj.get_somename()`.

If is is None (or unset), it *prints* all gettable properties from *obj*. Many properties have aliases for shorter typing, e.g. 'lw' is an alias for 'linewidth'. In the output, aliases and full property names will be listed as:

property or alias = value

e.g.:

linewidth or lw = 2

Examples using `matplotlib.pyplot.getp`

- sphx_glr_gallery_misc_set_and_get.py
- *Artist tutorial*

matplotlib.pyplot.ginput

```
matplotlib.pyplot.ginput(n=1, timeout=30, show_clicks=True,
                         mouse_add=<MouseButton.LEFT: 1>,
                         mouse_pop=<MouseButton.RIGHT: 3>,
                         mouse_stop=<MouseButton.MIDDLE: 2>)
```

Blocking call to interact with a figure.

Wait until the user clicks *n* times on the figure, and return the coordinates of each click in a list.

There are three possible interactions:

- Add a point.
- Remove the most recently added point.
- Stop the interaction and return the points added so far.

The actions are assigned to mouse buttons via the arguments *mouse_add*, *mouse_pop* and *mouse_stop*.

Parameters

n

[int, default: 1] Number of mouse clicks to accumulate. If negative, accumulate clicks until the input is terminated manually.

timeout

[float, default: 30 seconds] Number of seconds to wait before timing out. If zero or negative will never timeout.

show_clicks

[bool, default: True] If True, show a red cross at the location of each click.

mouse_add

[*MouseButton* or None, default: *MouseButton.LEFT*] Mouse button used to add points.

mouse_pop

[*MouseButton* or None, default: *MouseButton.RIGHT*] Mouse button used to remove the most recently added point.

mouse_stop

[*MouseButton* or None, default: *MouseButton.MIDDLE*] Mouse button used to stop input.

Returns

list of tuples

A list of the clicked (x, y) coordinates.

Notes

The keyboard can also be used to select points in case your mouse does not have one or more of the buttons. The delete and backspace keys act like right clicking (i.e., remove last point), the enter key terminates input and any other key (not already used by the window manager) selects a point.

Examples using `matplotlib.pyplot.ginput`

- sphx_glr_gallery_event_handling_ginput_manual_clabel_sgskip.py

matplotlib.pyplot.gray

matplotlib.pyplot.**gray**()
Set the colormap to "gray".

This changes the default colormap as well as the colormap of the current image if there is one. See `help(colormaps)` for more information.

Examples using `matplotlib.pyplot.gray`

matplotlib.pyplot.grid

matplotlib.pyplot.**grid**(*b=None*, *which='major'*, *axis='both'*, ***kwargs*)
Configure the grid lines.

Parameters

b

[bool or None, optional] Whether to show the grid lines. If any *kwargs* are supplied, it is assumed you want the grid on and *b* will be set to True.

If *b* is *None* and there are no *kwargs*, this toggles the visibility of the lines.

which

[{'major', 'minor', 'both'}, optional] The grid lines to apply the changes on.

axis

[{'both', 'x', 'y'}, optional] The axis to apply the changes on.

****kwargs**

[*Line2D* properties] Define the line properties of the grid, e.g.:

```
grid(color='r', linestyle='-', linewidth=2)
```

Valid keyword arguments are:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value
alpha	float or None
animated	bool
antialiased or aa	bool
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color or c	color
contains	unknown
dash_capstyle	{'butt', 'round', 'projecting'}
dash_joinstyle	{'miter', 'round', 'bevel'}
dashes	sequence of floats (on/off ink in points) or (None, None)
data	(2, N) array or two 1D arrays
drawstyle or ds	{'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'd
figure	*Figure*
fillstyle	{'full', 'left', 'right', 'bottom', 'top', 'none'}
gid	str
in_layout	bool
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float
marker	marker style string, *Path* or *MarkerStyle*
markeredgecolor or mec	color
markeredgewidth or mew	float
markerfacecolor or mfc	color
markerfacecoloralt or mfcalt	color
markersize or ms	float
markevery	None or int or (int, int) or slice or List[int] or float or (float, float)

Table 198 – continued from previous page

Property	Description
path_effects	*AbstractPathEffect*
picker	unknown
pickradius	float
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
solid_capstyle	{'butt', 'round', 'projecting'}
solid_joinstyle	{'miter', 'round', 'bevel'}
transform	*matplotlib.transforms.Transform*
url	str
visible	bool
xdata	1D array
ydata	1D array
zorder	float

Notes

The axis is drawn as a unit, so the effective zorder for drawing the grid is determined by the zorder of each axis, not by the zorder of the *Line2D* objects comprising the grid. Therefore, to set grid zorder, use *set_axisbelow* or, for more control, call the *set_zorder* method of each axis.

Examples using `matplotlib.pyplot.grid`

- sphx_glr_gallery_lines_bars_and_markers_step_demo.py
- sphx_glr_gallery_subplots_axes_and_figures_geo_demo.py
- sphx_glr_gallery_text_labels_and_annotations_multiline.py
- sphx_glr_gallery_pyplots_pyplot_text.py
- sphx_glr_gallery_misc_custom_projection.py
- sphx_glr_gallery_misc_customize_rc.py
- sphx_glr_gallery_misc_findobj_demo.py
- sphx_glr_gallery_scales_custom_scale.py
- sphx_glr_gallery_scales_symlog_demo.py
- sphx_glr_gallery_specialty_plots_skewt.py
- *Pyplot tutorial*

matplotlib.pyplot.hexbin

matplotlib.pyplot.**hexbin**(*x*, *y*, *C=None*, *gridsize=100*, *bins=None*, *xscale='linear'*, *yscale='linear'*, *extent=None*, *cmap=None*, *norm=None*, *vmin=None*, *vmax=None*, *alpha=None*, *linewidths=None*, *edgecolors='face'*, *reduce_C_function=<function mean at 0x7f5439899310>*, *mincnt=None*, *marginals=False*, ***, *data=None*, ***kwargs*)

Make a 2D hexagonal binning plot of points *x*, *y*.

If *C* is *None*, the value of the hexagon is determined by the number of points in the hexagon. Otherwise, *C* specifies values at the coordinate (x[i], y[i]). For each hexagon, these values are reduced using *reduce_C_function*.

Parameters

x, y

[array-like] The data positions. *x* and *y* must be of the same length.

C

[array-like, optional] If given, these values are accumulated in the bins. Otherwise, every point has a value of 1. Must be of the same length as *x* and *y*.

gridsize

[int or (int, int), default: 100] If a single int, the number of hexagons in the *x*-direction. The number of hexagons in the *y*-direction is chosen such that the hexagons are approximately regular.

Alternatively, if a tuple (*nx*, *ny*), the number of hexagons in the *x*-direction and the *y*-direction.

bins

['log' or int or sequence, default: None] Discretization of the hexagon values.

- If *None*, no binning is applied; the color of each hexagon directly corresponds to its count value.
- If 'log', use a logarithmic scale for the color map. Internally, $log_{10}(i+1)$ is used to determine the hexagon color. This is equivalent to `norm=LogNorm()`.
- If an integer, divide the counts in the specified number of bins, and color the hexagons accordingly.

- If a sequence of values, the values of the lower bound of the bins to be used.

xscale

[{'linear', 'log'}, default: 'linear'] Use a linear or log10 scale on the horizontal axis.

yscale

[{'linear', 'log'}, default: 'linear'] Use a linear or log10 scale on the vertical axis.

mincnt

[int > 0, default: *None*] If not *None*, only display cells with more than *mincnt* number of points in the cell.

marginals

[bool, default: *False*] If marginals is *True*, plot the marginal density as colormapped rectangles along the bottom of the x-axis and left of the y-axis.

extent

[float, default: *None*] The limits of the bins. The default assigns the limits based on *gridsize*, *x*, *y*, *xscale* and *yscale*.

If *xscale* or *yscale* is set to 'log', the limits are expected to be the exponent for a power of 10. E.g. for x-limits of 1 and 50 in 'linear' scale and y-limits of 10 and 1000 in 'log' scale, enter (1, 50, 1, 3).

Order of scalars is (left, right, bottom, top).

Returns

PolyCollection

A *PolyCollection* defining the hexagonal bins.

- *PolyCollection.get_offsets* contains a Mx2 array containing the x, y positions of the M hexagon centers.
- *PolyCollection.get_array* contains the values of the M hexagons.

If *marginals* is *True*, horizontal bar and vertical bar (both PolyCollections) will be attached to the return collection as attributes *hbar* and *vbar*.

Other Parameters

cmap

[str or *Colormap*, default: `rcParams["image.cmap"]` (default: `'viridis'`)] The Colormap instance or registered colormap name used to map the bin values to colors.

norm

[*Normalize*, optional] The Normalize instance scales the bin values to the canonical colormap range [0, 1] for mapping to colors. By default, the data range is mapped to the colorbar range using linear scaling.

vmin, vmax

[float, default: None] The colorbar range. If *None*, suitable min/max values are automatically chosen by the *Normalize* instance (defaults to the respective min/max values of the bins in case of the default linear scaling). It is deprecated to use *vmin*/*vmax* when *norm* is given.

alpha

[float between 0 and 1, optional] The alpha blending value, between 0 (transparent) and 1 (opaque).

linewidths

[float, default: *None*] If *None*, defaults to 1.0.

edgecolors

[{'face', 'none', *None*} or color, default: 'face'] The color of the hexagon edges. Possible values are:

- 'face': Draw the edges in the same color as the fill color.
- 'none': No edges are drawn. This can sometimes lead to unsightly unpainted pixels between the hexagons.
- *None*: Draw outlines in the default color.
- An explicit color.

reduce_C_function

[callable, default: `numpy.mean`] The function to aggregate *C* within the bins. It is ignored if *C* is not given. This must have the signature:

```
def reduce_C_function(C: array) -> float
```

Commonly used functions are:

- `numpy.mean`: average of the points
- `numpy.sum`: integral of the point values
- `numpy.max`: value taken from the largest point

****kwargs**

[*PolyCollection* properties] All other keyword arguments are passed on to *PolyCollection*:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a
alpha	float or None
animated	bool
antialiased or aa or antialiaseds	bool or list of bools
array	ndarray
capstyle	{'butt', 'round', 'projecting'}
clim	(vmin: float, vmax: float)
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
cmap	*Colormap* or str or None
color	color or list of rgba tuples
contains	unknown
edgecolor or ec or edgecolors	color or list of colors or 'face'
facecolor or facecolors or fc	color or list of colors
figure	*Figure*
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or dashes or linestyles or ls	str or tuple or list thereof
linewidth or linewidths or lw	float or list of floats
norm	*Normalize* or None
offset_position	unknown
offsets	array-like (N, 2) or (2,)
path_effects	*AbstractPathEffect*
picker	None or bool or callable
pickradius	unknown
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
urls	list of str or None
visible	bool
zorder	float

Notes

Note: In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, the following arguments can also be string `s`, which is interpreted as `data[s]` (unless this raises an exception): *x*, *y*.

Objects passed as **data** must support item access (`data[s]`) and membership test (`s in data`).

Examples using `matplotlib.pyplot.hexbin`

matplotlib.pyplot.hist

matplotlib.pyplot.**hist**(*x*, *bins=None*, *range=None*, *density=False*, *weights=None*, *cumulative=False*, *bottom=None*, *histtype='bar'*, *align='mid'*, *orientation='vertical'*, *rwidth=None*, *log=False*, *color=None*, *label=None*, *stacked=False*, ***, *data=None*, ***kwargs*)

Plot a histogram.

Compute and draw the histogram of *x*. The return value is a tuple (*n*, *bins*, *patches*) or ([*n0*, *n1*, ...], *bins*, [*patches0*, *patches1*, ...]) if the input contains multiple data. See the documentation of the *weights* parameter to draw a histogram of already-binned data.

Multiple data can be provided via *x* as a list of datasets of potentially different length ([*x0*, *x1*, ...]), or as a 2-D ndarray in which each column is a dataset. Note that the ndarray form is transposed relative to the list form.

Masked arrays are not supported.

The *bins*, *range*, *weights*, and *density* parameters behave as in `numpy.histogram`.

Parameters

x

[(n,) array or sequence of (n,) arrays] Input values, this takes either a single array or a sequence of arrays which are not required to be of the same length.

bins

[int or sequence or str, default: `rcParams["hist.bins"]` (default: `10`)] If *bins* is an integer, it defines the number of equal-width bins in the range.

If *bins* is a sequence, it defines the bin edges, including the left edge of the first bin and the right edge of the last bin; in this case, bins may be unequally spaced. All but the last (righthand-most) bin is half-open. In other words, if *bins* is:

```
[1, 2, 3, 4]
```

then the first bin is `[1, 2)` (including 1, but excluding 2) and the second `[2, 3)`. The last bin, however, is `[3, 4]`, which *includes* 4.

If *bins* is a string, it is one of the binning strategies supported by `numpy.histogram_bin_edges`: 'auto', 'fd', 'doane', 'scott', 'stone', 'rice', 'sturges', or 'sqrt'.

range

[tuple or None, default: None] The lower and upper range of the bins. Lower and upper outliers are ignored. If not provided, *range* is `(x.min(), x.max())`. Range has no effect if *bins* is a sequence.

If *bins* is a sequence or *range* is specified, autoscaling is based on the specified bin range instead of the range of x.

density

[bool, default: False] If `True`, draw and return a probability density: each bin will display the bin's raw count divided by the total number of counts *and the bin width* (`density = counts / (sum(counts) * np.diff(bins))`), so that the area under the histogram integrates to 1 (`np.sum(density * np.diff(bins)) == 1`).

If *stacked* is also `True`, the sum of the histograms is normalized to 1.

weights

[(n,) array-like or None, default: None] An array of weights, of the same shape as *x*. Each value in *x* only contributes its associated weight towards the bin count (instead of 1). If *density* is `True`, the weights are normalized, so that the integral of the density over the range remains 1.

This parameter can be used to draw a histogram of data that has already been binned, e.g. using `numpy.histogram` (by treating each bin as a single point with a weight equal to its count)

```
counts, bins = np.histogram(data)
plt.hist(bins[:-1], bins, weights=counts)
```

(or you may alternatively use *bar()*).

cumulative

[bool or -1, default: False] If `True`, then a histogram is computed where each bin gives the counts in that bin plus all bins for smaller values. The last bin gives the total number of datapoints.

If *density* is also `True` then the histogram is normalized such that the last bin equals 1.

If *cumulative* is a number less than 0 (e.g., -1), the direction of accumulation is reversed. In this case, if *density* is also `True`, then the histogram is normalized such that the first bin equals 1.

bottom

[array-like, scalar, or None, default: None] Location of the bottom of each bin, ie. bins are drawn from `bottom` to `bottom + hist(x, bins)` If a scalar, the bottom of each bin is shifted by the same amount. If an array, each bin is shifted independently and the length of bottom must match the number of bins. If None, defaults to 0.

histtype

[{'bar', 'barstacked', 'step', 'stepfilled'}, default: 'bar'] The type of histogram to draw.

- 'bar' is a traditional bar-type histogram. If multiple data are given the bars are arranged side by side.
- 'barstacked' is a bar-type histogram where multiple data are stacked on top of each other.
- 'step' generates a lineplot that is by default unfilled.
- 'stepfilled' generates a lineplot that is by default filled.

align

[{'left', 'mid', 'right'}, default: 'mid'] The horizontal alignment of the histogram bars.

- 'left': bars are centered on the left bin edges.
- 'mid': bars are centered between the bin edges.
- 'right': bars are centered on the right bin edges.

orientation

[{'vertical', 'horizontal'}, default: 'vertical'] If 'horizontal', *barh* will be used for bar-type histograms and the *bottom* kwarg will be the left edges.

rwidth

[float or None, default: None] The relative width of the bars as a fraction of the bin width. If `None`, automatically compute the width.

Ignored if *histtype* is 'step' or 'stepfilled'.

log

[bool, default: False] If `True`, the histogram axis will be set to a log scale. If *log* is `True` and *x* is a 1D array, empty bins will be filtered out and only the non-empty `(n, bins, patches)` will be returned.

color

[color or array-like of colors or None, default: None] Color or sequence of colors, one per dataset. Default (`None`) uses the standard line color sequence.

label

[str or None, default: None] String, or sequence of strings to match multiple datasets. Bar charts yield multiple patches per dataset, but only the first gets the label, so that *legend* will work as expected.

stacked

[bool, default: False] If `True`, multiple data are stacked on top of each other If `False` multiple data are arranged side by side if histtype is 'bar' or on top of each other if histtype is 'step'

Returns

n

[array or list of arrays] The values of the histogram bins. See *density* and *weights* for a description of the possible semantics. If input *x* is an array, then this is an array of length *nbins*. If input is a sequence of arrays `[data1, data2, ...]`, then this is a list of arrays with the values of the histograms for each of the arrays in the same order. The dtype of the array *n* (or of its element arrays) will always be float even if no weighting or normalization is used.

bins

[array] The edges of the bins. Length nbins + 1 (nbins left edges and right edge of last bin). Always a single array even when multiple data sets are passed in.

patches

[*BarContainer* or list of a single *Polygon* or list of such objects] Container of individual artists used to create the histogram or list of such containers if there are multiple input datasets.

Other Parameters

****kwargs**

Patch properties

See also:

hist2d

2D histograms

Notes

For large numbers of bins (>1000), 'step' and 'stepfilled' can be significantly faster than 'bar' and 'barstacked'.

Note: In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, the following arguments can also be string `s`, which is interpreted as `data[s]` (unless this raises an exception): *x*, *weights*.

Objects passed as **data** must support item access (`data[s]`) and membership test (`s in data`).

Examples using `matplotlib.pyplot.hist`

- sphx_glr_gallery_user_interfaces_svg_histogram_sgskip.py

matplotlib.pyplot.hist2d

matplotlib.pyplot.**hist2d**(*x*, *y*, *bins=10*, *range=None*, *density=False*, *weights=None*, *cmin=None*, *cmax=None*, *, *data=None*, ***kwargs*)

Make a 2D histogram plot.

Parameters

x, y

[array-like, shape (n,)] Input values

bins

[None or int or [int, int] or array-like or [array, array]] The bin specification:

- If int, the number of bins for the two dimensions (nx=ny=bins).
- If `[int, int]`, the number of bins in each dimension (nx, ny = bins).
- If array-like, the bin edges for the two dimensions (x_edges=y_edges=bins).
- If `[array, array]`, the bin edges in each dimension (x_edges, y_edges = bins).

The default value is 10.

range

[array-like shape(2, 2), optional] The leftmost and rightmost edges of the bins along each dimension (if not specified explicitly in the bins parameters): `[[xmin, xmax], [ymin, ymax]]`. All values outside of this range will be considered outliers and not tallied in the histogram.

density

[bool, default: False] Normalize histogram. See the documentation for the *density* parameter of *hist* for more details.

weights

[array-like, shape (n,), optional] An array of values w_i weighing each sample (x_i, y_i).

cmin, cmax

[float, default: None] All bins that has count less than *cmin* or more than *cmax* will not be displayed (set to NaN before passing to imshow) and these count values in the return value count histogram will also be set to nan upon return.

Returns

h

[2D array] The bi-dimensional histogram of samples x and y. Values in x are histogrammed along the first dimension and values in y are histogrammed along the second dimension.

xedges

[1D array] The bin edges along the x axis.

yedges

[1D array] The bin edges along the y axis.

image

[*QuadMesh*]

Other Parameters

cmap

[Colormap or str, optional] A *colors.Colormap* instance. If not set, use rc settings.

norm

[Normalize, optional] A *colors.Normalize* instance is used to scale luminance data to `[0, 1]`. If not set, defaults to *colors.Normalize()*.

vmin/vmax

[None or scalar, optional] Arguments passed to the *Normalize* instance.

alpha

[`0 <= scalar <= 1` or `None`, optional] The alpha blending value.

****kwargs**

Additional parameters are passed along to the *pcolormesh* method and *QuadMesh* constructor.

See also:

hist

1D histogram plotting

Notes

- Currently `hist2d` calculates its own axis limits, and any limits previously set are ignored.
- Rendering the histogram with a logarithmic color scale is accomplished by passing a *colors.LogNorm* instance to the *norm* keyword argument. Likewise, power-law normalization (similar in effect to gamma correction) can be accomplished with *colors.PowerNorm*.

Note: In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, the following arguments can also be string `s`, which is interpreted as `data[s]` (unless this raises an exception): *x*, *y*, *weights*.

Objects passed as **data** must support item access (`data[s]`) and membership test (`s in data`).

Examples using `matplotlib.pyplot.hist2d`

- sphx_glr_gallery_scales_power_norm.py

matplotlib.pyplot.hlines

`matplotlib.pyplot.`**`hlines`**(*y*, *xmin*, *xmax*, *colors=None*, *linestyles='solid'*, *label=''*, ***, *data=None*, ***kwargs*)

Plot horizontal lines at each *y* from *xmin* to *xmax*.

Parameters

y

[float or array-like] y-indexes where to plot the lines.

xmin, xmax

[float or array-like] Respective beginning and end of each line. If scalars are provided, all lines will have same length.

colors

[list of colors, default: `rcParams["lines.color"]` (default: `'C0'`)]

linestyles

[{'solid', 'dashed', 'dashdot', 'dotted'}, optional]

label

[str, default: '']

Returns

LineCollection

Other Parameters

****kwargs**

[*LineCollection* properties.]

See also:

vlines

vertical lines

axhline

horizontal line across the axes

Notes

Note: In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, the following arguments can also be string `s`, which is interpreted as `data[s]` (unless this raises an exception): *y*, *xmin*, *xmax*, *colors*.

Objects passed as **data** must support item access (`data[s]`) and membership test (`s in data`).

Examples using `matplotlib.pyplot.hlines`

matplotlib.pyplot.hot

matplotlib.pyplot.**hot**()
Set the colormap to "hot".

This changes the default colormap as well as the colormap of the current image if there is one. See `help(colormaps)` for more information.

Examples using `matplotlib.pyplot.hot`

matplotlib.pyplot.hsv

matplotlib.pyplot.**hsv**()
Set the colormap to "hsv".

This changes the default colormap as well as the colormap of the current image if there is one. See `help(colormaps)` for more information.

Examples using `matplotlib.pyplot.hsv`

matplotlib.pyplot.imread

matplotlib.pyplot.**imread**(*fname*, *format=None*)
Read an image from a file into an array.

Parameters

fname

[str or file-like] The image file to read: a filename, a URL or a file-like object opened in read-binary mode.

format

[str, optional] The image file format assumed for reading the data. If not given, the format is deduced from the filename. If nothing can be deduced, PNG is tried.

Returns

`numpy.array`

The image data. The returned array has shape

- (M, N) for grayscale images.
- (M, N, 3) for RGB images.
- (M, N, 4) for RGBA images.

Examples using `matplotlib.pyplot.imread`

- sphx_glr_gallery_images_contours_and_fields_image_clip_path.py
- sphx_glr_gallery_images_contours_and_fields_image_demo.py
- sphx_glr_gallery_images_contours_and_fields_watermark_image.py
- sphx_glr_gallery_text_labels_and_annotations_demo_annotation_box.py
- sphx_glr_gallery_text_labels_and_annotations_demo_text_path.py
- sphx_glr_gallery_misc_demo_ribbon_box.py

matplotlib.pyplot.imsave

`matplotlib.pyplot.`**`imsave`**(*fname*, *arr*, ***kwargs*)

Save an array as an image file.

Parameters

fname

[str or path-like or file-like] A path or a file-like object to store the image in. If *format* is not set, then the output format is inferred from the extension of *fname*, if any, and from `rcParams["savefig.format"]` (default: `'png'`) otherwise. If *format* is set, it determines the output format.

arr

[array-like] The image data. The shape can be one of MxN (luminance), MxNx3 (RGB) or MxNx4 (RGBA).

vmin, vmax

[float, optional] *vmin* and *vmax* set the color scaling for the image by fixing the values that map to the colormap color limits. If either *vmin* or *vmax* is None, that limit is determined from the *arr* min/max value.

cmap

[str or *Colormap*, default: `rcParams["image.cmap"]` (default: `'viridis'`)] A Colormap instance or registered colormap name. The colormap maps scalar data to colors. It is ignored for RGB(A) data.

format

[str, optional] The file format, e.g. 'png', 'pdf', 'svg', ... The behavior when this is unset is documented under *fname*.

origin

[{'upper', 'lower'}, default: `rcParams["image.origin"]` (default: `'upper'`)] Indicates whether the `(0, 0)` index of the array is in the upper left or lower left corner of the axes.

dpi

[float] The DPI to store in the metadata of the file. This does not affect the resolution of the output image. Depending on file format, this may be rounded to the nearest integer.

metadata

[dict, optional] Metadata in the image file. The supported keys depend on the output format, see the documentation of the respective backends for more information.

pil_kwargs

[dict, optional] Keyword arguments passed to `PIL.Image.Image.save`. If the 'pnginfo' key is present, it completely overrides *metadata*, including the default 'Software' key.

Examples using `matplotlib.pyplot.imsave`

matplotlib.pyplot.imshow

matplotlib.pyplot.**imshow**(*X*, *cmap=None*, *norm=None*, *aspect=None*, *interpolation=None*, *alpha=None*, *vmin=None*, *vmax=None*, *origin=None*, *extent=None*, ***, *filternorm=True*, *filterrad=4.0*, *resample=None*, *url=None*, *data=None*, ***kwargs*)

Display data as an image, i.e., on a 2D regular raster.

The input may either be actual RGB(A) data, or 2D scalar data, which will be rendered as a pseudocolor image. For displaying a grayscale image set up the color mapping using the parameters `cmap='gray', vmin=0, vmax=255`.

The number of pixels used to render an image is set by the axes size and the *dpi* of the figure. This can lead to aliasing artifacts when the image is resampled because the displayed image size will usually not match the size of *X* (see /gallery/images_contours_and_fields/image_antialiasing). The resampling can be controlled via the *interpolation* parameter and/or `rcParams["image.interpolation"]` (default: `'antialiased'`).

Parameters

X

[array-like or PIL image] The image data. Supported array shapes are:

- (M, N): an image with scalar data. The values are mapped to colors using normalization and a colormap. See parameters *norm*, *cmap*, *vmin*, *vmax*.
- (M, N, 3): an image with RGB values (0-1 float or 0-255 int).
- (M, N, 4): an image with RGBA values (0-1 float or 0-255 int), i.e. including transparency.

The first two dimensions (M, N) define the rows and columns of the image.

Out-of-range RGB(A) values are clipped.

cmap

[str or *Colormap*, default: `rcParams["image.cmap"]` (default: `'viridis'`)] The Colormap instance or registered colormap name used to map scalar data to colors. This parameter is ignored for RGB(A) data.

norm

[*Normalize*, optional] The *Normalize* instance used to scale scalar data to the [0, 1] range before mapping to colors using *cmap*. By default, a linear scaling mapping the lowest value to 0 and the highest to 1 is used. This parameter is ignored for RGB(A) data.

aspect

[{'equal', 'auto'} or float, default: `rcParams["image.aspect"]` (default: `'equal'`)] The aspect ratio of the axes. This parameter is particularly relevant for images since it determines whether data pixels are square.

This parameter is a shortcut for explicitly calling *Axes.set_aspect*. See there for further details.

- 'equal': Ensures an aspect ratio of 1. Pixels will be square (unless pixel sizes are explicitly made non-square in data coordinates using *extent*).
- 'auto': The axes is kept fixed and the aspect is adjusted so that the data fit in the axes. In general, this will result in non-square pixels.

interpolation

[str, default: `rcParams["image.interpolation"]` (default: `'antialiased'`)] The interpolation method used.

Supported values are 'none', 'antialiased', 'nearest', 'bilinear', 'bicubic', 'spline16', 'spline36', 'hanning', 'hamming', 'hermite', 'kaiser', 'quadric', 'catrom', 'gaussian', 'bessel', 'mitchell', 'sinc', 'lanczos'.

If *interpolation* is 'none', then no interpolation is performed on the Agg, ps, pdf and svg backends. Other backends will fall back to 'nearest'. Note that most SVG renderers perform interpolation at rendering and that the default interpolation method they implement may differ.

If *interpolation* is the default 'antialiased', then 'nearest' interpolation is used if the image is upsampled by more than a factor of three (i.e. the number of display pixels is at least three times the size of the data array). If the upsampling rate is smaller than 3, or the image is downsampled, then 'hanning' interpolation is used to act as an anti-aliasing filter, unless the image happens to be upsampled by exactly a factor of two or one.

See /gallery/images_contours_and_fields/interpolation_methods for an overview of the supported interpolation methods, and /gallery/images_contours_and_fields/image_antialiasing for a discussion of image antialiasing.

Some interpolation methods require an additional radius parameter, which can be set by *filterrad*. Additionally, the antigrain image resize filter is controlled by the parameter *filternorm*.

alpha

[float or array-like, optional] The alpha blending value, between 0 (transparent) and 1 (opaque). If *alpha* is an array, the alpha blending values are applied pixel by pixel, and *alpha* must have the same shape as *X*.

vmin, vmax

[float, optional] When using scalar data and no explicit *norm*, *vmin* and *vmax* define the data range that the colormap covers. By default, the colormap covers the complete value range of the

supplied data. It is deprecated to use *vmin*/*vmax* when *norm* is given.

origin

[{'upper', 'lower'}, default: `rcParams["image.origin"]` (default: `'upper'`)] Place the [0, 0] index of the array in the upper left or lower left corner of the axes. The convention (the default) 'upper' is typically used for matrices and images.

Note that the vertical axes points upward for 'lower' but downward for 'upper'.

See the *origin and extent in imshow* tutorial for examples and a more detailed description.

extent

[floats (left, right, bottom, top), optional] The bounding box in data coordinates that the image will fill. The image is stretched individually along x and y to fill the box.

The default extent is determined by the following conditions. Pixels have unit size in data coordinates. Their centers are on integer coordinates, and their center coordinates range from 0 to columns-1 horizontally and from 0 to rows-1 vertically.

Note that the direction of the vertical axis and thus the default values for top and bottom depend on *origin*:

- For `origin == 'upper'` the default is `(-0.5, numcols-0.5, numrows-0.5, -0.5)`.
- For `origin == 'lower'` the default is `(-0.5, numcols-0.5, -0.5, numrows-0.5)`.

See the *origin and extent in imshow* tutorial for examples and a more detailed description.

filternorm

[bool, default: True] A parameter for the antigrain image resize filter (see the antigrain documentation). If *filternorm* is set, the filter normalizes integer values and corrects the rounding errors. It doesn't do anything with the source floating point values, it corrects only integers according to the rule of 1.0 which means that any sum of pixel weights must be equal to 1.0. So, the filter function must produce a graph of the proper shape.

filterrad

[float > 0, default: 4.0] The filter radius for filters that have a radius parameter, i.e. when interpolation is one of: 'sinc', 'lanczos' or 'blackman'.

resample

[bool, default: `rcParams["image.resample"]` (default: `True`)] When *True*, use a full resampling method. When *False*, only resample when the output image is larger than the input image.

url

[str, optional] Set the url of the created *AxesImage*. See *Artist.set_url*.

Returns

AxesImage

Other Parameters

****kwargs**

[*Artist* properties] These parameters are passed on to the constructor of the *AxesImage* artist.

See also:

matshow

Plot a matrix or an array as an image.

Notes

Unless *extent* is used, pixel centers will be located at integer coordinates. In other words: the origin will coincide with the center of pixel (0, 0).

There are two common representations for RGB images with an alpha channel:

- Straight (unassociated) alpha: R, G, and B channels represent the color of the pixel, disregarding its opacity.
- Premultiplied (associated) alpha: R, G, and B channels represent the color of the pixel, adjusted for its opacity by multiplication.

imshow expects RGB images adopting the straight (unassociated) alpha representation.

Note: In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, every other argument can also be string `s`, which is interpreted as `data[s]` (unless this raises an exception).

Objects passed as **data** must support item access (`data[s]`) and membership test (`s in data`).

Examples using `matplotlib.pyplot.imshow`

- sphx_glr_gallery_misc_hyperlinks_sgskip.py

matplotlib.pyplot.inferno

matplotlib.pyplot.**inferno**()
Set the colormap to "inferno".

This changes the default colormap as well as the colormap of the current image if there is one. See `help(colormaps)` for more information.

Examples using `matplotlib.pyplot.inferno`

matplotlib.pyplot.install_repl_displayhook

matplotlib.pyplot.**install_repl_displayhook**()
Install a repl display hook so that any stale figure are automatically redrawn when control is returned to the repl.

This works both with IPython and with vanilla python shells.

Examples using `matplotlib.pyplot.install_repl_displayhook`

matplotlib.pyplot.ioff

matplotlib.pyplot.**ioff**()
Turn the interactive mode off.

See also:

ion
enable interactive mode

isinteractive
query current state

show
show windows (and maybe block)

pause

show windows, run GUI event loop, and block for a time

Examples using `matplotlib.pyplot.ioff`

- *Usage Guide*

matplotlib.pyplot.ion

`matplotlib.pyplot.ion()`

Turn the interactive mode on.

See also:

ioff

disable interactive mode

isinteractive

query current state

show

show windows (and maybe block)

pause

show windows, run GUI event loop, and block for a time

Examples using `matplotlib.pyplot.ion`

- *Usage Guide*

matplotlib.pyplot.isinteractive

`matplotlib.pyplot.isinteractive()`

Return if pyplot is in "interactive mode" or not.

If in interactive mode then:

- newly created figures will be shown immediately
- figures will automatically redraw on change
- *pyplot.show* will not block by default

If not in interactive mode then:

- newly created figures and changes to figures will not be reflected until explicitly asked to be
- *pyplot.show* will block by default

See also:

ion

enable interactive mode

ioff

disable interactive mode

show

show windows (and maybe block)

pause

show windows, run GUI event loop, and block for a time

Examples using `matplotlib.pyplot.isinteractive`

matplotlib.pyplot.jet

matplotlib.pyplot.**jet**()

Set the colormap to "jet".

This changes the default colormap as well as the colormap of the current image if there is one. See `help(colormaps)` for more information.

Examples using `matplotlib.pyplot.jet`

matplotlib.pyplot.legend

matplotlib.pyplot.**legend**(**args*, ***kwargs*)

Place a legend on the axes.

Call signatures:

```
legend()
legend(labels)
legend(handles, labels)
```

The call signatures correspond to three different ways how to use this method.

1. Automatic detection of elements to be shown in the legend

The elements to be added to the legend are automatically determined, when you do not pass in any extra arguments.

In this case, the labels are taken from the artist. You can specify them either at artist creation or by calling the *set_label()* method on the artist:

```
line, = ax.plot([1, 2, 3], label='Inline label')
ax.legend()
```

or:

```
line, = ax.plot([1, 2, 3])
line.set_label('Label via method')
ax.legend()
```

Specific lines can be excluded from the automatic legend element selection by defining a label starting with an underscore. This is default for all artists, so calling *Axes.legend* without any arguments and without setting the labels manually will result in no legend being drawn.

2. Labeling existing plot elements

To make a legend for lines which already exist on the axes (via plot for instance), simply call this function with an iterable of strings, one for each legend item. For example:

```
ax.plot([1, 2, 3])
ax.legend(['A simple line'])
```

Note: This way of using is discouraged, because the relation between plot elements and labels is only implicit by their order and can easily be mixed up.

3. Explicitly defining the elements in the legend

For full control of which artists have a legend entry, it is possible to pass an iterable of legend artists followed by an iterable of legend labels respectively:

```
legend((line1, line2, line3), ('label1', 'label2', 'label3'))
```

Parameters

handles

[sequence of *Artist*, optional] A list of Artists (lines, patches) to be added to the legend. Use this together with *labels*, if you need full control on what is shown in the legend and the automatic mechanism described above is not sufficient.

The length of handles and labels should be the same in this case. If they are not, they are truncated to the smaller length.

labels

[list of str, optional] A list of labels to show next to the artists. Use this together with *handles*, if you need full control on what

is shown in the legend and the automatic mechanism described above is not sufficient.

Returns

Legend

Other Parameters

loc

[str or pair of floats, default: `rcParams["legend.loc"]` (default: `'best'`) ('best' for axes, 'upper right' for figures)] The location of the legend.

The strings `'upper left'`, `'upper right'`, `'lower left'`, `'lower right'` place the legend at the corresponding corner of the axes/figure.

The strings `'upper center'`, `'lower center'`, `'center left'`, `'center right'` place the legend at the center of the corresponding edge of the axes/figure.

The string `'center'` places the legend at the center of the axes/figure.

The string `'best'` places the legend at the location, among the nine locations defined so far, with the minimum overlap with other drawn artists. This option can be quite slow for plots with large amounts of data; your plotting speed may benefit from providing a specific location.

The location can also be a 2-tuple giving the coordinates of the lower-left corner of the legend in axes coordinates (in which case *bbox_to_anchor* will be ignored).

For back-compatibility, `'center right'` (but no other location) can also be spelled `'right'`, and each "string" locations can also be given as a numeric value:

Location String	Location Code
'best'	0
'upper right'	1
'upper left'	2
'lower left'	3
'lower right'	4
'right'	5
'center left'	6
'center right'	7
'lower center'	8
'upper center'	9
'center'	10

bbox_to_anchor

[*BboxBase*, 2-tuple, or 4-tuple of floats] Box that is used to position the legend in conjunction with *loc*. Defaults to `axes.bbox` (if called as a method to *Axes.legend*) or `figure.bbox` (if *Figure.legend*). This argument allows arbitrary placement of the legend.

Bbox coordinates are interpreted in the coordinate system given by *bbox_transform*, with the default transform Axes or Figure coordinates, depending on which `legend` is called.

If a 4-tuple or *BboxBase* is given, then it specifies the bbox `(x, y, width, height)` that the legend is placed in. To put the legend in the best location in the bottom right quadrant of the axes (or figure):

```
loc='best', bbox_to_anchor=(0.5, 0., 0.5, 0.5)
```

A 2-tuple `(x, y)` places the corner of the legend specified by *loc* at x, y. For example, to put the legend's upper right-hand corner in the center of the axes (or figure) the following keywords can be used:

```
loc='upper right', bbox_to_anchor=(0.5, 0.5)
```

ncol

[int, default: 1] The number of columns that the legend has.

prop

[None or *matplotlib.font_manager.FontProperties* or dict] The font properties of the legend. If None (default), the current *matplotlib.rcParams* will be used.

fontsize

[int or {'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large'}] The font size of the legend. If the value is numeric the size will be the absolute font size in points. String values are relative to the current default font size. This argument is only used if *prop* is not specified.

labelcolor

[str or list] Sets the color of the text in the legend. Can be a valid color string (for example, 'red'), or a list of color strings. The labelcolor can also be made to match the color of the line or marker using 'linecolor', 'markerfacecolor' (or 'mfc'), or 'markeredgecolor' (or 'mec').

numpoints

[int, default: `rcParams["legend.numpoints"]` (default: `1`)] The number of marker points in the legend when creating a legend entry for a *Line2D* (line).

scatterpoints

[int, default: `rcParams["legend.scatterpoints"]` (default: `1`)] The number of marker points in the legend when creating a legend entry for a *PathCollection* (scatter plot).

scatteryoffsets

[iterable of floats, default: `[0.375, 0.5, 0.3125]`] The vertical offset (relative to the font size) for the markers created for a scatter plot legend entry. 0.0 is at the base the legend text, and 1.0 is at the top. To draw all markers at the same height, set to `[0.5]`.

markerscale

[float, default: `rcParams["legend.markerscale"]` (default: `1.0`)] The relative size of legend markers compared with the originally drawn ones.

markerfirst

[bool, default: True] If *True*, legend marker is placed to the left of the legend label. If *False*, legend marker is placed to the right of the legend label.

frameon

[bool, default: `rcParams["legend.frameon"]` (default: `True`)] Whether the legend should be drawn on a patch (frame).

fancybox

[bool, default: `rcParams["legend.fancybox"]` (default: `True`)] Whether round edges should be enabled around the *FancyBboxPatch* which makes up the legend's background.

shadow

[bool, default: `rcParams["legend.shadow"]` (default: `False`)] Whether to draw a shadow behind the legend.

framealpha

[float, default: `rcParams["legend.framealpha"]` (default: `0.8`)] The alpha transparency of the legend's background. If *shadow* is activated and *framealpha* is `None`, the default value is ignored.

facecolor

["inherit" or color, default: `rcParams["legend.facecolor"]` (default: `'inherit'`)] The legend's background color. If `"inherit"`, use `rcParams["axes.facecolor"]` (default: `'white'`).

edgecolor

["inherit" or color, default: `rcParams["legend.edgecolor"]` (default: `'0.8'`)] The legend's background patch edge color. If `"inherit"`, use take `rcParams["axes.edgecolor"]` (default: `'black'`).

mode

[{"expand", None}] If *mode* is set to `"expand"` the legend will be horizontally expanded to fill the axes area (or *bbox_to_anchor* if defines the legend's size).

bbox_transform

[None or *matplotlib.transforms.Transform*] The transform for the bounding box (*bbox_to_anchor*). For a value of `None` (default) the Axes' `transAxes` transform will be used.

title

[str or None] The legend's title. Default is no title (`None`).

title_fontsize

[int or {'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large'}, default: `rcParams["legend.title_fontsize"]` (default: `None`)] The font size of the legend's title.

borderpad

[float, default: `rcParams["legend.borderpad"]` (default: `0.4`)] The fractional whitespace inside the legend border, in font-size units.

labelspacing

[float, default: `rcParams["legend.labelspacing"]` (default: `0.5`)] The vertical space between the legend entries, in font-size units.

handlelength

[float, default: `rcParams["legend.handlelength"]` (default: `2.0`)]
The length of the legend handles, in font-size units.

handletextpad

[float, default: `rcParams["legend.handletextpad"]` (default: `0.8`)]
The pad between the legend handle and text, in font-size units.

borderaxespad

[float, default: `rcParams["legend.borderaxespad"]` (default: `0.5`)]
The pad between the axes and legend border, in font-size units.

columnspacing

[float, default: `rcParams["legend.columnspacing"]` (default: `2.0`)]
The spacing between columns, in font-size units.

handler_map

[dict or None] The custom dictionary mapping instances or types to a legend handler. This *handler_map* updates the default handler map found at *matplotlib.legend.Legend.get_legend_handler_map*.

Notes

Some artists are not supported by this function. See *Legend guide* for details.

Examples

Examples using `matplotlib.pyplot.legend`

- sphx_glr_gallery_user_interfaces_svg_histogram_sgskip.py
- sphx_glr_gallery_userdemo_pgf_preamble_sgskip.py

matplotlib.pyplot.locator_params

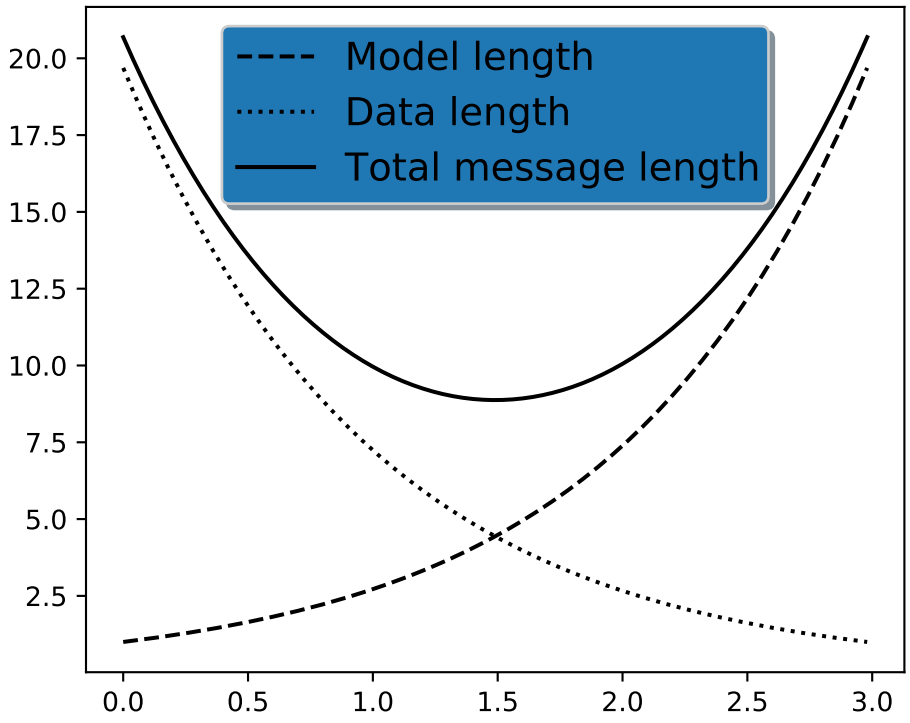
`matplotlib.pyplot.locator_params`(*axis='both'*, *tight=None*, ***kwargs*)
Control behavior of major tick locators.

Because the locator is involved in autoscaling, *autoscale_view* is called automatically after the parameters are changed.

Parameters

axis

[{'both', 'x', 'y'}, default: 'both'] The axis on which to operate.



tight

[bool or None, optional] Parameter passed to *autoscale_view*. Default is None, for no change.

Other Parameters

****kwargs**

Remaining keyword arguments are passed to directly to the `set_params()` method of the locator. Supported keywords depend on the type of the locator. See for example *set_params* for the *ticker.MaxNLocator* used by default for linear axes.

Examples

When plotting small subplots, one might want to reduce the maximum number of ticks and use tight bounds, for example:

```
ax.locator_params(tight=True, nbins=4)
```

Examples using `matplotlib.pyplot.locator_params`

matplotlib.pyplot.loglog

matplotlib.pyplot.**loglog**(**args*, ***kwargs*)

Make a plot with log scaling on both the x and y axis.

Call signatures:

```
loglog([x], y, [fmt], data=None, **kwargs)
loglog([x], y, [fmt], [x2], y2, [fmt2], ..., **kwargs)
```

This is just a thin wrapper around *plot* which additionally changes both the x-axis and the y-axis to log scaling. All of the concepts and parameters of plot can be used here as well.

The additional parameters *base*, *subs* and *nonpositive* control the x/y-axis properties. They are just forwarded to *Axes.set_xscale* and *Axes.set_yscale*. To use different properties on the x-axis and the y-axis, use e.g. `ax.set_xscale("log", base=10); ax.set_yscale("log", base=2)`.

Parameters

base

[float, default: 10] Base of the logarithm.

subs

[sequence, optional] The location of the minor ticks. If *None*, reasonable locations are automatically chosen depending on the number of decades in the plot. See *Axes.set_xscale*/*Axes.set_yscale* for details.

nonpositive

[{'mask', 'clip'}, default: 'mask'] Non-positive values can be masked as invalid, or clipped to a very small positive number.

Returns

lines

A list of *Line2D* objects representing the plotted data.

Other Parameters

****kwargs**

All parameters supported by *plot*.

Examples using `matplotlib.pyplot.loglog`

- *Sample plots in Matplotlib*

matplotlib.pyplot.magma

matplotlib.pyplot.**magma**()

Set the colormap to "magma".

This changes the default colormap as well as the colormap of the current image if there is one. See `help(colormaps)` for more information.

Examples using `matplotlib.pyplot.magma`

matplotlib.pyplot.magnitude_spectrum

matplotlib.pyplot.**magnitude_spectrum**(*x*, *Fs=None*, *Fc=None*, *window=None*, *pad_to=None*, *sides=None*, *scale=None*, ***, *data=None*, ***kwargs*)

Plot the magnitude spectrum.

Compute the magnitude spectrum of *x*. Data is padded to a length of *pad_to* and the windowing function *window* is applied to the signal.

Parameters

x

[1-D array or sequence] Array or sequence containing the data.

Fs

[float, default: 2] The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, *freqs*, in cycles per time unit.

window

[callable or ndarray, default: *window_hanning*] A function or a vector of length *NFFT*. To create window vectors see *window_hanning*, *window_none*, numpy.blackman, numpy.hamming, numpy.bartlett, scipy.signal, scipy.signal.get_window, etc. If a

function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

sides

[{'default', 'onesided', 'twosided'}, optional] Which sides of the spectrum to return. 'default' is one-sided for real data and two-sided for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

pad_to

[int, optional] The number of points to which the data segment is padded when performing the FFT. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to fft(). The default is None, which sets *pad_to* equal to the length of the input signal (i.e. no padding).

scale

[{'default', 'linear', 'dB'}] The scaling of the values in the *spec*. 'linear' is no scaling. 'dB' returns the values in dB scale, i.e., the dB amplitude (20 * log10). 'default' is 'linear'.

Fc

[int, default: 0] The center frequency of *x*, which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

Returns

spectrum

[1-D array] The values for the magnitude spectrum before scaling (real valued).

freqs

[1-D array] The frequencies corresponding to the elements in *spectrum*.

line

[*Line2D*] The line created by this function.

Other Parameters

****kwargs**

Keyword arguments control the *Line2D* properties:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value
alpha	float or None
animated	bool
antialiased or aa	bool
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color or c	color
contains	unknown
dash_capstyle	{'butt', 'round', 'projecting'}
dash_joinstyle	{'miter', 'round', 'bevel'}
dashes	sequence of floats (on/off ink in points) or (None, None)
data	(2, N) array or two 1D arrays
drawstyle or ds	{'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'd
figure	*Figure*
fillstyle	{'full', 'left', 'right', 'bottom', 'top', 'none'}
gid	str
in_layout	bool
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float
marker	marker style string, *Path* or *MarkerStyle*
markeredgecolor or mec	color
markeredgewidth or mew	float
markerfacecolor or mfc	color
markerfacecoloralt or mfcalt	color
markersize or ms	float
markevery	None or int or (int, int) or slice or List[int] or float or (float, float)
path_effects	*AbstractPathEffect*
picker	unknown
pickradius	float
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
solid_capstyle	{'butt', 'round', 'projecting'}
solid_joinstyle	{'miter', 'round', 'bevel'}
transform	*matplotlib.transforms.Transform*
url	str
visible	bool
xdata	1D array
ydata	1D array
zorder	float

See also:

psd

Plots the power spectral density.

angle_spectrum

Plots the angles of the corresponding frequencies.

phase_spectrum

Plots the phase (unwrapped angle) of the corresponding frequencies.

specgram

Can plot the magnitude spectrum of segments within the signal in a colormap.

Notes

Note: In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, the following arguments can also be string `s`, which is interpreted as `data[s]` (unless this raises an exception): *x*.

Objects passed as **data** must support item access (`data[s]`) and membership test (`s in data`).

Examples using `matplotlib.pyplot.magnitude_spectrum`

matplotlib.pyplot.margins

matplotlib.pyplot.**margins**(**margins*, *x=None*, *y=None*, *tight=True*)

Set or retrieve autoscaling margins.

The padding added to each limit of the axes is the *margin* times the data interval. All input parameters must be floats within the range [0, 1]. Passing both positional and keyword arguments is invalid and will raise a TypeError. If no arguments (positional or otherwise) are provided, the current margins will remain in place and simply be returned.

Specifying any margin changes only the autoscaling; for example, if *xmargin* is not None, then *xmargin* times the X data interval will be added to each end of that interval before it is used in autoscaling.

Parameters

***margins**

[float, optional] If a single positional argument is provided, it specifies both margins of the x-axis and y-axis limits. If two positional arguments are provided, they will be interpreted as *xmargin*, *ymargin*. If setting the margin on a single axis is desired, use the keyword arguments described below.

x, y

[float, optional] Specific margin values for the x-axis and y-axis, respectively. These cannot be used with positional arguments, but can be used individually to alter on e.g., only the y-axis.

tight

[bool or None, default: True] The *tight* parameter is passed to `autoscale_view()`, which is executed after a margin is changed; the default here is *True*, on the assumption that when margins are specified, no additional padding to match tick marks is usually desired. Set *tight* to *None* will preserve the previous setting.

Returns

xmargin, ymargin

[float]

Notes

If a previously used Axes method such as *pcolor()* has set `use_sticky_edges` to `True`, only the limits not set by the "sticky artists" will be modified. To force all of the margins to be set, set `use_sticky_edges` to `False` before calling *margins()*.

Examples using `matplotlib.pyplot.margins`

- sphx_glr_gallery_subplots_axes_and_figures_axes_margins.py
- sphx_glr_gallery_ticks_and_spines_ticklabels_rotation.py

matplotlib.pyplot.matshow

`matplotlib.pyplot.`**`matshow`**(*A*, *fignum=None*, ***kwargs*)

Display an array as a matrix in a new figure window.

The origin is set at the upper left hand corner and rows (first dimension of the array) are displayed horizontally. The aspect ratio of the figure window is that of the array, unless this would make an excessively short or narrow figure.

Tick labels for the xaxis are placed on top.

Parameters

A

[array-like(M, N)] The matrix to be displayed.

fignum

[None or int or False] If *None*, create a new figure window with automatic numbering.

If a nonzero integer, draw into the figure with the given number (create it if it does not exist).

If 0, use the current axes (or create one if it does not exist).

Note: Because of how *Axes.matshow* tries to set the figure aspect ratio to be the one of the array, strange things may happen if you reuse an existing figure.

Returns

AxesImage

Other Parameters

****kwargs**

[*imshow* arguments]

Examples using `matplotlib.pyplot.matshow`

- sphx_glr_gallery_images_contours_and_fields_matshow.py

matplotlib.pyplot.minorticks_off

matplotlib.pyplot.**minorticks_off**()
Remove minor ticks from the axes.

Examples using `matplotlib.pyplot.minorticks_off`

matplotlib.pyplot.minorticks_on

matplotlib.pyplot.**minorticks_on**()
Display minor ticks on the axes.

Displaying minor ticks may reduce performance; you may turn them off using *minorticks_off()* if drawing speed is a problem.

Examples using `matplotlib.pyplot.minorticks_on`

matplotlib.pyplot.new_figure_manager

matplotlib.pyplot.**new_figure_manager**(*num*, **args*, ***kwargs*)
Create a new figure manager instance.

Examples using `matplotlib.pyplot.new_figure_manager`

matplotlib.pyplot.nipy_spectral

matplotlib.pyplot.**nipy_spectral**()
Set the colormap to "nipy_spectral".

This changes the default colormap as well as the colormap of the current image if there is one. See `help(colormaps)` for more information.

Examples using `matplotlib.pyplot.nipy_spectral`

matplotlib.pyplot.pause

matplotlib.pyplot.**pause**(*interval*)
Run the GUI event loop for *interval* seconds.

If there is an active figure, it will be updated and displayed before the pause, and the GUI event loop (if any) will run during the pause.

This can be used for crude animation. For more complex animation use *matplotlib.animation*.

If there is no active figure, sleep for *interval* seconds instead.

See also:

matplotlib.animation

Complex animation

show

show figures and optional block forever

Examples using `matplotlib.pyplot.pause`

- sphx_glr_gallery_mplot3d_rotate_axes3d_sgskip.py
- sphx_glr_gallery_mplot3d_wire3d_animation_sgskip.py

matplotlib.pyplot.pcolor

matplotlib.pyplot.**pcolor**(**args*, *shading=None*, *alpha=None*, *norm=None*, *cmap=None*, *vmin=None*, *vmax=None*, *data=None*, ***kwargs*)

Create a pseudocolor plot with a non-regular rectangular grid.

Call signature:

```
pcolor([X, Y,] C, **kwargs)
```

X and *Y* can be used to specify the corners of the quadrilaterals.

Hint: `pcolor()` can be very slow for large arrays. In most cases you should use the similar but much faster *`pcolormesh`* instead. See *Differences between pcolor() and pcolormesh()* for a discussion of the differences.

Parameters

C

[array-like] A scalar 2-D array. The values will be color-mapped.

X, Y

[array-like, optional] The coordinates of the corners of quadrilaterals of a pcolormesh:

```
(X[i+1, j], Y[i+1, j])       (X[i+1, j+1], Y[i+1, j+1])
                      +------+
                      |      |
                      +------+
   (X[i, j], Y[i, j])       (X[i, j+1], Y[i, j+1])
```

Note that the column index corresponds to the x-coordinate, and the row index corresponds to y. For details, see the *Notes* section below.

If `shading='flat'` the dimensions of *X* and *Y* should be one greater than those of *C*, and the quadrilateral is colored due to the value at `C[i, j]`. If *X*, *Y* and *C* have equal dimensions, a warning will be raised and the last row and column of *C* will be ignored.

If `shading='nearest'`, the dimensions of *X* and *Y* should be the same as those of *C* (if not, a ValueError will be raised). The color `C[i, j]` will be centered on `(X[i, j], Y[i, j])`.

If *X* and/or *Y* are 1-D arrays or column vectors they will be expanded as needed into the appropriate 2-D arrays, making a rectangular grid.

shading

[{'flat', 'nearest', 'auto'}, optional] The fill style for the quadrilateral; defaults to 'flat' or `rcParams["pcolor.shading"]` (default: `'flat'`). Possible values:

- 'flat': A solid color is used for each quad. The color of the quad (i, j), (i+1, j), (i, j+1), (i+1, j+1) is given by `C[i, j]`. The dimensions of *X* and *Y* should be one greater than those of *C*; if they are the same as *C*, then a deprecation warning is raised, and the last row and column of *C* are dropped.
- 'nearest': Each grid point will have a color centered on it, extending halfway between the adjacent grid centers. The dimensions of *X* and *Y* must be the same as *C*.
- 'auto': Choose 'flat' if dimensions of *X* and *Y* are one larger than *C*. Choose 'nearest' if dimensions are the same.

See /gallery/images_contours_and_fields/pcolormesh_grids for more description.

cmap

[str or *Colormap*, default: `rcParams["image.cmap"]` (default: `'viridis'`)] A Colormap instance or registered colormap name. The colormap maps the *C* values to colors.

norm

[*Normalize*, optional] The Normalize instance scales the data values to the canonical colormap range [0, 1] for mapping to colors. By default, the data range is mapped to the colorbar range using linear scaling.

vmin, vmax

[float, default: None] The colorbar range. If *None*, suitable min/max values are automatically chosen by the *Normalize* instance (defaults to the respective min/max values of *C* in case of the default linear scaling). It is deprecated to use *vmin*/*vmax* when *norm* is given.

edgecolors

[{'none', None, 'face', color, color sequence}, optional] The color of the edges. Defaults to 'none'. Possible values:

- 'none' or '': No edge.
- *None*: `rcParams["patch.edgecolor"]` (default: `'black'`) will be used. Note that currently `rcParams["patch.force_edgecolor"]` (default: `False`) has to be True for this to work.
- 'face': Use the adjacent face color.
- A color or sequence of colors will set the edge color.

The singular form *edgecolor* works as an alias.

alpha

[float, default: None] The alpha blending value of the face color, between 0 (transparent) and 1 (opaque). Note: The edgecolor is currently not affected by this.

snap

[bool, default: False] Whether to snap the mesh to pixel boundaries.

Returns

matplotlib.collections.Collection

Other Parameters

antialiaseds

[bool, default: False] The default *antialiaseds* is False if the default *edgecolors*="none" is used. This eliminates artificial lines at patch boundaries, and works regardless of the value of alpha. If *edgecolors* is not "none", then the default *antialiaseds* is taken from `rcParams["patch.antialiased"]` (default: `True`). Stroking the edges may be preferred if *alpha* is 1, but will cause artifacts otherwise.

****kwargs**

Additionally, the following arguments are allowed. They are passed along to the *PolyCollection* constructor:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a
alpha	float or None
animated	bool
antialiased or aa or antialiaseds	bool or list of bools
array	ndarray
capstyle	{'butt', 'round', 'projecting'}
clim	(vmin: float, vmax: float)
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
cmap	*Colormap* or str or None
color	color or list of rgba tuples
contains	unknown
edgecolor or ec or edgecolors	color or list of colors or 'face'
facecolor or facecolors or fc	color or list of colors
figure	*Figure*
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or dashes or linestyles or ls	str or tuple or list thereof
linewidth or linewidths or lw	float or list of floats
norm	*Normalize* or None
offset_position	unknown
offsets	array-like (N, 2) or (2,)
path_effects	*AbstractPathEffect*
picker	None or bool or callable
pickradius	unknown
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
urls	list of str or None
visible	bool
zorder	float

See also:

pcolormesh

for an explanation of the differences between pcolor and pcolormesh.

imshow

If X and Y are each equidistant, *imshow* can be a faster alternative.

Notes

Masked arrays

X, Y and C may be masked arrays. If either `C[i, j]`, or one of the vertices surrounding `C[i, j]` (X or Y at `[i, j]`, `[i+1, j]`, `[i, j+1]`, `[i+1, j+1]`) is masked, nothing is plotted.

Grid orientation

The grid orientation follows the standard matrix convention: An array C with shape (nrows, ncolumns) is plotted with the column number as X and the row number as Y.

Note: In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, every other argument can also be string `s`, which is interpreted as `data[s]` (unless this raises an exception).

Objects passed as **data** must support item access (`data[s]`) and membership test (`s in data`).

Examples using `matplotlib.pyplot.pcolor`

- sphx_glr_gallery_images_contours_and_fields_pcolor_demo.py
- sphx_glr_gallery_subplots_axes_and_figures_axes_margins.py

matplotlib.pyplot.pcolormesh

matplotlib.pyplot.**pcolormesh**(**args*, *alpha=None*, *norm=None*, *cmap=None*, *vmin=None*, *vmax=None*, *shading=None*, *antialiased=False*, *data=None*, ***kwargs*)

Create a pseudocolor plot with a non-regular rectangular grid.

Call signature:

```
pcolormesh([X, Y,] C, **kwargs)
```

X and Y can be used to specify the corners of the quadrilaterals.

Hint: *pcolormesh* is similar to *pcolor*. It is much faster and preferred in most cases. For a detailed discussion on the differences see *Differences between*

pcolor() and pcolormesh().

Parameters

C

[array-like] A scalar 2-D array. The values will be color-mapped.

X, Y

[array-like, optional] The coordinates of the corners of quadrilaterals of a pcolormesh:

```
(X[i+1, j], Y[i+1, j])       (X[i+1, j+1], Y[i+1, j+1])
                      +-----+
                      |     |
                      +-----+
   (X[i, j], Y[i, j])       (X[i, j+1], Y[i, j+1])
```

Note that the column index corresponds to the x-coordinate, and the row index corresponds to y. For details, see the *Notes* section below.

If `shading='flat'` the dimensions of *X* and *Y* should be one greater than those of *C*, and the quadrilateral is colored due to the value at `C[i, j]`. If *X*, *Y* and *C* have equal dimensions, a warning will be raised and the last row and column of *C* will be ignored.

If `shading='nearest'` or `'gouraud'`, the dimensions of *X* and *Y* should be the same as those of *C* (if not, a ValueError will be raised). For `'nearest'` the color `C[i, j]` is centered on `(X[i, j], Y[i, j])`. For `'gouraud'`, a smooth interpolation is caried out between the quadrilateral corners.

If *X* and/or *Y* are 1-D arrays or column vectors they will be expanded as needed into the appropriate 2-D arrays, making a rectangular grid.

cmap

[str or *Colormap*, default: `rcParams["image.cmap"]` (default: `'viridis'`)] A Colormap instance or registered colormap name. The colormap maps the *C* values to colors.

norm

[*Normalize*, optional] The Normalize instance scales the data values to the canonical colormap range [0, 1] for mapping to colors. By default, the data range is mapped to the colorbar range using linear scaling.

vmin, vmax

[float, default: None] The colorbar range. If *None*, suitable min/max values are automatically chosen by the *Normalize* instance (defaults to the respective min/max values of *C* in case of the default linear scaling). It is deprecated to use *vmin*/*vmax* when *norm* is given.

edgecolors

[{'none', None, 'face', color, color sequence}, optional] The color of the edges. Defaults to 'none'. Possible values:

- 'none' or '': No edge.
- *None*: `rcParams["patch.edgecolor"]` (default: `'black'`) will be used. Note that currently `rcParams["patch.force_edgecolor"]` (default: `False`) has to be True for this to work.
- 'face': Use the adjacent face color.
- A color or sequence of colors will set the edge color.

The singular form *edgecolor* works as an alias.

alpha

[float, default: None] The alpha blending value, between 0 (transparent) and 1 (opaque).

shading

[{'flat', 'nearest', 'gouraud', 'auto'}, optional] The fill style for the quadrilateral; defaults to 'flat' or `rcParams["pcolor.shading"]` (default: `'flat'`). Possible values:

- 'flat': A solid color is used for each quad. The color of the quad (i, j), (i+1, j), (i, j+1), (i+1, j+1) is given by `C[i, j]`. The dimensions of *X* and *Y* should be one greater than those of *C*; if they are the same as *C*, then a deprecation warning is raised, and the last row and column of *C* are dropped.
- 'nearest': Each grid point will have a color centered on it, extending halfway between the adjacent grid centers. The dimensions of *X* and *Y* must be the same as *C*.
- 'gouraud': Each quad will be Gouraud shaded: The color of the corners (i', j') are given by `C[i', j']`. The color values of the area in between is interpolated from the corner values. The dimensions of *X* and *Y* must be the same as *C*. When Gouraud shading is used, *edgecolors* is ignored.
- 'auto': Choose 'flat' if dimensions of *X* and *Y* are one larger than *C*. Choose 'nearest' if dimensions are the same.

See /gallery/images_contours_and_fields/pcolormesh_grids for more description.

snap

[bool, default: False] Whether to snap the mesh to pixel boundaries.

Returns

matplotlib.collections.QuadMesh

Other Parameters

****kwargs**

Additionally, the following arguments are allowed. They are passed along to the *QuadMesh* constructor:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a
alpha	float or None
animated	bool
antialiased or aa or antialiaseds	bool or list of bools
array	ndarray
capstyle	{'butt', 'round', 'projecting'}
clim	(vmin: float, vmax: float)
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
cmap	*Colormap* or str or None
color	color or list of rgba tuples
contains	unknown
edgecolor or ec or edgecolors	color or list of colors or 'face'
facecolor or facecolors or fc	color or list of colors
figure	*Figure*
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or dashes or linestyles or ls	str or tuple or list thereof
linewidth or linewidths or lw	float or list of floats
norm	*Normalize* or None
offset_position	unknown
offsets	array-like (N, 2) or (2,)
path_effects	*AbstractPathEffect*
picker	None or bool or callable
pickradius	unknown

Table 202 – continued from previous page

Property	Description
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
urls	list of str or None
visible	bool
zorder	float

See also:

pcolor

An alternative implementation with slightly different features. For a detailed discussion on the differences see *Differences between pcolor() and pcolormesh()*.

imshow

If X and Y are each equidistant, *imshow* can be a faster alternative.

Notes

Masked arrays

C may be a masked array. If `C[i, j]` is masked, the corresponding quadrilateral will be transparent. Masking of X and Y is not supported. Use *pcolor* if you need this functionality.

Grid orientation

The grid orientation follows the standard matrix convention: An array C with shape (nrows, ncolumns) is plotted with the column number as X and the row number as Y.

Differences between pcolor() and pcolormesh()

Both methods are used to create a pseudocolor plot of a 2-D array using quadrilaterals.

The main difference lies in the created object and internal data handling: While *pcolor* returns a *PolyCollection*, *pcolormesh* returns a *QuadMesh*. The latter is more specialized for the given purpose and thus is faster. It should almost always be preferred.

There is also a slight difference in the handling of masked arrays. Both *pcolor* and *pcolormesh* support masked arrays for C. However, only *pcolor* supports masked arrays for X and Y. The reason lies in the internal handling of the

masked values. *pcolor* leaves out the respective polygons from the PolyCollection. *pcolormesh* sets the facecolor of the masked elements to transparent. You can see the difference when using edgecolors. While all edges are drawn irrespective of masking in a QuadMesh, the edge between two adjacent masked quadrilaterals in *pcolor* is not drawn as the corresponding polygons do not exist in the PolyCollection.

Another difference is the support of Gouraud shading in *pcolormesh*, which is not available with *pcolor*.

Note: In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, every other argument can also be string `s`, which is interpreted as `data[s]` (unless this raises an exception).

Objects passed as **data** must support item access (`data[s]`) and membership test (`s in data`).

Examples using `matplotlib.pyplot.pcolormesh`

- sphx_glr_gallery_images_contours_and_fields_pcolor_demo.py
- sphx_glr_gallery_images_contours_and_fields_pcolormesh_grids.py
- sphx_glr_gallery_images_contours_and_fields_pcolormesh_levels.py
- sphx_glr_gallery_images_contours_and_fields_quadmesh_demo.py
- *Sample plots in Matplotlib*

matplotlib.pyplot.phase_spectrum

matplotlib.pyplot.**phase_spectrum**(*x*, *Fs=None*, *Fc=None*, *window=None*, *pad_to=None*, *sides=None*, ***, *data=None*, ***kwargs*)

Plot the phase spectrum.

Compute the phase spectrum (unwrapped angle spectrum) of *x*. Data is padded to a length of *pad_to* and the windowing function *window* is applied to the signal.

Parameters

x

[1-D array or sequence] Array or sequence containing the data

Fs

[float, default: 2] The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, *freqs*, in cycles per time unit.

window

[callable or ndarray, default: *window_hanning*] A function or a vector of length *NFFT*. To create window vectors see *window_hanning*, *window_none*, numpy.blackman, numpy.hamming, numpy.bartlett, scipy.signal, scipy.signal.get_window, etc. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

sides

[{'default', 'onesided', 'twosided'}, optional] Which sides of the spectrum to return. 'default' is one-sided for real data and two-sided for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

pad_to

[int, optional] The number of points to which the data segment is padded when performing the FFT. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to fft(). The default is None, which sets *pad_to* equal to the length of the input signal (i.e. no padding).

Fc

[int, default: 0] The center frequency of *x*, which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

Returns

spectrum

[1-D array] The values for the phase spectrum in radians (real valued).

freqs

[1-D array] The frequencies corresponding to the elements in *spectrum*.

line

[*Line2D*] The line created by this function.

Other Parameters

****kwargs**

Keyword arguments control the *Line2D* properties:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value
alpha	float or None
animated	bool
antialiased or aa	bool
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color or c	color
contains	unknown
dash_capstyle	{'butt', 'round', 'projecting'}
dash_joinstyle	{'miter', 'round', 'bevel'}
dashes	sequence of floats (on/off ink in points) or (None, None)
data	(2, N) array or two 1D arrays
drawstyle or ds	{'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'd
figure	*Figure*
fillstyle	{'full', 'left', 'right', 'bottom', 'top', 'none'}
gid	str
in_layout	bool
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float
marker	marker style string, *Path* or *MarkerStyle*
markeredgecolor or mec	color
markeredgewidth or mew	float
markerfacecolor or mfc	color
markerfacecoloralt or mfcalt	color
markersize or ms	float
markevery	None or int or (int, int) or slice or List[int] or float or (float, float)
path_effects	*AbstractPathEffect*
picker	unknown
pickradius	float
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
solid_capstyle	{'butt', 'round', 'projecting'}
solid_joinstyle	{'miter', 'round', 'bevel'}
transform	*matplotlib.transforms.Transform*
url	str
visible	bool
xdata	1D array
ydata	1D array

Table 203 – continued from previous page

Property	Description
zorder	float

See also:

magnitude_spectrum

Plots the magnitudes of the corresponding frequencies.

angle_spectrum

Plots the wrapped version of this function.

specgram

Can plot the phase spectrum of segments within the signal in a colormap.

Notes

Note: In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, the following arguments can also be string `s`, which is interpreted as `data[s]` (unless this raises an exception): *x*.

Objects passed as **data** must support item access (`data[s]`) and membership test (`s in data`).

Examples using `matplotlib.pyplot.phase_spectrum`

matplotlib.pyplot.pie

`matplotlib.pyplot.pie`(*x*, *explode=None*, *labels=None*, *colors=None*, *autopct=None*, *pctdistance=0.6*, *shadow=False*, *labeldistance=1.1*, *startangle=0*, *radius=1*, *counterclock=True*, *wedgeprops=None*, *textprops=None*, *center=0, 0*, *frame=False*, *rotatelabels=False*, ***, *normalize=None*, *data=None*)

Plot a pie chart.

Make a pie chart of array *x*. The fractional area of each wedge is given by `x/sum(x)`. If `sum(x) < 1`, then the values of *x* give the fractional area directly and the array will not be normalized. The resulting pie will have an empty wedge of size `1 - sum(x)`.

The wedges are plotted counterclockwise, by default starting from the x-axis.

Parameters

x

[1D array-like] The wedge sizes.

explode

[array-like, default: None] If not *None*, is a `len(x)` array which specifies the fraction of the radius with which to offset each wedge.

labels

[list, default: None] A sequence of strings providing the labels for each wedge

colors

[array-like, default: None] A sequence of colors through which the pie chart will cycle. If *None*, will use the colors in the currently active cycle.

autopct

[None or str or callable, default: None] If not *None*, is a string or function used to label the wedges with their numeric value. The label will be placed inside the wedge. If it is a format string, the label will be `fmt % pct`. If it is a function, it will be called.

pctdistance

[float, default: 0.6] The ratio between the center of each pie slice and the start of the text generated by *autopct*. Ignored if *autopct* is *None*.

shadow

[bool, default: False] Draw a shadow beneath the pie.

normalize: None or bool, default: None

When *True*, always make a full pie by normalizing x so that `sum(x) == 1`. *False* makes a partial pie if `sum(x) <= 1` and raises a `ValueError` for `sum(x) > 1`.

When *None*, defaults to *True* if `sum(x) >= 1` and *False* if `sum(x) < 1`.

Please note that the previous default value of *None* is now deprecated, and the default will change to *True* in the next release. Please pass `normalize=False` explicitly if you want to draw a partial pie.

labeldistance

[float or None, default: 1.1] The radial distance at which the pie labels are drawn. If set to `None`, label are not drawn, but are stored for use in `legend()`

startangle

[float, default: 0 degrees] The angle by which the start of the pie is rotated, counterclockwise from the x-axis.

radius

[float, default: 1] The radius of the pie.

counterclock

[bool, default: True] Specify fractions direction, clockwise or counterclockwise.

wedgeprops

[dict, default: None] Dict of arguments passed to the wedge objects making the pie. For example, you can pass in `wedgeprops = {'linewidth': 3}` to set the width of the wedge border lines equal to 3. For more details, look at the doc/arguments of the wedge object. By default `clip_on=False`.

textprops

[dict, default: None] Dict of arguments to pass to the text objects.

center

[(float, float), default: (0, 0)] The coordinates of the center of the chart.

frame

[bool, default: False] Plot axes frame with the chart if true.

rotatelabels

[bool, default: False] Rotate each label to the angle of the corresponding slice if true.

Returns

patches

[list] A sequence of *matplotlib.patches.Wedge* instances

texts

[list] A list of the label *Text* instances.

autotexts

[list] A list of *Text* instances for the numeric labels. This will only be returned if the parameter *autopct* is not *None*.

Notes

The pie chart will probably look best if the figure and axes are square, or the Axes aspect is equal. This method sets the aspect ratio of the axis to "equal". The axes aspect ratio can be controlled with *Axes.set_aspect*.

Note: In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, the following arguments can also be string `s`, which is interpreted as `data[s]` (unless this raises an exception): *x*, *explode*, *labels*, *colors*.

Objects passed as **data** must support item access (`data[s]`) and membership test (`s in data`).

Examples using `matplotlib.pyplot.pie`

- sphx_glr_gallery_pie_and_polar_charts_pie_features.py
- sphx_glr_gallery_pie_and_polar_charts_pie_demo2.py
- sphx_glr_gallery_pie_and_polar_charts_nested_pie.py
- sphx_glr_gallery_pie_and_polar_charts_pie_and_donut_labels.py
- *Sample plots in Matplotlib*

matplotlib.pyplot.pink

matplotlib.pyplot.**pink**()
Set the colormap to "pink".

This changes the default colormap as well as the colormap of the current image if there is one. See `help(colormaps)` for more information.

Examples using `matplotlib.pyplot.pink`

matplotlib.pyplot.plasma

matplotlib.pyplot.**plasma**()
Set the colormap to "plasma".

This changes the default colormap as well as the colormap of the current image if there is one. See `help(colormaps)` for more information.

Examples using `matplotlib.pyplot.plasma`

matplotlib.pyplot.plot

matplotlib.pyplot.plot(**args*, *scalex=True*, *scaley=True*, *data=None*, ***kwargs*)

Plot y versus x as lines and/or markers.

Call signatures:

```
plot([x], y, [fmt], *, data=None, **kwargs)
plot([x], y, [fmt], [x2], y2, [fmt2], ..., **kwargs)
```

The coordinates of the points or line nodes are given by *x*, *y*.

The optional parameter *fmt* is a convenient way for defining basic formatting like color, marker and linestyle. It's a shortcut string notation described in the *Notes* section below.

```
>>> plot(x, y)        # plot x and y using default line style and color
>>> plot(x, y, 'bo')  # plot x and y using blue circle markers
>>> plot(y)           # plot y using x as index array 0..N-1
>>> plot(y, 'r+')     # ditto, but with red plusses
```

You can use *Line2D* properties as keyword arguments for more control on the appearance. Line properties and *fmt* can be mixed. The following two calls yield identical results:

```
>>> plot(x, y, 'go--', linewidth=2, markersize=12)
>>> plot(x, y, color='green', marker='o', linestyle='dashed',
...      linewidth=2, markersize=12)
```

When conflicting with *fmt*, keyword arguments take precedence.

Plotting labelled data

There's a convenient way for plotting objects with labelled data (i.e. data that can be accessed by index `obj['y']`). Instead of giving the data in *x* and *y*, you can provide the object in the *data* parameter and just give the labels for *x* and *y*:

```
>>> plot('xlabel', 'ylabel', data=obj)
```

All indexable objects are supported. This could e.g. be a `dict`, a `pandas.DataFrame` or a structured numpy array.

Plotting multiple sets of data

There are various ways to plot multiple sets of data.

- The most straight forward way is just to call *plot* multiple times. Example:

```
>>> plot(x1, y1, 'bo')
>>> plot(x2, y2, 'go')
```

- Alternatively, if your data is already a 2d array, you can pass it directly to *x*, *y*. A separate data set will be drawn for every column.

  Example: an array `a` where the first column represents the *x* values and the other columns are the *y* columns:

```
>>> plot(a[0], a[1:])
```

- The third way is to specify multiple sets of *[x]*, *y*, *[fmt]* groups:

```
>>> plot(x1, y1, 'g^', x2, y2, 'g-')
```

  In this case, any additional keyword argument applies to all datasets. Also this syntax cannot be combined with the *data* parameter.

By default, each line is assigned a different style specified by a 'style cycle'. The *fmt* and line property parameters are only necessary if you want explicit deviations from these defaults. Alternatively, you can also change the style cycle using `rcParams["axes.prop_cycle"]` (default: `cycler('color', ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728', '#9467bd', '#8c564b', '#e377c2', '#7f7f7f', '#bcbd22', '#17becf'])`).

Parameters

x, y

[array-like or scalar] The horizontal / vertical coordinates of the data points. *x* values are optional and default to `range(len(y))`.

Commonly, these parameters are 1D arrays.

They can also be scalars, or two-dimensional (in that case, the columns represent separate data sets).

These arguments cannot be passed as keywords.

fmt

[str, optional] A format string, e.g. 'ro' for red circles. See the *Notes* section for a full description of the format strings.

Format strings are just an abbreviation for quickly setting basic line properties. All of these and more can also be controlled by keyword arguments.

This argument cannot be passed as keyword.

data

[indexable object, optional] An object with labelled data. If given, provide the label names to plot in *x* and *y*.

Note: Technically there's a slight ambiguity in calls where the second label is a valid *fmt*. `plot('n', 'o', data=obj)` could be

`plt(x, y)` or `plt(y, fmt)`. In such cases, the former interpretation is chosen, but a warning is issued. You may suppress the warning by adding an empty format string `plot('n', 'o', '', data=obj)`.

Returns

list of *Line2D*

A list of lines representing the plotted data.

Other Parameters

scalex, scaley

[bool, default: True] These parameters determine if the view limits are adapted to the data limits. The values are passed on to `autoscale_view`.

****kwargs**

[*Line2D* properties, optional] *kwargs* are used to specify properties like a line label (for auto legends), linewidth, antialiasing, marker face color. Example:

```
>>> plot([1, 2, 3], [1, 2, 3], 'go-', label='line 1', linewidth=2)
>>> plot([1, 2, 3], [1, 4, 9], 'rs', label='line 2')
```

If you make multiple lines with one plot call, the kwargs apply to all those lines.

Here is a list of available *Line2D* properties:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value
alpha	float or None
animated	bool
antialiased or aa	bool
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color or c	color
contains	unknown
dash_capstyle	{'butt', 'round', 'projecting'}
dash_joinstyle	{'miter', 'round', 'bevel'}
dashes	sequence of floats (on/off ink in points) or (None, None)
data	(2, N) array or two 1D arrays
drawstyle or ds	{'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'd

Table 204 – continued from previous page

Property	Description
figure	*Figure*
fillstyle	{'full', 'left', 'right', 'bottom', 'top', 'none'}
gid	str
in_layout	bool
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float
marker	marker style string, *Path* or *MarkerStyle*
markeredgecolor or mec	color
markeredgewidth or mew	float
markerfacecolor or mfc	color
markerfacecoloralt or mfcalt	color
markersize or ms	float
markevery	None or int or (int, int) or slice or List[int] or float or (float, float)
path_effects	*AbstractPathEffect*
picker	unknown
pickradius	float
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
solid_capstyle	{'butt', 'round', 'projecting'}
solid_joinstyle	{'miter', 'round', 'bevel'}
transform	*matplotlib.transforms.Transform*
url	str
visible	bool
xdata	1D array
ydata	1D array
zorder	float

See also:

scatter

XY scatter plot with markers of varying size and/or color (sometimes also called bubble chart).

Notes

Format Strings

A format string consists of a part for color, marker and line:

```
fmt = '[marker][line][color]'
```

Each of them is optional. If not provided, the value from the style cycle is used. Exception: If `line` is given, but no `marker`, the data will be a line without markers.

Other combinations such as `[color][marker][line]` are also supported, but note that their parsing may be ambiguous.

Markers

character	description
`'.'`	point marker
`','`	pixel marker
`'o'`	circle marker
`'v'`	triangle_down marker
`'^'`	triangle_up marker
`'<'`	triangle_left marker
`'>'`	triangle_right marker
`'1'`	tri_down marker
`'2'`	tri_up marker
`'3'`	tri_left marker
`'4'`	tri_right marker
`'s'`	square marker
`'p'`	pentagon marker
`'*'`	star marker
`'h'`	hexagon1 marker
`'H'`	hexagon2 marker
`'+'`	plus marker
`'x'`	x marker
`'D'`	diamond marker
`'d'`	thin_diamond marker
`'|'`	vline marker
`'_'`	hline marker

Line Styles

character	description
`'-'`	solid line style
`'--'`	dashed line style
`'-.'`	dash-dot line style
`':'`	dotted line style

Example format strings:

```
'b'    # blue markers with default shape
'or'   # red circles
'-g'   # green solid line
'--'   # dashed line with default color
'^k:'  # black triangle_up markers connected by a dotted line
```

Colors

The supported color abbreviations are the single letter codes

character	color
'b'	blue
'g'	green
'r'	red
'c'	cyan
'm'	magenta
'y'	yellow
'k'	black
'w'	white

and the 'CN' colors that index into the default property cycle.

If the color is the only part of the format string, you can additionally use any *matplotlib.colors* spec, e.g. full names ('green') or hex strings ('#008000').

Examples using matplotlib.pyplot.plot

- *Frame grabbing*
- sphx_glr_gallery_misc_print_stdout_sgskip.py
- sphx_glr_gallery_user_interfaces_toolmanager_sgskip.py
- sphx_glr_gallery_userdemo_pgf_preamble_sgskip.py

matplotlib.pyplot.plot_date

matplotlib.pyplot.**plot_date**(*x*, *y*, *fmt='o'*, *tz=None*, *xdate=True*, *ydate=False*, ***, *data=None*, ***kwargs*)

Plot data that contains dates.

Similar to *plot*, this plots *y* vs. *x* as lines or markers. However, the axis labels are formatted as dates depending on *xdate* and *ydate*.

Parameters

x, y

[array-like] The coordinates of the data points. If *xdate* or *ydate* is *True*, the respective values *x* or *y* are interpreted as *Matplotlib dates*.

fmt

[str, optional] The plot format string. For details, see the corresponding parameter in *plot*.

tz

[timezone string or `datetime.tzinfo`, default: `rcParams["timezone"]` (default: `'UTC'`)] The time zone to use in labeling dates.

xdate

[bool, default: True] If *True*, the *x*-axis will be interpreted as Matplotlib dates.

ydate

[bool, default: False] If *True*, the *y*-axis will be interpreted as Matplotlib dates.

Returns

lines

A list of *Line2D* objects representing the plotted data.

Other Parameters

****kwargs**

Keyword arguments control the *Line2D* properties:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value
alpha	float or None
animated	bool
antialiased or aa	bool
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color or c	color
contains	unknown
dash_capstyle	{'butt', 'round', 'projecting'}
dash_joinstyle	{'miter', 'round', 'bevel'}

Table 205 – continued from previous page

Property	Description
dashes	sequence of floats (on/off ink in points) or (None, None)
data	(2, N) array or two 1D arrays
drawstyle or ds	{'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'd
figure	*Figure*
fillstyle	{'full', 'left', 'right', 'bottom', 'top', 'none'}
gid	str
in_layout	bool
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float
marker	marker style string, *Path* or *MarkerStyle*
markeredgecolor or mec	color
markeredgewidth or mew	float
markerfacecolor or mfc	color
markerfacecoloralt or mfcalt	color
markersize or ms	float
markevery	None or int or (int, int) or slice or List[int] or float or (float, float)
path_effects	*AbstractPathEffect*
picker	unknown
pickradius	float
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
solid_capstyle	{'butt', 'round', 'projecting'}
solid_joinstyle	{'miter', 'round', 'bevel'}
transform	*matplotlib.transforms.Transform*
url	str
visible	bool
xdata	1D array
ydata	1D array
zorder	float

See also:

matplotlib.dates

Helper functions on dates.

matplotlib.dates.date2num

Convert dates to num.

matplotlib.dates.num2date

Convert num to dates.

matplotlib.dates.drange

Create an equally spaced sequence of dates.

Notes

If you are using custom date tickers and formatters, it may be necessary to set the formatters/locators after the call to *plot_date*. *plot_date* will set the default tick locator to *AutoDateLocator* (if the tick locator is not already set to a *DateLocator* instance) and the default tick formatter to *AutoDateFormatter* (if the tick formatter is not already set to a *DateFormatter* instance).

Note: In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, the following arguments can also be string `s`, which is interpreted as `data[s]` (unless this raises an exception): *x*, *y*.

Objects passed as **data** must support item access (`data[s]`) and membership test (`s in data`).

Examples using `matplotlib.pyplot.plot_date`

- sphx_glr_gallery_ticks_and_spines_date_demo_rrule.py

matplotlib.pyplot.polar

matplotlib.pyplot.**polar**(**args*, ***kwargs*)

Make a polar plot.

call signature:

```
polar(theta, r, **kwargs)
```

Multiple *theta*, *r* arguments are supported, with format strings, as in *plot*.

Examples using `matplotlib.pyplot.polar`

- sphx_glr_gallery_misc_transoffset.py
- *Sample plots in Matplotlib*

matplotlib.pyplot.prism

matplotlib.pyplot.**prism**()

Set the colormap to "prism".

This changes the default colormap as well as the colormap of the current image if there is one. See `help(colormaps)` for more information.

Examples using `matplotlib.pyplot.prism`

matplotlib.pyplot.psd

matplotlib.pyplot.**psd**(*x*, *NFFT=None*, *Fs=None*, *Fc=None*, *detrend=None*, *window=None*, *noverlap=None*, *pad_to=None*, *sides=None*, *scale_by_freq=None*, *return_line=None*, ***, *data=None*, ***kwargs*)

Plot the power spectral density.

The power spectral density P_{xx} by Welch's average periodogram method. The vector *x* is divided into *NFFT* length segments. Each segment is detrended by function *detrend* and windowed by function *window*. *noverlap* gives the length of the overlap between segments. The $|\mathrm{fft}(i)|^2$ of each segment i are averaged to compute P_{xx}, with a scaling to correct for power loss due to windowing.

If len(*x*) < *NFFT*, it will be zero padded to *NFFT*.

Parameters

x

[1-D array or sequence] Array or sequence containing the data

Fs

[float, default: 2] The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, *freqs*, in cycles per time unit.

window

[callable or ndarray, default: *window_hanning*] A function or a vector of length *NFFT*. To create window vectors see *window_hanning*, *window_none*, `numpy.blackman`, `numpy.hamming`, `numpy.bartlett`, `scipy.signal`, `scipy.signal.get_window`, etc. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

sides

[{'default', 'onesided', 'twosided'}, optional] Which sides of the spectrum to return. 'default' is one-sided for real data and two-

sided for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

pad_to

[int, optional] The number of points to which the data segment is padded when performing the FFT. This can be different from *NFFT*, which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to fft(). The default is None, which sets *pad_to* equal to *NFFT*

NFFT

[int, default: 256] The number of data points used in each block for the FFT. A power 2 is most efficient. This should *NOT* be used to get zero padding, or the scaling of the result will be incorrect; use *pad_to* for this instead.

detrend

[{'none', 'mean', 'linear'} or callable, default 'none'] The function applied to each segment before fft-ing, designed to remove the mean or linear trend. Unlike in MATLAB, where the *detrend* parameter is a vector, in Matplotlib is it a function. The *mlab* module defines *detrend_none*, *detrend_mean*, and *detrend_linear*, but you can use a custom function as well. You can also use a string to choose one of the functions: 'none' calls *detrend_none*. 'mean' calls *detrend_mean*. 'linear' calls *detrend_linear*.

scale_by_freq

[bool, default: True] Whether the resulting density values should be scaled by the scaling frequency, which gives density in units of Hz^-1. This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

noverlap

[int, default: 0 (no overlap)] The number of points of overlap between segments.

Fc

[int, default: 0] The center frequency of *x*, which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

return_line

[bool, default: False] Whether to include the line object plotted in the returned values.

Returns

Pxx

[1-D array] The values for the power spectrum P_{xx} before scaling (real valued).

freqs

[1-D array] The frequencies corresponding to the elements in *Pxx*.

line

[*Line2D*] The line created by this function. Only returned if *return_line* is True.

Other Parameters

**kwargs

Keyword arguments control the *Line2D* properties:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value
alpha	float or None
animated	bool
antialiased or aa	bool
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color or c	color
contains	unknown
dash_capstyle	{'butt', 'round', 'projecting'}
dash_joinstyle	{'miter', 'round', 'bevel'}
dashes	sequence of floats (on/off ink in points) or (None, None)
data	(2, N) array or two 1D arrays
drawstyle or ds	{'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'd
figure	*Figure*
fillstyle	{'full', 'left', 'right', 'bottom', 'top', 'none'}
gid	str
in_layout	bool
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float
marker	marker style string, *Path* or *MarkerStyle*
markeredgecolor or mec	color
markeredgewidth or mew	float

Table 206 – continued from previous page

Property	Description
markerfacecolor or mfc	color
markerfacecoloralt or mfcalt	color
markersize or ms	float
markevery	None or int or (int, int) or slice or List[int] or float or (float, float)
path_effects	*AbstractPathEffect*
picker	unknown
pickradius	float
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
solid_capstyle	{'butt', 'round', 'projecting'}
solid_joinstyle	{'miter', 'round', 'bevel'}
transform	*matplotlib.transforms.Transform*
url	str
visible	bool
xdata	1D array
ydata	1D array
zorder	float

See also:

specgram

Differs in the default overlap; in not returning the mean of the segment periodograms; in returning the times of the segments; and in plotting a colormap instead of a line.

magnitude_spectrum

Plots the magnitude spectrum.

csd

Plots the spectral density between two signals.

Notes

For plotting, the power is plotted as $10\log_{10}(P_{xx})$ for decibels, though *Pxx* itself is returned.

References

Bendat & Piersol -- Random Data: Analysis and Measurement Procedures, John Wiley & Sons (1986)

Note: In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, the following arguments can also be string `s`, which is interpreted as `data[s]` (unless this raises an exception): *x*.

Objects passed as **data** must support item access (`data[s]`) and membership test (`s in data`).

Examples using `matplotlib.pyplot.psd`

- sphx_glr_gallery_lines_bars_and_markers_psd_demo.py

matplotlib.pyplot.quiver

matplotlib.pyplot.**quiver**(**args*, *data=None*, ***kw*)

Plot a 2D field of arrows.

Call signature:

```
quiver([X, Y], U, V, [C], **kw)
```

X, *Y* define the arrow locations, *U*, *V* define the arrow directions, and *C* optionally sets the color.

Arrow size

The default settings auto-scales the length of the arrows to a reasonable size. To change this behavior see the *scale* and *scale_units* parameters.

Arrow shape

The defaults give a slightly swept-back arrow; to make the head a triangle, make *headaxislength* the same as *headlength*. To make the arrow more pointed, reduce *headwidth* or increase *headlength* and *headaxislength*. To make the head smaller relative to the shaft, scale down all the head parameters. You will probably do best to leave minshaft alone.

Arrow outline

linewidths and *edgecolors* can be used to customize the arrow outlines.

Parameters

X, Y

[1D or 2D array-like, optional] The x and y coordinates of the arrow locations.

If not given, they will be generated as a uniform integer meshgrid based on the dimensions of *U* and *V*.

If *X* and *Y* are 1D but *U*, *V* are 2D, *X*, *Y* are expanded to 2D using `X, Y = np.meshgrid(X, Y)`. In this case `len(X)` and `len(Y)` must match the column and row dimensions of *U* and *V*.

U, V

[1D or 2D array-like] The x and y direction components of the arrow vectors.

They must have the same number of elements, matching the number of arrow locations. *U* and *V* may be masked. Only locations unmasked in *U*, *V*, and *C* will be drawn.

C

[1D or 2D array-like, optional] Numeric data that defines the arrow colors by colormapping via *norm* and *cmap*.

This does not support explicit colors. If you want to set colors directly, use *color* instead. The size of *C* must match the number of arrow locations.

units

[{'width', 'height', 'dots', 'inches', 'x', 'y' 'xy'}, default: 'width'] The arrow dimensions (except for *length*) are measured in multiples of this unit.

The following values are supported:

- 'width', 'height': The width or height of the axis.
- 'dots', 'inches': Pixels or inches based on the figure dpi.
- 'x', 'y', 'xy': *X*, *Y* or $\sqrt{X^2 + Y^2}$ in data units.

The arrows scale differently depending on the units. For 'x' or 'y', the arrows get larger as one zooms in; for other units, the arrow size is independent of the zoom state. For 'width or 'height', the arrow size increases with the width and height of the axes, respectively, when the window is resized; for 'dots' or 'inches', resizing does not change the arrows.

angles

[{'uv', 'xy'} or array-like, default: 'uv'] Method for determining the angle of the arrows.

- 'uv': The arrow axis aspect ratio is 1 so that if $U == V$ the orientation of the arrow on the plot is 45 degrees counter-clockwise from the horizontal axis (positive to the right).

  Use this if the arrows symbolize a quantity that is not based on X, Y data coordinates.

- 'xy': Arrows point from (x, y) to (x+u, y+v). Use this for plotting a gradient field, for example.

- Alternatively, arbitrary angles may be specified explicitly as an array of values in degrees, counter-clockwise from the horizontal axis.

  In this case U, V is only used to determine the length of the arrows.

Note: inverting a data axis will correspondingly invert the arrows only with `angles='xy'`.

scale

[float, optional] Number of data units per arrow length unit, e.g., m/s per plot width; a smaller scale parameter makes the arrow longer. Default is *None*.

If *None*, a simple autoscaling algorithm is used, based on the average vector length and the number of vectors. The arrow length unit is given by the *scale_units* parameter.

scale_units

[{'width', 'height', 'dots', 'inches', 'x', 'y', 'xy'}, optional] If the *scale* kwarg is *None*, the arrow length unit. Default is *None*.

e.g. *scale_units* is 'inches', *scale* is 2.0, and `(u, v) = (1, 0)`, then the vector will be 0.5 inches long.

If *scale_units* is 'width' or 'height', then the vector will be half the width/height of the axes.

If *scale_units* is 'x' then the vector will be 0.5 x-axis units. To plot vectors in the x-y plane, with u and v having the same units as x and y, use `angles='xy', scale_units='xy', scale=1`.

width

[float, optional] Shaft width in arrow units; default depends on choice of units, above, and number of vectors; a typical starting value is about 0.005 times the width of the plot.

headwidth

[float, default: 3] Head width as multiple of shaft width.

headlength

[float, default: 5] Head length as multiple of shaft width.

headaxislength

[float, default: 4.5] Head length at shaft intersection.

minshaft

[float, default: 1] Length below which arrow scales, in units of head length. Do not set this to less than 1, or small arrows will look terrible!

minlength

[float, default: 1] Minimum length as a multiple of shaft width; if an arrow length is less than this, plot a dot (hexagon) of this diameter instead.

pivot

[{'tail', 'mid', 'middle', 'tip'}, default: 'tail'] The part of the arrow that is anchored to the *X*, *Y* grid. The arrow rotates about this point.

'mid' is a synonym for 'middle'.

color

[color or color sequence, optional] Explicit color(s) for the arrows. If *C* has been set, *color* has no effect.

This is a synonym for the *PolyCollection* *facecolor* parameter.

Other Parameters

****kwargs**

[*PolyCollection* properties, optional] All other keyword arguments are passed on to *PolyCollection*:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a
alpha	float or None
animated	bool
antialiased or aa or antialiaseds	bool or list of bools
array	ndarray
capstyle	{'butt', 'round', 'projecting'}
clim	(vmin: float, vmax: float)
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
cmap	*Colormap* or str or None

Table 207 – continued from previous page

Property	Description
color	color or list of rgba tuples
contains	unknown
edgecolor or ec or edgecolors	color or list of colors or 'face'
facecolor or facecolors or fc	color or list of colors
figure	*Figure*
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or dashes or linestyles or ls	str or tuple or list thereof
linewidth or linewidths or lw	float or list of floats
norm	*Normalize* or None
offset_position	unknown
offsets	array-like (N, 2) or (2,)
path_effects	*AbstractPathEffect*
picker	None or bool or callable
pickradius	unknown
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
urls	list of str or None
visible	bool
zorder	float

See also:

Axes.quiverkey

Add a key to a quiver plot.

Examples using `matplotlib.pyplot.quiver`

- sphx_glr_gallery_images_contours_and_fields_quiver_demo.py
- sphx_glr_gallery_images_contours_and_fields_quiver_simple_demo.py
- sphx_glr_gallery_images_contours_and_fields_trigradient_demo.py
- *Sample plots in Matplotlib*

matplotlib.pyplot.quiverkey

matplotlib.pyplot.**quiverkey**(*Q*, *X*, *Y*, *U*, *label*, ***kw*)

Add a key to a quiver plot.

The positioning of the key depends on *X*, *Y*, *coordinates*, and *labelpos*. If *labelpos* is 'N' or 'S', *X*, *Y* give the position of the middle of the key arrow. If *labelpos* is 'E', *X*, *Y* positions the head, and if *labelpos* is 'W', *X*, *Y* positions the tail; in either of these two cases, *X*, *Y* is somewhere in the middle of the arrow+label key object.

Parameters

Q

[*matplotlib.quiver.Quiver*] A *Quiver* object as returned by a call to *quiver()*.

X, Y

[float] The location of the key.

U

[float] The length of the key.

label

[str] The key label (e.g., length and units of the key).

angle

[float, default: 0] The angle of the key arrow, in degrees anti-clockwise from the x-axis.

coordinates

[{'axes', 'figure', 'data', 'inches'}, default: 'axes'] Coordinate system and units for *X*, *Y*: 'axes' and 'figure' are normalized coordinate systems with (0, 0) in the lower left and (1, 1) in the upper right; 'data' are the axes data coordinates (used for the locations of the vectors in the quiver plot itself); 'inches' is position in the figure in inches, with (0, 0) at the lower left corner.

color

[color] Overrides face and edge colors from *Q*.

labelpos

[{'N', 'S', 'E', 'W'}] Position the label above, below, to the right, to the left of the arrow, respectively.

labelsep

[float, default: 0.1] Distance in inches between the arrow and the label.

labelcolor

[color, default: `rcParams["text.color"]` (default: `'black'`)] Label color.

fontproperties

[dict, optional] A dictionary with keyword arguments accepted by the *FontProperties* initializer: *family*, *style*, *variant*, *size*, *weight*.

****kwargs**

Any additional keyword arguments are used to override vector properties taken from *Q*.

Examples using `matplotlib.pyplot.quiverkey`

- sphx_glr_gallery_images_contours_and_fields_quiver_demo.py
- sphx_glr_gallery_images_contours_and_fields_quiver_simple_demo.py

matplotlib.pyplot.rc

matplotlib.pyplot.**rc**(*group*, ***kwargs*)

Set the current *rcParams*. *group* is the grouping for the rc, e.g., for `lines.linewidth` the group is `lines`, for `axes.facecolor`, the group is `axes`, and so on. Group may also be a list or tuple of group names, e.g., (*xtick*, *ytick*). *kwargs* is a dictionary attribute name/value pairs, e.g.,:

```
rc('lines', linewidth=2, color='r')
```

sets the current *rcParams* and is equivalent to:

```
rcParams['lines.linewidth'] = 2
rcParams['lines.color'] = 'r'
```

The following aliases are available to save typing for interactive users:

Alias	Property
'lw'	'linewidth'
'ls'	'linestyle'
'c'	'color'
'fc'	'facecolor'
'ec'	'edgecolor'
'mew'	'markeredgewidth'
'aa'	'antialiased'

Thus you could abbreviate the above call as:

```
rc('lines', lw=2, c='r')
```

Note you can use python's kwargs dictionary facility to store dictionaries of default parameters. e.g., you can customize the font rc as follows:

```
font = {'family' : 'monospace',
        'weight' : 'bold',
        'size'   : 'larger'}
rc('font', **font)  # pass in the font dict as kwargs
```

This enables you to easily switch between several configurations. Use `matplotlib.style.use('default')` or *rcdefaults()* to restore the default *rcParams* after changes.

Notes

Similar functionality is available by using the normal dict interface, i.e. `rcParams.update({"lines.linewidth": 2, ...})` (but `rcParams.update` does not support abbreviations or grouping).

Examples using `matplotlib.pyplot.rc`

- *Styling with cycler*

matplotlib.pyplot.rc_context

`matplotlib.pyplot.rc_context`(*rc=None*, *fname=None*)

Return a context manager for temporarily changing rcParams.

Parameters

rc

[dict] The rcParams to temporarily set.

fname

[str or path-like] A file with Matplotlib rc settings. If both *fname* and *rc* are given, settings from *rc* take precedence.

See also:

The matplotlibrc file

Examples

Passing explicit values via a dict:

```
with mpl.rc_context({'interactive': False}):
    fig, ax = plt.subplots()
    ax.plot(range(3), range(3))
    fig.savefig('example.png')
    plt.close(fig)
```

Loading settings from a file:

```
with mpl.rc_context(fname='print.rc'):
    plt.plot(x, y)  # uses 'print.rc'
```

Examples using `matplotlib.pyplot.rc_context`

- sphx_glr_gallery_text_labels_and_annotations_usetex_baseline_test.py
- sphx_glr_gallery_misc_logos2.py
- sphx_glr_gallery_ticks_and_spines_auto_ticks.py

matplotlib.pyplot.rcdefaults

matplotlib.pyplot.**rcdefaults**()

Restore the *rcParams* from Matplotlib's internal default style.

Style-blacklisted *rcParams* (defined in `matplotlib.style.core.STYLE_BLACKLIST`) are not updated.

See also:

matplotlib.rc_file_defaults

Restore the *rcParams* from the rc file originally loaded by Matplotlib.

matplotlib.style.use

Use a specific style file. Call `style.use('default')` to restore the default style.

Examples using `matplotlib.pyplot.rcdefaults`

- sphx_glr_gallery_lines_bars_and_markers_barh.py
- sphx_glr_gallery_misc_customize_rc.py

matplotlib.pyplot.rgrids

matplotlib.pyplot.**rgrids**(*radii=None*, *labels=None*, *angle=None*, *fmt=None*, ***kwargs*)

Get or set the radial gridlines on the current polar plot.

Call signatures:

```
lines, labels = rgrids()
lines, labels = rgrids(radii, labels=None, angle=22.5, fmt=None, **kwargs)
```

When called with no arguments, *rgrids* simply returns the tuple (*lines*, *labels*). When called with arguments, the labels will appear at the specified radial distances and angle.

Parameters

radii

[tuple with floats] The radii for the radial gridlines

labels

[tuple with strings or None] The labels to use at each radial gridline. The *matplotlib.ticker.ScalarFormatter* will be used if None.

angle

[float] The angular position of the radius labels in degrees.

fmt

[str or None] Format string used in *matplotlib.ticker.FormatStrFormatter*. For example '%f'.

Returns

lines

[list of *lines.Line2D*] The radial gridlines.

labels

[list of *text.Text*] The tick labels.

Other Parameters

****kwargs**

kwargs are optional *Text* properties for the labels.

See also:

pyplot.thetagrids

projections.polar.PolarAxes.set_rgrids

Axis.get_gridlines

Axis.get_ticklabels

Examples

```
# set the locations of the radial gridlines
lines, labels = rgrids( (0.25, 0.5, 1.0) )

# set the locations and labels of the radial gridlines
lines, labels = rgrids( (0.25, 0.5, 1.0), ('Tom', 'Dick', 'Harry' ))
```

Examples using `matplotlib.pyplot.rgrids`

matplotlib.pyplot.savefig

matplotlib.pyplot.**savefig**(**args*, ***kwargs*)

Save the current figure.

Call signature:

```
savefig(fname, dpi=None, facecolor='w', edgecolor='w',
        orientation='portrait', papertype=None, format=None,
        transparent=False, bbox_inches=None, pad_inches=0.1,
        frameon=None, metadata=None)
```

The available output formats depend on the backend being used.

Parameters

fname

[str or path-like or file-like] A path, or a Python file-like object, or possibly some backend-dependent object such as *matplotlib.backends.backend_pdf.PdfPages*.

If *format* is set, it determines the output format, and the file is saved as *fname*. Note that *fname* is used verbatim, and there is no attempt to make the extension, if any, of *fname* match *format*, and no extension is appended.

If *format* is not set, then the format is inferred from the extension of *fname*, if there is one. If *format* is not set and *fname* has no extension, then the file is saved with `rcParams["savefig.format"]` (default: `'png'`) and the appropriate extension is appended to *fname*.

Other Parameters

dpi

[float or 'figure', default: `rcParams["savefig.dpi"]` (default: `'figure'`)] The resolution in dots per inch. If 'figure', use the figure's dpi value.

quality

[int, default: `rcParams["savefig.jpeg_quality"]` (default: `95`)] Applicable only if *format* is 'jpg' or 'jpeg', ignored otherwise.

The image quality, on a scale from 1 (worst) to 95 (best). Values above 95 should be avoided; 100 disables portions of the JPEG compression algorithm, and results in large files with hardly any gain in image quality.

This parameter is deprecated.

optimize

[bool, default: False] Applicable only if *format* is 'jpg' or 'jpeg', ignored otherwise.

Whether the encoder should make an extra pass over the image in order to select optimal encoder settings.

This parameter is deprecated.

progressive

[bool, default: False] Applicable only if *format* is 'jpg' or 'jpeg', ignored otherwise.

Whether the image should be stored as a progressive JPEG file.

This parameter is deprecated.

facecolor

[color or 'auto', default: `rcParams["savefig.facecolor"]` (default: `'auto'`)] The facecolor of the figure. If 'auto', use the current figure facecolor.

edgecolor

[color or 'auto', default: `rcParams["savefig.edgecolor"]` (default: `'auto'`)] The edgecolor of the figure. If 'auto', use the current figure edgecolor.

orientation

[{'landscape', 'portrait'}] Currently only supported by the postscript backend.

papertype

[str] One of 'letter', 'legal', 'executive', 'ledger', 'a0' through 'a10', 'b0' through 'b10'. Only supported for postscript output.

format

[str] The file format, e.g. 'png', 'pdf', 'svg', ... The behavior when this is unset is documented under *fname*.

transparent

[bool] If *True*, the axes patches will all be transparent; the figure patch will also be transparent unless facecolor and/or edgecolor are specified via kwargs. This is useful, for example, for displaying a plot on top of a colored background on a web page. The transparency of these patches will be restored to their original values upon exit of this function.

bbox_inches

[str or *Bbox*, default: `rcParams["savefig.bbox"]` (default: `None`)] Bounding box in inches: only the given portion of the figure is saved. If 'tight', try to figure out the tight bbox of the figure.

pad_inches

[float, default: `rcParams["savefig.pad_inches"]` (default: `0.1`)] Amount of padding around the figure when bbox_inches is 'tight'.

bbox_extra_artists

[list of *Artist*, optional] A list of extra artists that will be considered when the tight bbox is calculated.

backend

[str, optional] Use a non-default backend to render the file, e.g. to render a png file with the "cairo" backend rather than the default "agg", or a pdf file with the "pgf" backend rather than the default "pdf". Note that the default backend is normally sufficient. See *The builtin backends* for a list of valid backends for each file format. Custom backends can be referenced as "module://...".

metadata

[dict, optional] Key/value pairs to store in the image metadata. The supported keys and defaults depend on the image format and backend:

- 'png' with Agg backend: See the parameter `metadata` of *print_png*.
- 'pdf' with pdf backend: See the parameter `metadata` of *PdfPages*.
- 'svg' with svg backend: See the parameter `metadata` of *print_svg*.
- 'eps' and 'ps' with PS backend: Only 'Creator' is supported.

pil_kwargs

[dict, optional] Additional keyword arguments that are passed to `PIL.Image.Image.save` when saving the figure.

Examples using `matplotlib.pyplot.savefig`

- sphx_glr_gallery_misc_print_stdout_sgskip.py
- sphx_glr_gallery_user_interfaces_svg_histogram_sgskip.py
- sphx_glr_gallery_user_interfaces_svg_tooltip_sgskip.py
- sphx_glr_gallery_userdemo_pgf_preamble_sgskip.py

matplotlib.pyplot.sca

`matplotlib.pyplot.`**sca**(*ax*)

Set the current Axes to *ax* and the current Figure to the parent of *ax*.

Examples using `matplotlib.pyplot.sca`

matplotlib.pyplot.scatter

`matplotlib.pyplot.`**scatter**(*x*, *y*, *s=None*, *c=None*, *marker=None*, *cmap=None*, *norm=None*, *vmin=None*, *vmax=None*, *alpha=None*, *linewidths=None*, *verts=<deprecated parameter>*, *edgecolors=None*, ***, *plotnonfinite=False*, *data=None*, ***kwargs*)

A scatter plot of *y* vs. *x* with varying marker size and/or color.

Parameters

x, y

[float or array-like, shape (n,)] The data positions.

s

[float or array-like, shape (n,), optional] The marker size in points**2. Default is `rcParams['lines.markersize'] ** 2`.

c

[array-like or list of colors or color, optional] The marker colors. Possible values:

- A scalar or sequence of n numbers to be mapped to colors using *cmap* and *norm*.
- A 2-D array in which the rows are RGB or RGBA.
- A sequence of colors of length n.
- A single color format string.

Note that *c* should not be a single numeric RGB or RGBA sequence because that is indistinguishable from an array of values to be colormapped. If you want to specify the same RGB or RGBA value for all points, use a 2-D array with a single row. Otherwise, value- matching will have precedence in case of a size matching with *x* and *y*.

If you wish to specify a single color for all points prefer the *color* keyword argument.

Defaults to `None`. In that case the marker color is determined by the value of *color*, *facecolor* or *facecolors*. In case those are not specified or `None`, the marker color is determined by the next color of the `Axes`' current "shape and fill" color cycle. This cycle defaults to `rcParams["axes.prop_cycle"]` (default: `cycler('color', ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728', '#9467bd', '#8c564b', '#e377c2', '#7f7f7f', '#bcbd22', '#17becf'])`).

marker

[*MarkerStyle*, default: `rcParams["scatter.marker"]` (default: `'o'`)] The marker style. *marker* can be either an instance of the class or the text shorthand for a particular marker. See *matplotlib.markers* for more information about marker styles.

cmap

[str or *Colormap*, default: `rcParams["image.cmap"]` (default: `'viridis'`)] A *Colormap* instance or registered colormap name. *cmap* is only used if *c* is an array of floats.

norm

[*Normalize*, default: None] If *c* is an array of floats, *norm* is used to scale the color data, *c*, in the range 0 to 1, in order to map into the colormap *cmap*. If *None*, use the default *colors.Normalize*.

vmin, vmax

[float, default: None] *vmin* and *vmax* are used in conjunction with the default norm to map the color array *c* to the colormap *cmap*. If None, the respective min and max of the color array is used. It is deprecated to use *vmin*/*vmax* when *norm* is given.

alpha

[float, default: None] The alpha blending value, between 0 (transparent) and 1 (opaque).

linewidths

[float or array-like, default: `rcParams["lines.linewidth"]` (default: `1.5`)] The linewidth of the marker edges. Note: The default *edgecolors* is 'face'. You may want to change this as well.

edgecolors

[{'face', 'none', *None*} or color or sequence of color, default: `rcParams["scatter.edgecolors"]` (default: `'face'`)] The edge color of the marker. Possible values:

- 'face': The edge color will always be the same as the face color.
- 'none': No patch boundary will be drawn.
- A color or sequence of colors.

For non-filled markers, the *edgecolors* kwarg is ignored and forced to 'face' internally.

plotnonfinite

[bool, default: False] Set to plot points with nonfinite *c*, in conjunction with *set_bad*.

Returns

PathCollection

Other Parameters

****kwargs**

[*Collection* properties]

See also:

plot

To plot scatter plots when markers are identical in size and color.

Notes

- The *plot* function will be faster for scatterplots where markers don't vary in size or color.
- Any or all of *x*, *y*, *s*, and *c* may be masked arrays, in which case all masks will be combined and only unmasked points will be plotted.
- Fundamentally, scatter works with 1-D arrays; *x*, *y*, *s*, and *c* may be input as N-D arrays, but within scatter they will be flattened. The exception is *c*, which will be flattened only if its size matches the size of *x* and *y*.

Note: In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, the following arguments can also be string `s`, which is interpreted as `data[s]` (unless this raises an exception): *x*, *y*, *s*, *linewidths*, *edgecolors*, *c*, *facecolor*, *facecolors*, *color*.

Objects passed as **data** must support item access (`data[s]`) and membership test (`s in data`).

Examples using `matplotlib.pyplot.scatter`

- sphx_glr_gallery_misc_hyperlinks_sgskip.py

matplotlib.pyplot.sci

`matplotlib.pyplot.sci`(*im*)

Set the current image.

This image will be the target of colormap functions like *viridis*, and other functions such as *clim*. The current image is an attribute of the current axes.

Examples using `matplotlib.pyplot.sci`

- sphx_glr_gallery_shapes_and_collections_line_collection.py

matplotlib.pyplot.semilogx

matplotlib.pyplot.**semilogx**(**args*, ***kwargs*)

Make a plot with log scaling on the x axis.

Call signatures:

```
semilogx([x], y, [fmt], data=None, **kwargs)
semilogx([x], y, [fmt], [x2], y2, [fmt2], ..., **kwargs)
```

This is just a thin wrapper around *plot* which additionally changes the x-axis to log scaling. All of the concepts and parameters of plot can be used here as well.

The additional parameters *base*, *subs*, and *nonpositive* control the x-axis properties. They are just forwarded to *Axes.set_xscale*.

Parameters

base

[float, default: 10] Base of the x logarithm.

subs

[array-like, optional] The location of the minor xticks. If *None*, reasonable locations are automatically chosen depending on the number of decades in the plot. See *Axes.set_xscale* for details.

nonposition

[{'mask', 'clip'}, default: 'mask'] Non-positive values in x can be masked as invalid, or clipped to a very small positive number.

Returns

lines

A list of *Line2D* objects representing the plotted data.

Other Parameters

****kwargs**

All parameters supported by *plot*.

Examples using `matplotlib.pyplot.semilogx`

- *Sample plots in Matplotlib*

matplotlib.pyplot.semilogy

matplotlib.pyplot.**semilogy**(**args*, ***kwargs*)

Make a plot with log scaling on the y axis.

Call signatures:

```
semilogy([x], y, [fmt], data=None, **kwargs)
semilogy([x], y, [fmt], [x2], y2, [fmt2], ..., **kwargs)
```

This is just a thin wrapper around *plot* which additionally changes the y-axis to log scaling. All of the concepts and parameters of plot can be used here as well.

The additional parameters *base*, *subs*, and *nonpositive* control the y-axis properties. They are just forwarded to *Axes.set_yscale*.

Parameters

base

[float, default: 10] Base of the y logarithm.

subs

[array-like, optional] The location of the minor yticks. If *None*, reasonable locations are automatically chosen depending on the number of decades in the plot. See *Axes.set_yscale* for details.

nonposition

[{'mask', 'clip'}, default: 'mask'] Non-positive values in y can be masked as invalid, or clipped to a very small positive number.

Returns

lines

A list of *Line2D* objects representing the plotted data.

Other Parameters

****kwargs**

All parameters supported by *plot*.

Examples using `matplotlib.pyplot.semilogy`

- *Sample plots in Matplotlib*

matplotlib.pyplot.set_cmap

matplotlib.pyplot.**set_cmap**(*cmap*)

Set the default colormap, and applies it to the current image if any.

Parameters

cmap

[*Colormap* or str] A colormap instance or the name of a registered colormap.

See also:

colormaps

matplotlib.cm.register_cmap

matplotlib.cm.get_cmap

Examples using `matplotlib.pyplot.set_cmap`

matplotlib.pyplot.setp

matplotlib.pyplot.**setp**(*obj*, **args*, ***kwargs*)

Set a property on an artist object.

matplotlib supports the use of *setp()* ("set property") and *getp()* to set and get object properties, as well as to do introspection on the object. For example, to set the linestyle of a line to be dashed, you can do:

```
>>> line, = plot([1, 2, 3])
>>> setp(line, linestyle='--')
```

If you want to know the valid types of arguments, you can provide the name of the property you want to set without a value:

```
>>> setp(line, 'linestyle')
    linestyle: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
```

If you want to see all the properties that can be set, and their possible values, you can do:

```
>>> setp(line)
    ... long output listing omitted
```

By default *setp* prints to sys.stdout, but this can be modified using the *file* keyword-only argument:

```
>>> with fopen('output.log') as f:
>>>     setp(line, file=f)
```

setp() operates on a single instance or a iterable of instances. If you are in query mode introspecting the possible values, only the first instance in the sequence is used. When actually setting values, all the instances will be set. e.g., suppose you have a list of two lines, the following will make both lines thicker and red:

```
>>> x = arange(0, 1, 0.01)
>>> y1 = sin(2*pi*x)
>>> y2 = sin(4*pi*x)
>>> lines = plot(x, y1, x, y2)
>>> setp(lines, linewidth=2, color='r')
```

setp() works with the MATLAB style string/value pairs or with python kwargs. For example, the following are equivalent:

```
>>> setp(lines, 'linewidth', 2, 'color', 'r')  # MATLAB style
>>> setp(lines, linewidth=2, color='r')        # python style
```

Examples using matplotlib.pyplot.setp

- sphx_glr_gallery_event_handling_ginput_manual_clabel_sgskip.py

matplotlib.pyplot.show

matplotlib.pyplot.**show**(*, *block=None*)

Display all open figures.

In non-interactive mode, *block* defaults to True. All figures will display and show will not return until all windows are closed. If there are no figures, return immediately.

In interactive mode *block* defaults to False. This will ensure that all of the figures are shown and this function immediately returns.

Parameters

block

[bool, optional] If True block and run the GUI main loop until all windows are closed.

If `False` ensure that all windows are displayed and return immediately. In this case, you are responsible for ensuring that the event loop is running to have responsive figures.

See also:

ion

enable interactive mode

ioff

disable interactive mode

Examples using `matplotlib.pyplot.show`

- sphx_glr_gallery_text_labels_and_annotations_font_family_rc_sgskip.py
- sphx_glr_gallery_event_handling_ginput_manual_clabel_sgskip.py
- sphx_glr_gallery_event_handling_pong_sgskip.py
- sphx_glr_gallery_misc_multiprocess_sgskip.py
- sphx_glr_gallery_user_interfaces_pylab_with_gtk_sgskip.py
- sphx_glr_gallery_user_interfaces_toolmanager_sgskip.py
- sphx_glr_gallery_widgets_lasso_selector_demo_sgskip.py

matplotlib.pyplot.specgram

matplotlib.pyplot.**specgram**(*x*, *NFFT=None*, *Fs=None*, *Fc=None*, *detrend=None*, *window=None*, *noverlap=None*, *cmap=None*, *xextent=None*, *pad_to=None*, *sides=None*, *scale_by_freq=None*, *mode=None*, *scale=None*, *vmin=None*, *vmax=None*, ***, *data=None*, ***kwargs*)

Plot a spectrogram.

Compute and plot a spectrogram of data in *x*. Data are split into *NFFT* length segments and the spectrum of each section is computed. The windowing function *window* is applied to each segment, and the amount of overlap of each segment is specified with *noverlap*. The spectrogram is plotted as a colormap (using imshow).

Parameters

x

[1-D array or sequence] Array or sequence containing the data.

Fs

[float, default: 2] The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, *freqs*, in cycles per time unit.

window

[callable or ndarray, default: *window_hanning*] A function or a vector of length *NFFT*. To create window vectors see *window_hanning*, *window_none*, numpy.blackman, numpy.hamming, numpy.bartlett, scipy.signal, scipy.signal.get_window, etc. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

sides

[{'default', 'onesided', 'twosided'}, optional] Which sides of the spectrum to return. 'default' is one-sided for real data and two-sided for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

pad_to

[int, optional] The number of points to which the data segment is padded when performing the FFT. This can be different from *NFFT*, which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to fft(). The default is None, which sets *pad_to* equal to *NFFT*

NFFT

[int, default: 256] The number of data points used in each block for the FFT. A power 2 is most efficient. This should *NOT* be used to get zero padding, or the scaling of the result will be incorrect; use *pad_to* for this instead.

detrend

[{'none', 'mean', 'linear'} or callable, default 'none'] The function applied to each segment before fft-ing, designed to remove the mean or linear trend. Unlike in MATLAB, where the *detrend* parameter is a vector, in Matplotlib is it a function. The *mlab* module defines *detrend_none*, *detrend_mean*, and *detrend_linear*, but you can use a custom function as well. You can also use a string to choose one of the functions: 'none' calls *detrend_none*. 'mean' calls *detrend_mean*. 'linear' calls *detrend_linear*.

scale_by_freq

[bool, default: True] Whether the resulting density values should be scaled by the scaling frequency, which gives density in units of Hz^-1. This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

mode

[{'default', 'psd', 'magnitude', 'angle', 'phase'}] What sort of spectrum to use. Default is 'psd', which takes the power spectral density. 'magnitude' returns the magnitude spectrum. 'angle' returns the phase spectrum without unwrapping. 'phase' returns the phase spectrum with unwrapping.

noverlap

[int] The number of points of overlap between blocks. The default value is 128.

scale

[{'default', 'linear', 'dB'}] The scaling of the values in the *spec*. 'linear' is no scaling. 'dB' returns the values in dB scale. When *mode* is 'psd', this is dB power (10 * log10). Otherwise this is dB amplitude (20 * log10). 'default' is 'dB' if *mode* is 'psd' or 'magnitude' and 'linear' otherwise. This must be 'linear' if *mode* is 'angle' or 'phase'.

Fc

[int, default: 0] The center frequency of *x*, which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

cmap

[*Colormap*, default: `rcParams["image.cmap"]` (default: `'viridis'`)]

xextent

[*None* or (xmin, xmax)] The image extent along the x-axis. The default sets *xmin* to the left border of the first bin (*spectrum* column) and *xmax* to the right border of the last bin. Note that for *noverlap>0* the width of the bins is smaller than those of the segments.

****kwargs**

Additional keyword arguments are passed on to *imshow* which makes the specgram image.

Returns

spectrum

[2-D array] Columns are the periodograms of successive segments.

freqs

[1-D array] The frequencies corresponding to the rows in *spectrum*.

t

[1-D array] The times corresponding to midpoints of segments (i.e., the columns in *spectrum*).

im

[*AxesImage*] The image created by imshow containing the spectrogram.

See also:

psd

Differs in the default overlap; in returning the mean of the segment periodograms; in not returning times; and in generating a line plot instead of colormap.

magnitude_spectrum

A single spectrum, similar to having a single segment when *mode* is 'magnitude'. Plots a line instead of a colormap.

angle_spectrum

A single spectrum, similar to having a single segment when *mode* is 'angle'. Plots a line instead of a colormap.

phase_spectrum

A single spectrum, similar to having a single segment when *mode* is 'phase'. Plots a line instead of a colormap.

Notes

The parameters *detrend* and *scale_by_freq* do only apply when *mode* is set to 'psd'.

Note: In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, the following arguments can also be string `s`, which is interpreted as `data[s]` (unless this raises an exception): *x*.

Objects passed as **data** must support item access (`data[s]`) and membership test (`s in data`).

Examples using `matplotlib.pyplot.specgram`

- sphx_glr_gallery_images_contours_and_fields_specgram_demo.py
- *Sample plots in Matplotlib*

matplotlib.pyplot.spring

matplotlib.pyplot.**spring**()

Set the colormap to "spring".

This changes the default colormap as well as the colormap of the current image if there is one. See `help(colormaps)` for more information.

Examples using `matplotlib.pyplot.spring`

matplotlib.pyplot.spy

matplotlib.pyplot.**spy**(*Z*, *precision=0*, *marker=None*, *markersize=None*, *aspect='equal'*, *origin='upper'*, ***kwargs*)

Plot the sparsity pattern of a 2D array.

This visualizes the non-zero values of the array.

Two plotting styles are available: image and marker. Both are available for full arrays, but only the marker style works for `scipy.sparse.spmatrix` instances.

Image style

If *marker* and *markersize* are *None*, *imshow* is used. Any extra remaining keyword arguments are passed to this method.

Marker style

If *Z* is a `scipy.sparse.spmatrix` or *marker* or *markersize* are *None*, a *Line2D* object will be returned with the value of marker determining the marker type, and any remaining keyword arguments passed to *plot*.

Parameters

Z

[array-like (M, N)] The array to be plotted.

precision

[float or 'present', default: 0] If *precision* is 0, any non-zero value will be plotted. Otherwise, values of $|Z| > precision$ will be plotted.

For `scipy.sparse.spmatrix` instances, you can also pass 'present'. In this case any value present in the array will be plotted, even if it is identically zero.

aspect

[{'equal', 'auto', None} or float, default: 'equal'] The aspect ratio of the axes. This parameter is particularly relevant for images since it determines whether data pixels are square.

This parameter is a shortcut for explicitly calling *Axes.set_aspect*. See there for further details.

- 'equal': Ensures an aspect ratio of 1. Pixels will be square.
- 'auto': The axes is kept fixed and the aspect is adjusted so that the data fit in the axes. In general, this will result in non-square pixels.
- *None*: Use `rcParams["image.aspect"]` (default: `'equal'`).

origin

[{'upper', 'lower'}, default: `rcParams["image.origin"]` (default: `'upper'`)] Place the [0, 0] index of the array in the upper left or lower left corner of the axes. The convention 'upper' is typically used for matrices and images.

Returns

AxesImage **or** *Line2D*

The return type depends on the plotting style (see above).

Other Parameters

****kwargs**

The supported additional parameters depend on the plotting style.

For the image style, you can pass the following additional parameters of *imshow*:

- *cmap*
- *alpha*
- *url*

- any *Artist* properties (passed on to the *AxesImage*)

For the marker style, you can pass any *Line2D* property except for *linestyle*:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value
alpha	float or None
animated	bool
antialiased or aa	bool
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color or c	color
contains	unknown
dash_capstyle	{'butt', 'round', 'projecting'}
dash_joinstyle	{'miter', 'round', 'bevel'}
dashes	sequence of floats (on/off ink in points) or (None, None)
data	(2, N) array or two 1D arrays
drawstyle or ds	{'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'd
figure	*Figure*
fillstyle	{'full', 'left', 'right', 'bottom', 'top', 'none'}
gid	str
in_layout	bool
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float
marker	marker style string, *Path* or *MarkerStyle*
markeredgecolor or mec	color
markeredgewidth or mew	float
markerfacecolor or mfc	color
markerfacecoloralt or mfcalt	color
markersize or ms	float
markevery	None or int or (int, int) or slice or List[int] or float or (float, float)
path_effects	*AbstractPathEffect*
picker	unknown
pickradius	float
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
solid_capstyle	{'butt', 'round', 'projecting'}
solid_joinstyle	{'miter', 'round', 'bevel'}
transform	*matplotlib.transforms.Transform*
url	str
visible	bool
xdata	1D array

Table 208 – continued from previous page

Property	Description
ydata	1D array
zorder	float

Examples using `matplotlib.pyplot.spy`

- sphx_glr_gallery_images_contours_and_fields_spy_demos.py

matplotlib.pyplot.stackplot

matplotlib.pyplot.**stackplot**(*x*, **args*, *labels=()*, *colors=None*, *baseline='zero'*, *data=None*, ***kwargs*)

Draw a stacked area plot.

Parameters

x

[1d array of dimension N]

y

[2d array (dimension MxN), or sequence of 1d arrays (each dimension 1xN)] The data is assumed to be unstacked. Each of the following calls is legal:

```
stackplot(x, y)               # where y is MxN
stackplot(x, y1, y2, y3, y4)  # where y1, y2, y3, y4, are all 1xNm
```

baseline

[{'zero', 'sym', 'wiggle', 'weighted_wiggle'}] Method used to calculate the baseline:

- `'zero'`: Constant zero baseline, i.e. a simple stacked plot.
- `'sym'`: Symmetric around zero and is sometimes called 'ThemeRiver'.
- `'wiggle'`: Minimizes the sum of the squared slopes.
- `'weighted_wiggle'`: Does the same but weights to account for size of each layer. It is also called 'Streamgraph'-layout. More details can be found at http://leebyron.com/streamgraph/.

labels

[Length N sequence of strings] Labels to assign to each data series.

colors

[Length N sequence of colors] A list or tuple of colors. These will be cycled through and used to colour the stacked areas.

****kwargs**

All other keyword arguments are passed to *Axes.fill_between*.

Returns

list of *PolyCollection*

A list of *PolyCollection* instances, one for each element in the stacked area plot.

Notes

Note: In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, every other argument can also be string `s`, which is interpreted as `data[s]` (unless this raises an exception).

Objects passed as **data** must support item access (`data[s]`) and membership test (`s in data`).

Examples using `matplotlib.pyplot.stackplot`

matplotlib.pyplot.stem

`matplotlib.pyplot.stem`(**args*, *linefmt=None*, *markerfmt=None*, *basefmt=None*, *bottom=0*, *label=None*, *use_line_collection=True*, *data=None*)

Create a stem plot.

A stem plot plots vertical lines at each *x* location from the baseline to *y*, and places a marker there.

Call signature:

```
stem([x,] y, linefmt=None, markerfmt=None, basefmt=None)
```

The x-positions are optional. The formats may be provided either as positional or as keyword-arguments.

Parameters

x

[array-like, optional] The x-positions of the stems. Default: (0, 1, ..., len(y) - 1).

y

[array-like] The y-values of the stem heads.

linefmt

[str, optional] A string defining the properties of the vertical lines. Usually, this will be a color or a color and a linestyle:

Character	Line Style
`'-'`	solid line
`'--'`	dashed line
`'-.'`	dash-dot line
`':'`	dotted line

Default: 'C0-', i.e. solid line with the first color of the color cycle.

Note: While it is technically possible to specify valid formats other than color or color and linestyle (e.g. 'rx' or '-.'), this is beyond the intention of the method and will most likely not result in a reasonable plot.

markerfmt

[str, optional] A string defining the properties of the markers at the stem heads. Default: 'C0o', i.e. filled circles with the first color of the color cycle.

basefmt

[str, default: 'C3-' ('C2-' in classic mode)] A format string defining the properties of the baseline.

bottom

[float, default: 0] The y-position of the baseline.

label

[str, default: None] The label to use for the stems in legends.

use_line_collection

[bool, default: True] If `True`, store and plot the stem lines as a *LineCollection* instead of individual lines, which significantly increases performance. If `False`, defaults to the old behavior of using a list of *Line2D* objects. This parameter may be deprecated in the future.

Returns

StemContainer

The container may be treated like a tuple (*markerline*, *stemlines*, *baseline*)

Notes

See also:

The MATLAB function stem which inspired this method.

Note: In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, every other argument can also be string `s`, which is interpreted as `data[s]` (unless this raises an exception).

Objects passed as **data** must support item access (`data[s]`) and membership test (`s in data`).

Examples using `matplotlib.pyplot.stem`

- sphx_glr_gallery_lines_bars_and_markers_stem_plot.py

matplotlib.pyplot.step

matplotlib.pyplot.**step**(*x*, *y*, **args*, *where='pre'*, *data=None*, ***kwargs*)

Make a step plot.

Call signatures:

```
step(x, y, [fmt], *, data=None, where='pre', **kwargs)
step(x, y, [fmt], x2, y2, [fmt2], ..., *, where='pre', **kwargs)
```

This is just a thin wrapper around *plot* which changes some formatting options. Most of the concepts and parameters of plot can be used here as well.

Parameters

x

[array-like] 1-D sequence of x positions. It is assumed, but not checked, that it is uniformly increasing.

y

[array-like] 1-D sequence of y levels.

fmt

[str, optional] A format string, e.g. 'g' for a green line. See *plot* for a more detailed description.

Note: While full format strings are accepted, it is recommended to only specify the color. Line styles are currently ignored (use the keyword argument *linestyle* instead). Markers are accepted and plotted on the given positions, however, this is a rarely needed feature for step plots.

data

[indexable object, optional] An object with labelled data. If given, provide the label names to plot in *x* and *y*.

where

[{'pre', 'post', 'mid'}, default: 'pre'] Define where the steps should be placed:

- 'pre': The y value is continued constantly to the left from every *x* position, i.e. the interval `(x[i-1], x[i]]` has the value `y[i]`.
- 'post': The y value is continued constantly to the right from every *x* position, i.e. the interval `[x[i], x[i+1])` has the value `y[i]`.
- 'mid': Steps occur half-way between the *x* positions.

Returns

lines

A list of *Line2D* objects representing the plotted data.

Other Parameters

****kwargs**

Additional parameters are the same as those for *plot*.

Notes

Examples using `matplotlib.pyplot.step`

- sphx_glr_gallery_lines_bars_and_markers_step_demo.py

matplotlib.pyplot.streamplot

matplotlib.pyplot.**streamplot**(*x*, *y*, *u*, *v*, *density=1*, *linewidth=None*, *color=None*, *cmap=None*, *norm=None*, *arrowsize=1*, *arrowstyle='-|>'*, *minlength=0.1*, *transform=None*, *zorder=None*, *start_points=None*, *maxlength=4.0*, *integration_direction='both'*, ***, *data=None*)

Draw streamlines of a vector flow.

Parameters

x, y

[1D arrays] An evenly spaced grid.

u, v

[2D arrays] *x* and *y*-velocities. The number of rows and columns must match the length of *y* and *x*, respectively.

density

[float or (float, float)] Controls the closeness of streamlines. When `density = 1`, the domain is divided into a 30x30 grid. *density* linearly scales this grid. Each cell in the grid can have, at most, one traversing streamline. For different densities in each direction, use a tuple (density_x, density_y).

linewidth

[float or 2D array] The width of the stream lines. With a 2D array the line width can be varied across the grid. The array must have the same shape as *u* and *v*.

color

[color or 2D array] The streamline color. If given an array, its values are converted to colors using *cmap* and *norm*. The array must have the same shape as *u* and *v*.

cmap

[*Colormap*] Colormap used to plot streamlines and arrows. This is only used if *color* is an array.

norm

[*Normalize*] Normalize object used to scale luminance data to 0, 1. If `None`, stretch (min, max) to (0, 1). This is only used if *color* is an array.

arrowsize

[float] Scaling factor for the arrow size.

arrowstyle

[str] Arrow style specification. See *FancyArrowPatch*.

minlength

[float] Minimum length of streamline in axes coordinates.

start_points

[Nx2 array] Coordinates of starting points for the streamlines in data coordinates (the same coordinates as the *x* and *y* arrays).

zorder

[int] The zorder of the stream lines and arrows. Artists with lower zorder values are drawn first.

maxlength

[float] Maximum length of streamline in axes coordinates.

integration_direction

[{'forward', 'backward', 'both'}, default: 'both'] Integrate the streamline in forward, backward or both directions.

Returns

StreamplotSet

Container object with attributes

- `lines`: *LineCollection* of streamlines
- `arrows`: *PatchCollection* containing *FancyArrowPatch* objects representing the arrows half-way along stream lines.

This container will probably change in the future to allow changes to the colormap, alpha, etc. for both lines and arrows, but these changes should be backward compatible.

Notes

Note: In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, the following arguments can also be string `s`, which is interpreted as `data[s]` (unless this raises an exception): *x*, *y*, *u*, *v*, *start_points*.

Objects passed as **data** must support item access (`data[s]`) and membership test (`s in data`).

Examples using `matplotlib.pyplot.streamplot`

- sphx_glr_gallery_images_contours_and_fields_plot_streamplot.py
- *Sample plots in Matplotlib*

matplotlib.pyplot.subplot

matplotlib.pyplot.**subplot**(**args*, ***kwargs*)

Add a subplot to the current figure.

Wrapper of *Figure.add_subplot* with a difference in behavior explained in the notes section.

Call signatures:

```
subplot(nrows, ncols, index, **kwargs)
subplot(pos, **kwargs)
subplot(**kwargs)
subplot(ax)
```

Parameters

***args**

[int, (int, int, *index*), or *SubplotSpec*, default: (1, 1, 1)] The position of the subplot described by one of

- Three integers (*nrows*, *ncols*, *index*). The subplot will take the *index* position on a grid with *nrows* rows and *ncols* columns. *index* starts at 1 in the upper left corner and increases to the right. *index* can also be a two-tuple specifying the (*first*, *last*) indices (1-based, and including *last*) of the subplot, e.g., `fig.add_subplot(3, 1, (1, 2))` makes a subplot that spans the upper 2/3 of the figure.
- A 3-digit integer. The digits are interpreted as if given separately as three single-digit integers, i.e. `fig.add_subplot(235)` is the same as `fig.add_subplot(2, 3, 5)`. Note that this can only be used if there are no more than 9 subplots.
- A *SubplotSpec*.

projection

[{None, 'aitoff', 'hammer', 'lambert', 'mollweide', 'polar', 'rectilinear', str}, optional] The projection type of the subplot (*Axes*). *str* is the name of a custom projection, see *projections*. The default None results in a 'rectilinear' projection.

polar

[bool, default: False] If True, equivalent to projection='polar'.

sharex, sharey

[*Axes*, optional] Share the x or y *axis* with sharex and/or sharey. The axis will have the same limits, ticks, and scale as the axis of the shared axes.

label

[str] A label for the returned axes.

Returns

axes.SubplotBase, **or another subclass of** *Axes*

The axes of the subplot. The returned axes base class depends on the projection used. It is *Axes* if rectilinear projection is used and *projections.polar.PolarAxes* if polar projection is used. The returned axes is then a subplot subclass of the base class.

Other Parameters

****kwargs**

This method also takes the keyword arguments for the returned axes base class; except for the *figure* argument. The keyword arguments for the rectilinear base class *Axes* can be found in the following table but there might also be other keyword arguments if another projection is used.

Property	Description
adjustable	{'box', 'datalim'}
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and re
alpha	float or None
anchor	2-tuple of floats or {'C', 'SW', 'S', 'SE', ...}
animated	bool
aspect	{'auto'} or num
autoscale_on	bool
autoscalex_on	bool
autoscaley_on	bool
axes_locator	Callable[[Axes, Renderer], Bbox]
axisbelow	bool or 'line'
box_aspect	None, or a number
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None

Table 209 – continued from previous page

Property	Description
contains	unknown
facecolor or fc	color
figure	*Figure*
frame_on	bool
gid	str
in_layout	bool
label	object
navigate	bool
navigate_mode	unknown
path_effects	*AbstractPathEffect*
picker	None or bool or callable
position	[left, bottom, width, height] or *Bbox*
prop_cycle	unknown
rasterization_zorder	float or None
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
title	str
transform	*Transform*
url	str
visible	bool
xbound	unknown
xlabel	str
xlim	(bottom: float, top: float)
xmargin	float greater than -0.5
xscale	{"linear", "log", "symlog", "logit", ...}
xticklabels	unknown
xticks	unknown
ybound	unknown
ylabel	str
ylim	(bottom: float, top: float)
ymargin	float greater than -0.5
yscale	{"linear", "log", "symlog", "logit", ...}
yticklabels	unknown
yticks	unknown
zorder	float

See also:

Figure.add_subplot

pyplot.subplots

pyplot.axes

Figure.subplots

Notes

Creating a subplot will delete any pre-existing subplot that overlaps with it beyond sharing a boundary:

```
import matplotlib.pyplot as plt
# plot a line, implicitly creating a subplot(111)
plt.plot([1, 2, 3])
# now create a subplot which represents the top plot of a grid
# with 2 rows and 1 column. Since this subplot will overlap the
# first, the plot (and its axes) previously created, will be removed
plt.subplot(211)
```

If you do not want this behavior, use the *Figure.add_subplot* method or the *pyplot.axes* function instead.

If the figure already has a subplot with key (*args*, *kwargs*) then it will simply make that subplot current and return it. This behavior is deprecated. Meanwhile, if you do not want this behavior (i.e., you want to force the creation of a new subplot), you must use a unique set of args and kwargs. The axes *label* attribute has been exposed for this purpose: if you want two subplots that are otherwise identical to be added to the figure, make sure you give them unique labels.

In rare circumstances, *add_subplot* may be called with a single argument, a subplot axes instance already created in the present figure but not in the figure's list of axes.

Examples

```
plt.subplot(221)

# equivalent but more general
ax1=plt.subplot(2, 2, 1)

# add a subplot with no frame
ax2=plt.subplot(222, frameon=False)

# add a polar subplot
plt.subplot(223, projection='polar')

# add a red subplot that shares the x-axis with ax1
plt.subplot(224, sharex=ax1, facecolor='red')

# delete ax2 from the figure
plt.delaxes(ax2)

# add ax2 to the figure again
plt.subplot(ax2)
```

Examples using `matplotlib.pyplot.subplot`

- sphx_glr_gallery_lines_bars_and_markers_psd_demo.py
- sphx_glr_gallery_subplots_axes_and_figures_axes_margins.py
- sphx_glr_gallery_subplots_axes_and_figures_axes_zoom_effect.py
- sphx_glr_gallery_subplots_axes_and_figures_demo_tight_layout.py
- sphx_glr_gallery_subplots_axes_and_figures_geo_demo.py
- sphx_glr_gallery_subplots_axes_and_figures_multiple_figs_demo.py
- sphx_glr_gallery_subplots_axes_and_figures_shared_axis_demo.py
- sphx_glr_gallery_subplots_axes_and_figures_subplot.py
- sphx_glr_gallery_subplots_axes_and_figures_subplots_adjust.py
- sphx_glr_gallery_pie_and_polar_charts_polar_bar.py
- sphx_glr_gallery_text_labels_and_annotations_demo_text_path.py
- sphx_glr_gallery_text_labels_and_annotations_multiline.py
- sphx_glr_gallery_pyplots_pyplot_two_subplots.py
- sphx_glr_gallery_pyplots_whats_new_98_4_legend.py
- sphx_glr_gallery_shapes_and_collections_ellipse_demo.py
- sphx_glr_gallery_axes_grid1_simple_colorbar.py
- *MATPLOTLIB UNCHAINED*
- sphx_glr_gallery_event_handling_trifinder_event_demo.py
- sphx_glr_gallery_misc_custom_projection.py
- sphx_glr_gallery_misc_customize_rc.py
- sphx_glr_gallery_misc_patheffect_demo.py
- sphx_glr_gallery_misc_transoffset.py
- sphx_glr_gallery_recipes_share_axis_lims_views.py
- sphx_glr_gallery_scales_symlog_demo.py
- sphx_glr_gallery_userdemo_simple_legend01.py
- *Pyplot tutorial*
- *Sample plots in Matplotlib*
- *Legend guide*
- *Customizing Figure Layouts Using GridSpec and Other Functions*
- *Constrained Layout Guide*

- *Tight Layout guide*

matplotlib.pyplot.subplot2grid

matplotlib.pyplot.**subplot2grid**(*shape*, *loc*, *rowspan=1*, *colspan=1*, *fig=None*, ***kwargs*)

Create a subplot at a specific location inside a regular grid.

Parameters

shape

[(int, int)] Number of rows and of columns of the grid in which to place axis.

loc

[(int, int)] Row number and column number of the axis location within the grid.

rowspan

[int, default: 1] Number of rows for the axis to span to the right.

colspan

[int, default: 1] Number of columns for the axis to span downwards.

fig

[*Figure*, optional] Figure to place the subplot in. Defaults to the current figure.

****kwargs**

Additional keyword arguments are handed to *add_subplot*.

Returns

axes.SubplotBase, **or another subclass of** *Axes*

The axes of the subplot. The returned axes base class depends on the projection used. It is *Axes* if rectilinear projection is used and *projections.polar.PolarAxes* if polar projection is used. The returned axes is then a subplot subclass of the base class.

Notes

The following call

```
ax = subplot2grid((nrows, ncols), (row, col), rowspan, colspan)
```

is identical to

```
fig = gcf()
gs = fig.add_gridspec(nrows, ncols)
ax = fig.add_subplot(gs[row:row+rowspan, col:col+colspan])
```

Examples using `matplotlib.pyplot.subplot2grid`

- sphx_glr_gallery_subplots_axes_and_figures_demo_tight_layout.py
- sphx_glr_gallery_userdemo_demo_gridspec01.py
- *Customizing Figure Layouts Using GridSpec and Other Functions*
- *Constrained Layout Guide*
- *Tight Layout guide*

matplotlib.pyplot.subplot_mosaic

matplotlib.pyplot.**subplot_mosaic**(*layout*, ***, *subplot_kw=None*, *gridspec_kw=None*, *empty_sentinel='.'*, ***fig_kw*)

Build a layout of Axes based on ASCII art or nested lists.

This is a helper function to build complex GridSpec layouts visually.

Note: This API is provisional and may be revised in the future based on early user feedback.

Parameters

layout

[list of list of {hashable or nested} or str] A visual layout of how you want your Axes to be arranged labeled as strings. For example

```
x = [['A panel', 'A panel', 'edge'],
     ['C panel', '.',       'edge']]
```

Produces 4 axes:

- 'A panel' which is 1 row high and spans the first two columns
- 'edge' which is 2 rows high and is on the right edge
- 'C panel' which in 1 row and 1 column wide in the bottom left
- a blank space 1 row and 1 column wide in the bottom center

Any of the entries in the layout can be a list of lists of the same form to create nested layouts.

If input is a str, then it must be of the form

```
'''
AAE
C.E
'''
```

where each character is a column and each line is a row. This only allows only single character Axes labels and does not allow nesting but is very terse.

subplot_kw

[dict, optional] Dictionary with keywords passed to the *Figure.add_subplot* call used to create each subplot.

gridspec_kw

[dict, optional] Dictionary with keywords passed to the *GridSpec* constructor used to create the grid the subplots are placed on.

empty_sentinel

[object, optional] Entry in the layout to mean "leave this space empty". Defaults to `'.'`. Note, if *layout* is a string, it is processed via `inspect.cleandoc` to remove leading white space, which may interfere with using white-space as the empty sentinel.

****fig_kw**

All additional keyword arguments are passed to the *pyplot.figure* call.

Returns

fig

[*Figure*] The new figure

dict[label, Axes]

A dictionary mapping the labels to the Axes objects.

Examples using `matplotlib.pyplot.subplot_mosaic`

matplotlib.pyplot.subplot_tool

`matplotlib.pyplot.subplot_tool`(*targetfig=None*)

Launch a subplot tool window for a figure.

A *matplotlib.widgets.SubplotTool* instance is returned.

Examples using `matplotlib.pyplot.subplot_tool`

- sphx_glr_gallery_subplots_axes_and_figures_subplot_toolbar.py

matplotlib.pyplot.subplots

`matplotlib.pyplot.subplots`(*nrows=1*, *ncols=1*, ***, *sharex=False*, *sharey=False*, *squeeze=True*, *subplot_kw=None*, *gridspec_kw=None*, ***fig_kw*)

Create a figure and a set of subplots.

This utility wrapper makes it convenient to create common layouts of subplots, including the enclosing figure object, in a single call.

Parameters

nrows, ncols

[int, default: 1] Number of rows/columns of the subplot grid.

sharex, sharey

[bool or {'none', 'all', 'row', 'col'}, default: False] Controls sharing of properties among x (*sharex*) or y (*sharey*) axes:

- True or 'all': x- or y-axis will be shared among all subplots.
- False or 'none': each subplot x- or y-axis will be independent.
- 'row': each subplot row will share an x- or y-axis.
- 'col': each subplot column will share an x- or y-axis.

When subplots have a shared x-axis along a column, only the x tick labels of the bottom subplot are created. Similarly, when subplots have a shared y-axis along a row, only the y tick labels of the first column subplot are created. To later turn other subplots' ticklabels on, use *tick_params*.

squeeze

[bool, default: True]

- If True, extra dimensions are squeezed out from the returned array of *Axes*:
  - if only one subplot is constructed (nrows=ncols=1), the resulting single Axes object is returned as a scalar.
  - for Nx1 or 1xM subplots, the returned object is a 1D numpy object array of Axes objects.
  - for NxM, subplots with N>1 and M>1 are returned as a 2D array.
- If False, no squeezing at all is done: the returned Axes object is always a 2D array containing Axes instances, even if it ends up being 1x1.

subplot_kw

[dict, optional] Dict with keywords passed to the *add_subplot* call used to create each subplot.

gridspec_kw

[dict, optional] Dict with keywords passed to the *GridSpec* constructor used to create the grid the subplots are placed on.

****fig_kw**

All additional keyword arguments are passed to the *pyplot.figure* call.

Returns

fig

[*Figure*]

ax

[*axes.Axes* or array of Axes] *ax* can be either a single *Axes* object or an array of Axes objects if more than one subplot was created. The dimensions of the resulting array can be controlled with the squeeze keyword, see above.

Typical idioms for handling the return value are:

```
# using the variable ax for single a Axes
fig, ax = plt.subplots()

# using the variable axs for multiple Axes
fig, axs = plt.subplots(2, 2)

# using tuple unpacking for multiple Axes
fig, (ax1, ax2) = plt.subplot(1, 2)
fig, ((ax1, ax2), (ax3, ax4)) = plt.subplot(2, 2)
```

The names `ax` and pluralized `axs` are preferred over `axes` because for the latter it's not clear if it refers to a single *Axes* instance or a collection of these.

See also:

pyplot.figure

pyplot.subplot

pyplot.axes

Figure.subplots

Figure.add_subplot

Examples

```
# First create some toy data:
x = np.linspace(0, 2*np.pi, 400)
y = np.sin(x**2)

# Create just a figure and only one subplot
fig, ax = plt.subplots()
ax.plot(x, y)
ax.set_title('Simple plot')

# Create two subplots and unpack the output array immediately
f, (ax1, ax2) = plt.subplots(1, 2, sharey=True)
ax1.plot(x, y)
ax1.set_title('Sharing Y axis')
ax2.scatter(x, y)

# Create four polar axes and access them through the returned array
fig, axs = plt.subplots(2, 2, subplot_kw=dict(polar=True))
axs[0, 0].plot(x, y)
axs[1, 1].scatter(x, y)

# Share a X axis with each column of subplots
plt.subplots(2, 2, sharex='col')

# Share a Y axis with each row of subplots
plt.subplots(2, 2, sharey='row')

# Share both X and Y axes with all subplots
plt.subplots(2, 2, sharex='all', sharey='all')

# Note that this is the same as
plt.subplots(2, 2, sharex=True, sharey=True)

# Create figure number 10 with a single subplot
```

(continues on next page)

(continued from previous page)

```
# and clears it if it already exists.
fig, ax = plt.subplots(num=10, clear=True)
```

Examples using `matplotlib.pyplot.subplots`

- sphx_glr_gallery_text_labels_and_annotations_font_family_rc_sgskip.py
- sphx_glr_gallery_event_handling_pong_sgskip.py
- sphx_glr_gallery_misc_multiprocess_sgskip.py
- sphx_glr_gallery_user_interfaces_pylab_with_gtk_sgskip.py
- sphx_glr_gallery_user_interfaces_svg_tooltip_sgskip.py
- sphx_glr_gallery_widgets_lasso_selector_demo_sgskip.py

matplotlib.pyplot.subplots_adjust

matplotlib.pyplot.**subplots_adjust**(*left=None*, *bottom=None*, *right=None*, *top=None*, *wspace=None*, *hspace=None*)

Adjust the subplot layout parameters.

Unset parameters are left unmodified; initial values are given by `rcParams["figure.subplot.[name]"]`.

Parameters

left

[float, optional] The position of the left edge of the subplots, as a fraction of the figure width.

right

[float, optional] The position of the right edge of the subplots, as a fraction of the figure width.

bottom

[float, optional] The position of the bottom edge of the subplots, as a fraction of the figure height.

top

[float, optional] The position of the top edge of the subplots, as a fraction of the figure height.

wspace

[float, optional] The width of the padding between subplots, as a fraction of the average axes width.

hspace

[float, optional] The height of the padding between subplots, as a fraction of the average axes height.

Examples using `matplotlib.pyplot.subplots_adjust`

- sphx_glr_gallery_images_contours_and_fields_irregulardatagrid.py
- sphx_glr_gallery_subplots_axes_and_figures_subplots_adjust.py
- sphx_glr_gallery_statistics_customized_violin.py
- sphx_glr_gallery_text_labels_and_annotations_multiline.py
- sphx_glr_gallery_text_labels_and_annotations_text_fontdict.py
- sphx_glr_gallery_axisartist_demo_parasite_axes2.py
- sphx_glr_gallery_misc_table_demo.py
- sphx_glr_gallery_ticks_and_spines_ticklabels_rotation.py
- sphx_glr_gallery_widgets_buttons.py
- sphx_glr_gallery_widgets_check_buttons.py
- sphx_glr_gallery_widgets_radio_buttons.py
- sphx_glr_gallery_widgets_slider_demo.py
- *Pyplot tutorial*
- *Customizing Figure Layouts Using GridSpec and Other Functions*

matplotlib.pyplot.summer

matplotlib.pyplot.**summer**()

Set the colormap to "summer".

This changes the default colormap as well as the colormap of the current image if there is one. See `help(colormaps)` for more information.

Examples using `matplotlib.pyplot.summer`

matplotlib.pyplot.suptitle

matplotlib.pyplot.**suptitle**(*t*, ***kwargs*)

Add a centered title to the figure.

Parameters

t

[str] The title text.

x

[float, default 0.5] The x location of the text in figure coordinates.

y

[float, default 0.98] The y location of the text in figure coordinates.

horizontalalignment, ha

[{'center', 'left', right'}, default: 'center'] The horizontal alignment of the text relative to (*x*, *y*).

verticalalignment, va

[{'top', 'center', 'bottom', 'baseline'}, default: 'top'] The vertical alignment of the text relative to (*x*, *y*).

fontsize, size

[default: `rcParams["figure.titlesize"]` (default: `'large'`)] The font size of the text. See *Text.set_size* for possible values.

fontweight, weight

[default: `rcParams["figure.titleweight"]` (default: `'normal'`)] The font weight of the text. See *Text.set_weight* for possible values.

Returns

text

The *Text* instance of the title.

Other Parameters

fontproperties

[None or dict, optional] A dict of font properties. If *fontproperties* is given the default values for font size and weight are taken from the *FontProperties* defaults. `rcParams["figure.titlesize"]` (default: `'large'`) and `rcParams["figure.titleweight"]` (default: `'normal'`) are ignored in this case.

****kwargs**

Additional kwargs are *matplotlib.text.Text* properties.

Examples

```
>>> fig.suptitle('This is the figure title', fontsize=12)
```

Examples using `matplotlib.pyplot.suptitle`

- sphx_glr_gallery_subplots_axes_and_figures_gridspec_nested.py
- *Pyplot tutorial*

matplotlib.pyplot.switch_backend

matplotlib.pyplot.**switch_backend**(*newbackend*)

Close all open figures and set the Matplotlib backend.

The argument is case-insensitive. Switching to an interactive backend is possible only if no event loop for another interactive backend has started. Switching to and from non-interactive backends is always possible.

Parameters

newbackend

[str] The name of the backend to use.

Examples using `matplotlib.pyplot.switch_backend`

matplotlib.pyplot.table

matplotlib.pyplot.**table**(*cellText=None*, *cellColours=None*, *cellLoc='right'*, *colWidths=None*, *rowLabels=None*, *rowColours=None*, *rowLoc='left'*, *colLabels=None*, *colColours=None*, *colLoc='center'*, *loc='bottom'*, *bbox=None*, *edges='closed'*, ***kwargs*)

Add a table to an *Axes*.

At least one of *cellText* or *cellColours* must be specified. These parameters must be 2D lists, in which the outer lists define the rows and the inner list define the column values per row. Each row must have the same number of elements.

The table can optionally have row and column headers, which are configured using *rowLabels*, *rowColours*, *rowLoc* and *colLabels*, *colColours*, *colLoc* respectively.

For finer grained control over tables, use the *Table* class and add it to the axes with *Axes.add_table*.

Parameters

cellText

[2D list of str, optional] The texts to place into the table cells.

Note: Line breaks in the strings are currently not accounted for and will result in the text exceeding the cell boundaries.

cellColours

[2D list of colors, optional] The background colors of the cells.

cellLoc

[{'left', 'center', 'right'}, default: 'right'] The alignment of the text within the cells.

colWidths

[list of float, optional] The column widths in units of the axes. If not given, all columns will have a width of *1 / ncols*.

rowLabels

[list of str, optional] The text of the row header cells.

rowColours

[list of colors, optional] The colors of the row header cells.

rowLoc

[{'left', 'center', 'right'}, default: 'left'] The text alignment of the row header cells.

colLabels

[list of str, optional] The text of the column header cells.

colColours

[list of colors, optional] The colors of the column header cells.

colLoc

[{'left', 'center', 'right'}, default: 'left'] The text alignment of the column header cells.

loc

[str, optional] The position of the cell with respect to *ax*. This must be one of the *codes*.

bbox

[*Bbox*, optional] A bounding box to draw the table into. If this is not *None*, this overrides *loc*.

edges

[substring of 'BRTL' or {'open', 'closed', 'horizontal', 'vertical'}]
The cell edges to be drawn with a line. See also *visible_edges*.

Returns

Table

The created table.

Other Parameters

**kwargs

Table properties.

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array
alpha	float or None
animated	bool
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
contains	unknown
figure	*Figure*
fontsize	float
gid	str
in_layout	bool
label	object
path_effects	*AbstractPathEffect*
picker	None or bool or callable
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
visible	bool
zorder	float

Examples using `matplotlib.pyplot.table`

- sphx_glr_gallery_misc_table_demo.py
- *Sample plots in Matplotlib*

matplotlib.pyplot.text

`matplotlib.pyplot.text`(*x*, *y*, *s*, *fontdict=None*, ***kwargs*)

Add text to the axes.

Add the text *s* to the axes at location *x*, *y* in data coordinates.

Parameters

x, y

[float] The position to place the text. By default, this is in data coordinates. The coordinate system can be changed using the *transform* parameter.

s

[str] The text.

fontdict

[dict, default: None] A dictionary to override the default text properties. If fontdict is None, the defaults are determined by *rcParams*.

Returns

Text

The created *Text* instance.

Other Parameters

****kwargs**

[*Text* properties.] Other miscellaneous text parameters.

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array an
alpha	float or None
animated	bool
backgroundcolor	color
bbox	dict with properties for *patches.FancyBboxPatch*
clip_box	*Bbox*

Tabl

Property	Description
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color or c	color
contains	unknown
figure	*Figure*
fontfamily or family	{FONTNAME, 'serif', 'sans-serif', 'cursive', 'fantasy', '
fontproperties or font or font_properties	*font_manager.FontProperties* or `str` or `pathlib.Path`
fontsize or size	float or {'xx-small', 'x-small', 'small', 'medium', 'large',
fontstretch or stretch	{a numeric value in range 0-1000, 'ultra-condensed',
fontstyle or style	{'normal', 'italic', 'oblique'}
fontvariant or variant	{'normal', 'small-caps'}
fontweight or weight	{a numeric value in range 0-1000, 'ultralight', 'light',
gid	str
horizontalalignment or ha	{'center', 'right', 'left'}
in_layout	bool
label	object
linespacing	float (multiple of font size)
multialignment or ma	{'left', 'right', 'center'}
path_effects	*AbstractPathEffect*
picker	None or bool or callable
position	(float, float)
rasterized	bool or None
rotation	float or {'vertical', 'horizontal'}
rotation_mode	{None, 'default', 'anchor'}
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
text	object
transform	*Transform*
url	str
usetex	bool or None
verticalalignment or va	{'center', 'top', 'bottom', 'baseline', 'center_baseline'}
visible	bool
wrap	bool
x	float
y	float
zorder	float

Examples

Individual keyword arguments can be used to override any given parameter:

```
>>> text(x, y, s, fontsize=12)
```

The default transform specifies that text is in data coords, alternatively, you can specify text in axis coords ((0, 0) is lower-left and (1, 1) is upper-right). The example below places text in the center of the axes:

```
>>> text(0.5, 0.5, 'matplotlib', horizontalalignment='center',
...      verticalalignment='center', transform=ax.transAxes)
```

You can put a rectangular box around the text instance (e.g., to set a background color) by using the keyword *bbox*. *bbox* is a dictionary of *Rectangle* properties. For example:

```
>>> text(x, y, s, bbox=dict(facecolor='red', alpha=0.5))
```

Examples using `matplotlib.pyplot.text`

- sphx_glr_gallery_lines_bars_and_markers_horizontal_barchart_distribution.py
- sphx_glr_gallery_text_labels_and_annotations_arrow_demo.py
- sphx_glr_gallery_text_labels_and_annotations_autowrap.py
- sphx_glr_gallery_text_labels_and_annotations_fancytextbox_demo.py
- sphx_glr_gallery_text_labels_and_annotations_multiline.py
- sphx_glr_gallery_text_labels_and_annotations_text_fontdict.py
- sphx_glr_gallery_pyplots_pyplot_mathtext.py
- sphx_glr_gallery_pyplots_pyplot_text.py
- sphx_glr_gallery_shapes_and_collections_artist_reference.py
- sphx_glr_gallery_event_handling_close_event.py
- sphx_glr_gallery_misc_transoffset.py
- sphx_glr_gallery_userdemo_pgf_fonts.py
- sphx_glr_gallery_userdemo_pgf_texsystem.py
- *Pyplot tutorial*
- *Path effects guide*
- *Text properties and layout*
- *Annotations*

matplotlib.pyplot.thetagrids

matplotlib.pyplot.**thetagrids**(*angles=None*, *labels=None*, *fmt=None*, ***kwargs*)

Get or set the theta gridlines on the current polar plot.

Call signatures:

```
lines, labels = thetagrids()
lines, labels = thetagrids(angles, labels=None, fmt=None, **kwargs)
```

When called with no arguments, *thetagrids* simply returns the tuple (*lines*, *labels*). When called with arguments, the labels will appear at the specified angles.

Parameters

angles

[tuple with floats, degrees] The angles of the theta gridlines.

labels

[tuple with strings or None] The labels to use at each radial gridline. The *projections.polar.ThetaFormatter* will be used if None.

fmt

[str or None] Format string used in *matplotlib.ticker.FormatStrFormatter*. For example '%f'. Note that the angle in radians will be used.

Returns

lines

[list of *lines.Line2D*] The theta gridlines.

labels

[list of *text.Text*] The tick labels.

Other Parameters

****kwargs**

kwargs are optional *Text* properties for the labels.

See also:

pyplot.rgrids

projections.polar.PolarAxes.set_thetagrids

Axis.get_gridlines

Axis.get_ticklabels

Examples

```
# set the locations of the angular gridlines
lines, labels = thetagrids(range(45, 360, 90))

# set the locations and labels of the angular gridlines
lines, labels = thetagrids(range(45, 360, 90), ('NE', 'NW', 'SW', 'SE'))
```

Examples using `matplotlib.pyplot.thetagrids`

matplotlib.pyplot.tick_params

matplotlib.pyplot.**tick_params**(*axis='both', **kwargs*)

Change the appearance of ticks, tick labels, and gridlines.

Tick properties that are not explicitly set using the keyword arguments remain unchanged unless *reset* is True.

Parameters

axis

[{'x', 'y', 'both'}, default: 'both'] The axis to which the parameters are applied.

which

[{'major', 'minor', 'both'}, default: 'major'] The group of ticks to which the parameters are applied.

reset

[bool, default: False] Whether to reset the ticks to defaults before updating them.

Other Parameters

direction

[{'in', 'out', 'inout'}] Puts ticks inside the axes, outside the axes, or both.

length

[float] Tick length in points.

width

[float] Tick width in points.

color

[color] Tick color.

pad

[float] Distance in points between tick and label.

labelsize

[float or str] Tick label font size in points or as a string (e.g., 'large').

labelcolor

[color] Tick label color.

colors

[color] Tick color and label color.

zorder

[float] Tick and label zorder.

bottom, top, left, right

[bool] Whether to draw the respective ticks.

labelbottom, labeltop, labelleft, labelright

[bool] Whether to draw the respective tick labels.

labelrotation

[float] Tick label rotation

grid_color

[color] Gridline color.

grid_alpha

[float] Transparency of gridlines: 0 (transparent) to 1 (opaque).

grid_linewidth

[float] Width of gridlines in points.

grid_linestyle

[str] Any valid *Line2D* line style spec.

Examples

```
ax.tick_params(direction='out', length=6, width=2, colors='r',
               grid_color='r', grid_alpha=0.5)
```

This will make all major ticks be red, pointing out of the box, and with dimensions 6 points by 2 points. Tick labels will also be red. Gridlines will be red and translucent.

Examples using `matplotlib.pyplot.tick_params`

matplotlib.pyplot.ticklabel_format

matplotlib.pyplot.**ticklabel_format**(*, *axis='both'*, *style=''*, *scilimits=None*, *useOffset=None*, *useLocale=None*, *useMathText=None*)

Configure the *ScalarFormatter* used by default for linear axes.

If a parameter is not set, the corresponding property of the formatter is left unchanged.

Parameters

axis

[{'x', 'y', 'both'}, default: 'both'] The axes to configure. Only major ticks are affected.

style

[{'sci', 'scientific', 'plain'}] Whether to use scientific notation. The formatter default is to use scientific notation.

scilimits

[pair of ints (m, n)] Scientific notation is used only for numbers outside the range 10^m to 10^n (and only if the formatter is configured to use scientific notation at all). Use (0, 0) to include all numbers. Use (m, m) where m != 0 to fix the order of magnitude to 10^m. The formatter default is `rcParams["axes.formatter.limits"]` (default: `[-5, 6]`).

useOffset

[bool or float] If True, the offset is calculated as needed. If False, no offset is used. If a numeric value, it sets the offset. The formatter default is `rcParams["axes.formatter.useoffset"]` (default: `True`).

useLocale

[bool] Whether to format the number using the current locale or using the C (English) locale. This affects e.g. the decimal separator. The formatter default is `rcParams["axes.formatter.use_locale"]` (default: `False`).

useMathText

[bool] Render the offset and scientific notation in mathtext. The formatter default is `rcParams["axes.formatter.use_mathtext"]` (default: `False`).

Raises

AttributeError

If the current formatter is not a *ScalarFormatter*.

Examples using `matplotlib.pyplot.ticklabel_format`

matplotlib.pyplot.tight_layout

`matplotlib.pyplot.`**`tight_layout`**(*, *pad=1.08*, *h_pad=None*, *w_pad=None*, *rect=None*)

Adjust the padding between and around subplots.

Parameters

pad

[float, default: 1.08] Padding between the figure edge and the edges of subplots, as a fraction of the font size.

h_pad, w_pad

[float, default: *pad*] Padding (height/width) between edges of adjacent subplots, as a fraction of the font size.

rect

[tuple (left, bottom, right, top), default: (0, 0, 1, 1)] A rectangle in normalized figure coordinates into which the whole subplots area (including labels) will fit.

Examples using `matplotlib.pyplot.tight_layout`

- sphx_glr_gallery_userdemo_pgf_preamble_sgskip.py

matplotlib.pyplot.title

matplotlib.pyplot.**title**(*label*, *fontdict=None*, *loc=None*, *pad=None*, ***, *y=None*, ***kwargs*)

Set a title for the axes.

Set one of the three available axes titles. The available titles are positioned above the axes in the center, flush with the left edge, and flush with the right edge.

Parameters

label

[str] Text to use for the title

fontdict

[dict] A dictionary controlling the appearance of the title text, the default *fontdict* is:

```
{'fontsize': rcParams['axes.titlesize'],
 'fontweight': rcParams['axes.titleweight'],
 'color': rcParams['axes.titlecolor'],
 'verticalalignment': 'baseline',
 'horizontalalignment': loc}
```

loc

[{'center', 'left', 'right'}, default: `rcParams["axes.titlelocation"]` (default: `'center'`)] Which title to set.

y

[float, default: `rcParams["axes.titley"]` (default: `None`)] Vertical axes loation for the title (1.0 is the top). If None (the default), y is determined automatically to avoid decorators on the axes.

pad

[float, default: `rcParams["axes.titlepad"]` (default: `6.0`)] The offset of the title from the top of the axes, in points.

Returns

Text

The matplotlib text instance representing the title

Other Parameters

****kwargs**

[*Text* properties] Other keyword arguments are text properties, see *Text* for a list of valid text properties.

Examples using `matplotlib.pyplot.title`

- sphx_glr_gallery_event_handling_ginput_manual_clabel_sgskip.py
- sphx_glr_gallery_user_interfaces_svg_histogram_sgskip.py

matplotlib.pyplot.tricontour

`matplotlib.pyplot.tricontour`(**args*, ***kwargs*)

Draw contour lines on an unstructured triangular grid.

The triangulation can be specified in one of two ways; either

```
tricontour(triangulation, ...)
```

where *triangulation* is a *Triangulation* object, or

```
tricontour(x, y, ...)
tricontour(x, y, triangles, ...)
tricontour(x, y, triangles=triangles, ...)
tricontour(x, y, mask=mask, ...)
tricontour(x, y, triangles, mask=mask, ...)
```

in which case a *Triangulation* object will be created. See that class' docstring for an explanation of these cases.

The remaining arguments may be:

```
tricontour(..., Z)
```

where *Z* is the array of values to contour, one per point in the triangulation. The level values are chosen automatically.

```
tricontour(..., Z, levels)
```

contour up to *levels+1* automatically chosen contour levels (*levels* intervals).

```
tricontour(..., Z, levels)
```

draw contour lines at the values specified in sequence *levels*, which must be in increasing order.

```
tricontour(Z, **kwargs)
```

Use keyword arguments to control colors, linewidth, origin, cmap ... see below for more details.

Parameters

triangulation

[*Triangulation*, optional] The unstructured triangular grid.

If specified, then *x*, *y*, *triangles*, and *mask* are not accepted.

x, y

[array-like, optional] The coordinates of the values in *Z*.

triangles

[int array-like of shape (ntri, 3), optional] For each triangle, the indices of the three points that make up the triangle, ordered in an anticlockwise manner. If not specified, the Delaunay triangulation is calculated.

mask

[bool array-like of shape (ntri), optional] Which triangles are masked out.

Z

[array-like(N, M)] The height values over which the contour is drawn.

levels

[int or array-like, optional] Determines the number and positions of the contour lines / regions.

If an int *n*, use *MaxNLocator*, which tries to automatically choose no more than *n+1* "nice" contour levels between *vmin* and *vmax*.

If array-like, draw contour lines at the specified levels. The values must be in increasing order.

Returns

TriContourSet

Other Parameters

colors

[color string or sequence of colors, optional] The colors of the levels, i.e., the contour lines.

The sequence is cycled for the levels in ascending order. If the sequence is shorter than the number of levels, it's repeated.

As a shortcut, single color strings may be used in place of one-element lists, i.e. `'red'` instead of `['red']` to color all levels with the same color. This shortcut does only work for color strings, not for other ways of specifying colors.

By default (value *None*), the colormap specified by *cmap* will be used.

alpha

[float, default: 1] The alpha blending value, between 0 (transparent) and 1 (opaque).

cmap

[str or *Colormap*, default: `rcParams["image.cmap"]` (default: `'viridis'`)] A *Colormap* instance or registered colormap name. The colormap maps the level values to colors.

If both *colors* and *cmap* are given, an error is raised.

norm

[*Normalize*, optional] If a colormap is used, the *Normalize* instance scales the level values to the canonical colormap range [0, 1] for mapping to colors. If not given, the default linear scaling is used.

origin

[{*None*, 'upper', 'lower', 'image'}, default: None] Determines the orientation and exact position of *Z* by specifying the position of `Z[0, 0]`. This is only relevant, if *X*, *Y* are not given.

- *None*: `Z[0, 0]` is at X=0, Y=0 in the lower left corner.
- 'lower': `Z[0, 0]` is at X=0.5, Y=0.5 in the lower left corner.
- 'upper': `Z[0, 0]` is at X=N+0.5, Y=0.5 in the upper left corner.
- 'image': Use the value from `rcParams["image.origin"]` (default: `'upper'`).

extent

[(x0, x1, y0, y1), optional] If *origin* is not *None*, then *extent* is interpreted as in *imshow*: it gives the outer pixel boundaries. In this case, the position of Z[0, 0] is the center of the pixel, not a corner. If *origin* is *None*, then (*x0*, *y0*) is the position of Z[0, 0], and (*x1*, *y1*) is the position of Z[-1, -1].

This argument is ignored if *X* and *Y* are specified in the call to contour.

locator

[ticker.Locator subclass, optional] The locator is used to determine the contour levels if they are not given explicitly via *levels*. Defaults to *MaxNLocator*.

extend

[{'neither', 'both', 'min', 'max'}, default: 'neither'] Determines the `tricontour`-coloring of values that are outside the *levels* range.

If 'neither', values outside the *levels* range are not colored. If 'min', 'max' or 'both', color the values below, above or below and above the *levels* range.

Values below `min(levels)` and above `max(levels)` are mapped to the under/over values of the *Colormap*. Note that most colormaps do not have dedicated colors for these by default, so that the over and under values are the edge values of the colormap. You may want to set these values explicitly using *Colormap.set_under* and *Colormap.set_over*.

Note: An existing *TriContourSet* does not get notified if properties of its colormap are changed. Therefore, an explicit call to *ContourSet.changed()* is needed after modifying the colormap. The explicit call can be left out, if a colorbar is assigned to the *TriContourSet* because it internally calls *ContourSet.changed()*.

---

xunits, yunits

[registered units, optional] Override axis units by specifying an instance of a *matplotlib.units.ConversionInterface*.

linewidths

[float or array-like, default: `rcParams["contour.linewidth"]` (default: `None`)] The line width of the contour lines.

If a number, all levels will be plotted with this linewidth.

If a sequence, the levels in ascending order will be plotted with the linewidths in the order specified.

If None, this falls back to `rcParams["lines.linewidth"]` (default: `1.5`).

linestyles

[{*None*, 'solid', 'dashed', 'dashdot', 'dotted'}, optional] If *linestyles* is *None*, the default is 'solid' unless the lines are monochrome. In that case, negative contours will take their

linestyle from `rcParams["contour.negative_linestyle"]` (default: `'dashed'`) setting.

linestyles can also be an iterable of the above strings specifying a set of linestyles to be used. If this iterable is shorter than the number of contour levels it will be repeated as necessary.

Examples using `matplotlib.pyplot.tricontour`

- sphx_glr_gallery_images_contours_and_fields_irregulardatagrid.py
- sphx_glr_gallery_images_contours_and_fields_tricontour_smooth_delaunay.py
- sphx_glr_gallery_images_contours_and_fields_tricontour_smooth_user.py
- sphx_glr_gallery_images_contours_and_fields_trigradient_demo.py

matplotlib.pyplot.tricontourf

matplotlib.pyplot.**tricontourf**(**args*, ***kwargs*)

Draw contour regions on an unstructured triangular grid.

The triangulation can be specified in one of two ways; either

```
tricontourf(triangulation, ...)
```

where *triangulation* is a *Triangulation* object, or

```
tricontourf(x, y, ...)
tricontourf(x, y, triangles, ...)
tricontourf(x, y, triangles=triangles, ...)
tricontourf(x, y, mask=mask, ...)
tricontourf(x, y, triangles, mask=mask, ...)
```

in which case a *Triangulation* object will be created. See that class' docstring for an explanation of these cases.

The remaining arguments may be:

```
tricontourf(..., Z)
```

where *Z* is the array of values to contour, one per point in the triangulation. The level values are chosen automatically.

```
tricontourf(..., Z, levels)
```

contour up to *levels+1* automatically chosen contour levels (*levels* intervals).

```
tricontourf(..., Z, levels)
```

draw contour regions at the values specified in sequence *levels*, which must be in increasing order.

```
tricontourf(Z, **kwargs)
```

Use keyword arguments to control colors, linewidth, origin, cmap ... see below for more details.

Parameters

triangulation

[*Triangulation*, optional] The unstructured triangular grid.

If specified, then *x*, *y*, *triangles*, and *mask* are not accepted.

x, y

[array-like, optional] The coordinates of the values in *Z*.

triangles

[int array-like of shape (ntri, 3), optional] For each triangle, the indices of the three points that make up the triangle, ordered in an anticlockwise manner. If not specified, the Delaunay triangulation is calculated.

mask

[bool array-like of shape (ntri), optional] Which triangles are masked out.

Z

[array-like(N, M)] The height values over which the contour is drawn.

levels

[int or array-like, optional] Determines the number and positions of the contour lines / regions.

If an int *n*, use *MaxNLocator*, which tries to automatically choose no more than *n+1* "nice" contour levels between *vmin* and *vmax*.

If array-like, draw contour lines at the specified levels. The values must be in increasing order.

Returns

TriContourSet

Other Parameters

colors

[color string or sequence of colors, optional] The colors of the levels, i.e., the contour regions.

The sequence is cycled for the levels in ascending order. If the sequence is shorter than the number of levels, it's repeated.

As a shortcut, single color strings may be used in place of one-element lists, i.e. `'red'` instead of `['red']` to color all levels with the same color. This shortcut does only work for color strings, not for other ways of specifying colors.

By default (value *None*), the colormap specified by *cmap* will be used.

alpha

[float, default: 1] The alpha blending value, between 0 (transparent) and 1 (opaque).

cmap

[str or *Colormap*, default: `rcParams["image.cmap"]` (default: `'viridis'`)] A *Colormap* instance or registered colormap name. The colormap maps the level values to colors.

If both *colors* and *cmap* are given, an error is raised.

norm

[*Normalize*, optional] If a colormap is used, the *Normalize* instance scales the level values to the canonical colormap range [0, 1] for mapping to colors. If not given, the default linear scaling is used.

origin

[{*None*, 'upper', 'lower', 'image'}, default: None] Determines the orientation and exact position of *Z* by specifying the position of `Z[0, 0]`. This is only relevant, if *X*, *Y* are not given.

- *None*: `Z[0, 0]` is at X=0, Y=0 in the lower left corner.
- 'lower': `Z[0, 0]` is at X=0.5, Y=0.5 in the lower left corner.
- 'upper': `Z[0, 0]` is at X=N+0.5, Y=0.5 in the upper left corner.
- 'image': Use the value from `rcParams["image.origin"]` (default: `'upper'`).

extent

[(x0, x1, y0, y1), optional] If *origin* is not *None*, then *extent* is interpreted as in *imshow*: it gives the outer pixel boundaries. In this case, the position of Z[0, 0] is the center of the pixel, not a corner. If *origin* is *None*, then (*x0*, *y0*) is the position of Z[0, 0], and (*x1*, *y1*) is the position of Z[-1, -1].

This argument is ignored if *X* and *Y* are specified in the call to contour.

locator

[ticker.Locator subclass, optional] The locator is used to determine the contour levels if they are not given explicitly via *levels*. Defaults to *MaxNLocator*.

extend

[{'neither', 'both', 'min', 'max'}, default: 'neither'] Determines the `tricontourf`-coloring of values that are outside the *levels* range.

If 'neither', values outside the *levels* range are not colored. If 'min', 'max' or 'both', color the values below, above or below and above the *levels* range.

Values below `min(levels)` and above `max(levels)` are mapped to the under/over values of the *Colormap*. Note that most colormaps do not have dedicated colors for these by default, so that the over and under values are the edge values of the colormap. You may want to set these values explicitly using *Colormap.set_under* and *Colormap.set_over*.

Note: An existing *TriContourSet* does not get notified if properties of its colormap are changed. Therefore, an explicit call to *ContourSet.changed()* is needed after modifying the colormap. The explicit call can be left out, if a colorbar is assigned to the *TriContourSet* because it internally calls *ContourSet.changed()*.

---

xunits, yunits

[registered units, optional] Override axis units by specifying an instance of a *matplotlib.units.ConversionInterface*.

antialiased

[bool, default: True] Whether to use antialiasing.

Notes

tricontourf fills intervals that are closed at the top; that is, for boundaries *z1* and *z2*, the filled region is:

```
z1 < Z <= z2
```

except for the lowest interval, which is closed on both sides (i.e. it includes the lowest value).

Examples using `matplotlib.pyplot.tricontourf`

- sphx_glr_gallery_images_contours_and_fields_irregulardatagrid.py
- sphx_glr_gallery_images_contours_and_fields_tricontour_demo.py
- sphx_glr_gallery_images_contours_and_fields_tricontour_smooth_delaunay.py
- sphx_glr_gallery_images_contours_and_fields_tricontour_smooth_user.py
- sphx_glr_gallery_images_contours_and_fields_triinterp_demo.py

matplotlib.pyplot.tripcolor

matplotlib.pyplot.**tripcolor**(**args*, *alpha=1.0*, *norm=None*, *cmap=None*, *vmin=None*, *vmax=None*, *shading='flat'*, *facecolors=None*, ***kwargs*)

Create a pseudocolor plot of an unstructured triangular grid.

The triangulation can be specified in one of two ways; either:

```
tripcolor(triangulation, ...)
```

where triangulation is a *Triangulation* object, or

```
tripcolor(x, y, ...)
tripcolor(x, y, triangles, ...)
tripcolor(x, y, triangles=triangles, ...)
tripcolor(x, y, mask=mask, ...)
tripcolor(x, y, triangles, mask=mask, ...)
```

in which case a Triangulation object will be created. See *Triangulation* for a explanation of these possibilities.

The next argument must be *C*, the array of color values, either one per point in the triangulation if color values are defined at points, or one per triangle in the triangulation if color values are defined at triangles. If there are the same number of points and triangles in the triangulation it is assumed that color values are defined at points; to force the use of color values at triangles use the kwarg `facecolors=C` instead of just `C`.

shading may be 'flat' (the default) or 'gouraud'. If *shading* is 'flat' and C values are defined at points, the color values used for each triangle are from the mean C of the triangle's three points. If *shading* is 'gouraud' then color values must be defined at points.

The remaining kwargs are the same as for *pcolor*.

Examples using `matplotlib.pyplot.tripcolor`

- sphx_glr_gallery_images_contours_and_fields_tripcolor_demo.py

matplotlib.pyplot.triplot

matplotlib.pyplot.**triplot**(**args*, ***kwargs*)

Draw a unstructured triangular grid as lines and/or markers.

The triangulation to plot can be specified in one of two ways; either:

```
triplot(triangulation, ...)
```

where triangulation is a *Triangulation* object, or

```
triplot(x, y, ...)
triplot(x, y, triangles, ...)
triplot(x, y, triangles=triangles, ...)
triplot(x, y, mask=mask, ...)
triplot(x, y, triangles, mask=mask, ...)
```

in which case a Triangulation object will be created. See *Triangulation* for a explanation of these possibilities.

The remaining args and kwargs are the same as for *plot*.

Returns

lines

[*Line2D*] The drawn triangles edges.

markers

[*Line2D*] The drawn marker nodes.

Examples using `matplotlib.pyplot.triplot`

- sphx_glr_gallery_images_contours_and_fields_tricontour_smooth_delaunay.py
- sphx_glr_gallery_images_contours_and_fields_trigradient_demo.py
- sphx_glr_gallery_images_contours_and_fields_triinterp_demo.py
- sphx_glr_gallery_images_contours_and_fields_triplot_demo.py
- sphx_glr_gallery_event_handling_trifinder_event_demo.py

matplotlib.pyplot.twinx

matplotlib.pyplot.**twinx**(*ax=None*)

Make and return a second axes that shares the *x*-axis. The new axes will overlay *ax* (or the current axes if *ax* is *None*), and its ticks will be on the right.

Examples

/gallery/subplots_axes_and_figures/two_scales

Examples using `matplotlib.pyplot.twinx`

matplotlib.pyplot.twiny

matplotlib.pyplot.**twiny**(*ax=None*)

Make and return a second axes that shares the *y*-axis. The new axes will overlay *ax* (or the current axes if *ax* is *None*), and its ticks will be on the top.

Examples

/gallery/subplots_axes_and_figures/two_scales

Examples using `matplotlib.pyplot.twiny`

matplotlib.pyplot.uninstall_repl_displayhook

matplotlib.pyplot.**uninstall_repl_displayhook**()

Uninstall the matplotlib display hook.

> **Warning:** Need IPython >= 2 for this to work. For IPython < 2 will raise a `NotImplementedError`

> **Warning:** If you are using vanilla python and have installed another display hook this will reset `sys.displayhook` to what ever function was there when matplotlib installed it's displayhook, possibly discarding your changes.

Examples using `matplotlib.pyplot.uninstall_repl_displayhook`

matplotlib.pyplot.violinplot

`matplotlib.pyplot.violinplot`(*dataset*, *positions=None*, *vert=True*, *widths=0.5*, *showmeans=False*, *showextrema=True*, *showmedians=False*, *quantiles=None*, *points=100*, *bw_method=None*, *, *data=None*)

Make a violin plot.

Make a violin plot for each column of *dataset* or each vector in sequence *dataset*. Each filled area extends to represent the entire data range, with optional lines at the mean, the median, the minimum, the maximum, and user-specified quantiles.

Parameters

dataset

[Array or a sequence of vectors.] The input data.

positions

[array-like, default: [1, 2, ..., n]] Sets the positions of the violins. The ticks and limits are automatically set to match the positions.

vert

[bool, default: True.] If true, creates a vertical violin plot. Otherwise, creates a horizontal violin plot.

widths

[array-like, default: 0.5] Either a scalar or a vector that sets the maximal width of each violin. The default is 0.5, which uses about half of the available horizontal space.

showmeans

[bool, default: False] If `True`, will toggle rendering of the means.

showextrema

[bool, default: True] If `True`, will toggle rendering of the extrema.

showmedians

[bool, default: False] If `True`, will toggle rendering of the medians.

quantiles

[array-like, default: None] If not None, set a list of floats in interval [0, 1] for each violin, which stands for the quantiles that will be rendered for that violin.

points

[int, default: 100] Defines the number of points to evaluate each of the gaussian kernel density estimations at.

bw_method

[str, scalar or callable, optional] The method used to calculate the estimator bandwidth. This can be 'scott', 'silverman', a scalar constant or a callable. If a scalar, this will be used directly as `kde.factor`. If a callable, it should take a `GaussianKDE` instance as its only parameter and return a scalar. If None (default), 'scott' is used.

Returns

dict

A dictionary mapping each component of the violinplot to a list of the corresponding collection instances created. The dictionary has the following keys:

- `bodies`: A list of the *PolyCollection* instances containing the filled area of each violin.
- `cmeans`: A *LineCollection* instance that marks the mean values of each of the violin's distribution.
- `cmins`: A *LineCollection* instance that marks the bottom of each violin's distribution.
- `cmaxes`: A *LineCollection* instance that marks the top of each violin's distribution.
- `cbars`: A *LineCollection* instance that marks the centers of each violin's distribution.
- `cmedians`: A *LineCollection* instance that marks the median values of each of the violin's distribution.
- `cquantiles`: A *LineCollection* instance created to identify the quantile values of each of the violin's distribution.

Notes

Note: In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, the following arguments can also be string `s`, which is interpreted as `data[s]` (unless this raises an exception): *dataset*.

Objects passed as **data** must support item access (`data[s]`) and membership test (`s in data`).

Examples using `matplotlib.pyplot.violinplot`

matplotlib.pyplot.viridis

matplotlib.pyplot.**viridis**()

Set the colormap to "viridis".

This changes the default colormap as well as the colormap of the current image if there is one. See `help(colormaps)` for more information.

Examples using `matplotlib.pyplot.viridis`

matplotlib.pyplot.vlines

matplotlib.pyplot.**vlines**(*x*, *ymin*, *ymax*, *colors=None*, *linestyles='solid'*, *label=''*, ***, *data=None*, ***kwargs*)

Plot vertical lines.

Plot vertical lines at each *x* from *ymin* to *ymax*.

Parameters

x

[float or array-like] x-indexes where to plot the lines.

ymin, ymax

[float or array-like] Respective beginning and end of each line. If scalars are provided, all lines will have same length.

colors

[list of colors, default: `rcParams["lines.color"]` (default: `'C0'`)]

linestyles

[{'solid', 'dashed', 'dashdot', 'dotted'}, optional]

label

[str, default: '']

Returns

LineCollection

Other Parameters

****kwargs**

[*LineCollection* properties.]

See also:

hlines

horizontal lines

axvline

vertical line across the axes

Notes

Note: In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, the following arguments can also be string `s`, which is interpreted as `data[s]` (unless this raises an exception): *x*, *ymin*, *ymax*, *colors*.

Objects passed as **data** must support item access (`data[s]`) and membership test (`s in data`).

Examples using `matplotlib.pyplot.vlines`

matplotlib.pyplot.waitforbuttonpress

matplotlib.pyplot.**waitforbuttonpress**(*timeout=-1*)

Blocking call to interact with the figure.

Wait for user input and return True if a key was pressed, False if a mouse button was pressed and None if no input was given within *timeout* seconds. Negative values deactivate *timeout*.

Examples using `matplotlib.pyplot.waitforbuttonpress`

- sphx_glr_gallery_event_handling_ginput_manual_clabel_sgskip.py

matplotlib.pyplot.winter

matplotlib.pyplot.**winter**()

Set the colormap to "winter".

This changes the default colormap as well as the colormap of the current image if there is one. See `help(colormaps)` for more information.

Examples using `matplotlib.pyplot.winter`

matplotlib.pyplot.xcorr

matplotlib.pyplot.**xcorr**(*x*, *y*, *normed=True*, *detrend=<function detrend_none at 0x7f5434697820>*, *usevlines=True*, *maxlags=10*, ***, *data=None*, ***kwargs*)

Plot the cross correlation between *x* and *y*.

The correlation with lag k is defined as $\sum_n x[n+k] \cdot y^*[n]$, where y^* is the complex conjugate of y.

Parameters

x, y

[array-like of length n]

detrend

[callable, default: *mlab.detrend_none* (no detrending)] A detrending function applied to *x* and *y*. It must have the signature

```
detrend(x: np.ndarray) -> np.ndarray
```

normed

[bool, default: True] If `True`, input vectors are normalised to unit length.

usevlines

[bool, default: True] Determines the plot style.

If `True`, vertical lines are plotted from 0 to the xcorr value using *Axes.vlines*. Additionally, a horizontal line is plotted at y=0 using *Axes.axhline*.

If `False`, markers are plotted at the xcorr values using *Axes.plot*.

maxlags

[int, default: 10] Number of lags to show. If None, will return all `2 * len(x) - 1` lags.

Returns

lags

[array (length `2*maxlags+1`)] The lag vector.

c

[array (length `2*maxlags+1`)] The auto correlation vector.

line

[*LineCollection* or *Line2D*] *Artist* added to the axes of the correlation:

- *LineCollection* if *usevlines* is True.
- *Line2D* if *usevlines* is False.

b

[*Line2D* or None] Horizontal line at 0 if *usevlines* is True None *usevlines* is False.

Other Parameters

linestyle

[*Line2D* property, optional] The linestyle for plotting the data points. Only used if *usevlines* is `False`.

marker

[str, default: 'o'] The marker for plotting the data points. Only used if *usevlines* is `False`.

****kwargs**

Additional parameters are passed to *Axes.vlines* and *Axes.axhline* if *usevlines* is `True`; otherwise they are passed to *Axes.plot*.

Notes

The cross correlation is performed with `numpy.correlate` with `mode = "full"`.

Note: In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, the following arguments can also be string `s`, which is interpreted as `data[s]` (unless this raises an exception): *x*, *y*.

Objects passed as **data** must support item access (`data[s]`) and membership test (`s in data`).

Examples using `matplotlib.pyplot.xcorr`

- sphx_glr_gallery_lines_bars_and_markers_xcorr_acorr_demo.py

matplotlib.pyplot.xkcd

`matplotlib.pyplot.`**xkcd**(*scale=1*, *length=100*, *randomness=2*)

Turn on xkcd sketch-style drawing mode. This will only have effect on things drawn after this function is called.

For best results, the "Humor Sans" font should be installed: it is not included with Matplotlib.

Parameters

scale

[float, optional] The amplitude of the wiggle perpendicular to the source line.

length

[float, optional] The length of the wiggle along the line.

randomness

[float, optional] The scale factor by which the length is shrunken or expanded.

Notes

This function works by a number of rcParams, so it will probably override others you have set before.

If you want the effects of this function to be temporary, it can be used as a context manager, for example:

```
with plt.xkcd():
    # This figure will be in XKCD-style
    fig1 = plt.figure()
    # ...

# This figure will be in regular style
fig2 = plt.figure()
```

Examples using `matplotlib.pyplot.xkcd`

- sphx_glr_gallery_showcase_xkcd.py

matplotlib.pyplot.xlabel

matplotlib.pyplot.**xlabel**(*xlabel*, *fontdict=None*, *labelpad=None*, ***, *loc=None*, ***kwargs*)

Set the label for the x-axis.

Parameters

xlabel

[str] The label text.

labelpad

[float, default: None] Spacing in points from the axes bounding box including ticks and tick labels.

loc

[{'left', 'center', 'right'}, default: `rcParams["xaxis.labellocation"]` (default: `'center'`)] The label position. This is a high-level alternative for passing parameters *x* and *horizontalalignment*.

Other Parameters

****kwargs**

[*Text* properties] *Text* properties control the appearance of the label.

See also:

text

Documents the properties supported by *Text*.

Examples using `matplotlib.pyplot.xlabel`

- sphx_glr_gallery_userdemo_pgf_preamble_sgskip.py

matplotlib.pyplot.xlim

matplotlib.pyplot.**xlim**(**args*, ***kwargs*)

Get or set the x limits of the current axes.

Call signatures:

```
left, right = xlim()  # return the current xlim
xlim((left, right))   # set the xlim to left, right
xlim(left, right)     # set the xlim to left, right
```

If you do not specify args, you can pass *left* or *right* as kwargs, i.e.:

```
xlim(right=3)  # adjust the right leaving left unchanged
xlim(left=1)  # adjust the left leaving right unchanged
```

Setting limits turns autoscaling off for the x-axis.

Returns

left, right

A tuple of the new x-axis limits.

Notes

Calling this function with no arguments (e.g. `xlim()`) is the pyplot equivalent of calling *get_xlim* on the current axes. Calling this function with arguments is the pyplot equivalent of calling *set_xlim* on the current axes. All arguments are passed though.

Examples using `matplotlib.pyplot.xlim`

- *Frame grabbing*
- sphx_glr_gallery_event_handling_ginput_manual_clabel_sgskip.py

matplotlib.pyplot.xscale

`matplotlib.pyplot.`**`xscale`**`(`*value*, ***kwargs*`)`

Set the x-axis scale.

Parameters

value

[{"linear", "log", "symlog", "logit", ...}] The axis scale type to apply.

****kwargs**

Different keyword arguments are accepted, depending on the scale. See the respective class keyword arguments:

- *matplotlib.scale.LinearScale*
- *matplotlib.scale.LogScale*
- *matplotlib.scale.SymmetricalLogScale*
- *matplotlib.scale.LogitScale*

Notes

By default, Matplotlib supports the above mentioned scales. Additionally, custom scales may be registered using *matplotlib.scale.register_scale*. These scales can then also be used here.

Examples using `matplotlib.pyplot.xscale`

- sphx_glr_gallery_scales_symlog_demo.py

matplotlib.pyplot.xticks

matplotlib.pyplot.**xticks**(*ticks=None*, *labels=None*, ***kwargs*)

Get or set the current tick locations and labels of the x-axis.

Pass no arguments to return the current values without modifying them.

Parameters

ticks

[array-like, optional] The list of xtick locations. Passing an empty list removes all xticks.

labels

[array-like, optional] The labels to place at the given *ticks* locations. This argument can only be passed if *ticks* is passed as well.

****kwargs**

Text properties can be used to control the appearance of the labels.

Returns

locs

The list of xtick locations.

labels

The list of xlabel *Text* objects.

Notes

Calling this function with no arguments (e.g. `xticks()`) is the pyplot equivalent of calling *get_xticks* and *get_xticklabels* on the current axes. Calling this function with arguments is the pyplot equivalent of calling *set_xticks* and *set_xticklabels* on the current axes.

Examples

```
>>> locs, labels = xticks()  # Get the current locations and labels.
>>> xticks(np.arange(0, 1, step=0.2))  # Set label locations.
>>> xticks(np.arange(3), ['Tom', 'Dick', 'Sue'])  # Set text labels.
>>> xticks([0, 1, 2], ['January', 'February', 'March'],
...        rotation=20)  # Set text labels and properties.
>>> xticks([])  # Disable xticks.
```

Examples using `matplotlib.pyplot.xticks`

- sphx_glr_gallery_subplots_axes_and_figures_secondary_axis.py
- sphx_glr_gallery_text_labels_and_annotations_arrow_demo.py
- sphx_glr_gallery_text_labels_and_annotations_multiline.py
- sphx_glr_gallery_axes_grid1_demo_colorbar_of_inset_axes.py
- sphx_glr_gallery_misc_table_demo.py
- sphx_glr_gallery_ticks_and_spines_ticklabels_rotation.py

matplotlib.pyplot.ylabel

`matplotlib.pyplot.ylabel`(*ylabel*, *fontdict=None*, *labelpad=None*, *, *loc=None*, ***kwargs*)

Set the label for the y-axis.

Parameters

ylabel

[str] The label text.

labelpad

[float, default: None] Spacing in points from the axes bounding box including ticks and tick labels.

loc

[{'bottom', 'center', 'top'}, default: `rcParams["yaxis.labellocation"]` (default: `'center'`)] The label position. This is a high-level alternative for passing parameters *y* and *horizontalalignment*.

Other Parameters

****kwargs**

[*Text* properties] *Text* properties control the appearance of the label.

See also:

text

Documents the properties supported by *Text*.

Examples using `matplotlib.pyplot.ylabel`

- sphx_glr_gallery_userdemo_pgf_preamble_sgskip.py

matplotlib.pyplot.ylim

`matplotlib.pyplot.ylim`(**args*, ***kwargs*)

Get or set the y-limits of the current axes.

Call signatures:

```
bottom, top = ylim()  # return the current ylim
ylim((bottom, top))   # set the ylim to bottom, top
ylim(bottom, top)     # set the ylim to bottom, top
```

If you do not specify args, you can alternatively pass *bottom* or *top* as kwargs, i.e.:

```
ylim(top=3)  # adjust the top leaving bottom unchanged
ylim(bottom=1)  # adjust the bottom leaving top unchanged
```

Setting limits turns autoscaling off for the y-axis.

Returns

bottom, top

A tuple of the new y-axis limits.

Notes

Calling this function with no arguments (e.g. `ylim()`) is the pyplot equivalent of calling *get_ylim* on the current axes. Calling this function with arguments is the pyplot equivalent of calling *set_ylim* on the current axes. All arguments are passed though.

Examples using `matplotlib.pyplot.ylim`

- *Frame grabbing*
- sphx_glr_gallery_event_handling_ginput_manual_clabel_sgskip.py

matplotlib.pyplot.yscale

`matplotlib.pyplot.yscale(`*value*, ***kwargs*`)`

Set the y-axis scale.

Parameters

value

[{"linear", "log", "symlog", "logit", ...}] The axis scale type to apply.

****kwargs**

Different keyword arguments are accepted, depending on the scale. See the respective class keyword arguments:

- *matplotlib.scale.LinearScale*
- *matplotlib.scale.LogScale*
- *matplotlib.scale.SymmetricalLogScale*
- *matplotlib.scale.LogitScale*

Notes

By default, Matplotlib supports the above mentioned scales. Additionally, custom scales may be registered using *matplotlib.scale.register_scale*. These scales can then also be used here.

Examples using `matplotlib.pyplot.yscale`

- sphx_glr_gallery_scales_symlog_demo.py
- *Pyplot tutorial*

matplotlib.pyplot.yticks

matplotlib.pyplot.**yticks**(*ticks=None*, *labels=None*, ***kwargs*)

Get or set the current tick locations and labels of the y-axis.

Pass no arguments to return the current values without modifying them.

Parameters

ticks

[array-like, optional] The list of ytick locations. Passing an empty list removes all yticks.

labels

[array-like, optional] The labels to place at the given *ticks* locations. This argument can only be passed if *ticks* is passed as well.

****kwargs**

Text properties can be used to control the appearance of the labels.

Returns

locs

The list of ytick locations.

labels

The list of ylabel *Text* objects.

Notes

Calling this function with no arguments (e.g. `yticks()`) is the pyplot equivalent of calling *get_yticks* and *get_yticklabels* on the current axes. Calling this function with arguments is the pyplot equivalent of calling *set_yticks* and *set_yticklabels* on the current axes.

Examples

```
>>> locs, labels = yticks()  # Get the current locations and labels.
>>> yticks(np.arange(0, 1, step=0.2))  # Set label locations.
>>> yticks(np.arange(3), ['Tom', 'Dick', 'Sue'])  # Set text labels.
>>> yticks([0, 1, 2], ['January', 'February', 'March'],
...        rotation=45)  # Set text labels and properties.
>>> yticks([])  # Disable yticks.
```

Examples using `matplotlib.pyplot.yticks`

- sphx_glr_gallery_text_labels_and_annotations_arrow_demo.py
- sphx_glr_gallery_axes_grid1_demo_colorbar_of_inset_axes.py
- sphx_glr_gallery_misc_table_demo.py

`matplotlib.pyplot.plotting()`

Function	Description
acorr	Plot the autocorrelation of *x*.
angle_spectrum	Plot the angle spectrum.
annotate	Annotate the point *xy* with text *text*.
arrow	Add an arrow to the axes.
autoscale	Autoscale the axis view to the data (toggle).
axes	Add an axes to the current figure and make it the current axes.
axhline	Add a horizontal line across the axis.
axhspan	Add a horizontal span (rectangle) across the axis.
axis	Convenience method to get or set some axis properties.
axline	Add an infinitely long straight line.
axvline	Add a vertical line across the axes.
axvspan	Add a vertical span (rectangle) across the axes.
bar	Make a bar plot.
barbs	Plot a 2D field of barbs.
barh	Make a horizontal bar plot.
box	Turn the axes box on or off on the current axes.
boxplot	Make a box and whisker plot.
broken_barh	Plot a horizontal sequence of rectangles.
cla	Clear the current axes.
clabel	Label a contour plot.
clf	Clear the current figure.
clim	Set the color limits of the current image.
close	Close a figure window.
cohere	Plot the coherence between *x* and *y*.
colorbar	Add a colorbar to a plot.

Table 211 – continued from previous page

Function	Description
contour	Plot contours.
contourf	Plot contours.
csd	Plot the cross-spectral density.
delaxes	Remove an *Axes* (defaulting to the current axes) from its figure.
draw	Redraw the current figure.
draw_if_interactive	
errorbar	Plot y versus x as lines and/or markers with attached errorbars.
eventplot	Plot identical parallel lines at the given positions.
figimage	Add a non-resampled image to the figure.
figlegend	Place a legend on the figure.
fignum_exists	Return whether the figure with the given id exists.
figtext	Add text to figure.
figure	Create a new figure, or activate an existing figure.
fill	Plot filled polygons.
fill_between	Fill the area between two horizontal curves.
fill_betweenx	Fill the area between two vertical curves.
findobj	Find artist objects.
gca	Get the current axes, creating one if necessary.
gcf	Get the current figure.
gci	Get the current colorable artist.
get	Return the value of an object's *property*, or print all of them.
get_figlabels	Return a list of existing figure labels.
get_fignums	Return a list of existing figure numbers.
getp	Return the value of an object's *property*, or print all of them.
grid	Configure the grid lines.
hexbin	Make a 2D hexagonal binning plot of points *x*, *y*.
hist	Plot a histogram.
hist2d	Make a 2D histogram plot.
hlines	Plot horizontal lines at each *y* from *xmin* to *xmax*.
imread	Read an image from a file into an array.
imsave	Save an array as an image file.
imshow	Display data as an image, i.e., on a 2D regular raster.
install_repl_displayhook	Install a repl display hook so that any stale figure are automatically
ioff	Turn the interactive mode off.
ion	Turn the interactive mode on.
isinteractive	Return if pyplot is in "interactive mode" or not.
legend	Place a legend on the axes.
locator_params	Control behavior of major tick locators.
loglog	Make a plot with log scaling on both the x and y axis.
magnitude_spectrum	Plot the magnitude spectrum.
margins	Set or retrieve autoscaling margins.
matshow	Display an array as a matrix in a new figure window.
minorticks_off	Remove minor ticks from the axes.

Table 211 – continued from previous page

Function	Description
minorticks_on	Display minor ticks on the axes.
new_figure_manager	Create a new figure manager instance.
pause	Run the GUI event loop for *interval* seconds.
pcolor	Create a pseudocolor plot with a non-regular rectangular grid.
pcolormesh	Create a pseudocolor plot with a non-regular rectangular grid.
phase_spectrum	Plot the phase spectrum.
pie	Plot a pie chart.
plot	Plot y versus x as lines and/or markers.
plot_date	Plot data that contains dates.
polar	Make a polar plot.
psd	Plot the power spectral density.
quiver	Plot a 2D field of arrows.
quiverkey	Add a key to a quiver plot.
rc	Set the current *rcParams*.
rc_context	Return a context manager for temporarily changing rcParams.
rcdefaults	Restore the *rcParams* from Matplotlib's internal default style.
rgrids	Get or set the radial gridlines on the current polar plot.
savefig	Save the current figure.
sca	Set the current Axes to *ax* and the current Figure to the parent of *a*
scatter	A scatter plot of *y* vs.
sci	Set the current image.
semilogx	Make a plot with log scaling on the x axis.
semilogy	Make a plot with log scaling on the y axis.
set_cmap	Set the default colormap, and applies it to the current image if any.
setp	Set a property on an artist object.
show	Display all open figures.
specgram	Plot a spectrogram.
spy	Plot the sparsity pattern of a 2D array.
stackplot	Draw a stacked area plot.
stem	Create a stem plot.
step	Make a step plot.
streamplot	Draw streamlines of a vector flow.
subplot	Add a subplot to the current figure.
subplot2grid	Create a subplot at a specific location inside a regular grid.
subplot_mosaic	Build a layout of Axes based on ASCII art or nested lists.
subplot_tool	Launch a subplot tool window for a figure.
subplots	Create a figure and a set of subplots.
subplots_adjust	Adjust the subplot layout parameters.
suptitle	Add a centered title to the figure.
switch_backend	Close all open figures and set the Matplotlib backend.
table	Add a table to an *Axes*.
text	Add text to the axes.
thetagrids	Get or set the theta gridlines on the current polar plot.

Table 211 – continued from previous page

Function	Description
tick_params	Change the appearance of ticks, tick labels, and gridlines.
ticklabel_format	Configure the *ScalarFormatter* used by default for linear axes.
tight_layout	Adjust the padding between and around subplots.
title	Set a title for the axes.
tricontour	Draw contour lines on an unstructured triangular grid.
tricontourf	Draw contour regions on an unstructured triangular grid.
tripcolor	Create a pseudocolor plot of an unstructured triangular grid.
triplot	Draw a unstructured triangular grid as lines and/or markers.
twinx	Make and return a second axes that shares the *x*-axis.
twiny	Make and return a second axes that shares the *y*-axis.
uninstall_repl_displayhook	Uninstall the matplotlib display hook.
violinplot	Make a violin plot.
vlines	Plot vertical lines.
xcorr	Plot the cross correlation between *x* and *y*.
xkcd	Turn on xkcd sketch-style drawing mode.
xlabel	Set the label for the x-axis.
xlim	Get or set the x limits of the current axes.
xscale	Set the x-axis scale.
xticks	Get or set the current tick locations and labels of the x-axis.
ylabel	Set the label for the y-axis.
ylim	Get or set the y-limits of the current axes.
yscale	Set the y-axis scale.
yticks	Get or set the current tick locations and labels of the y-axis.

18.38.2 Colors in Matplotlib

There are many colormaps you can use to map data onto color values. Below we list several ways in which color can be utilized in Matplotlib.

For a more in-depth look at colormaps, see the *Choosing Colormaps in Matplotlib* tutorial.

matplotlib.pyplot.**colormaps**()

Matplotlib provides a number of colormaps, and others can be added using *register_cmap()*. This function documents the built-in colormaps, and will also return a list of all registered colormaps if called.

You can set the colormap for an image, pcolor, scatter, etc, using a keyword argument:

```
imshow(X, cmap=cm.hot)
```

or using the *set_cmap()* function:

```
imshow(X)
pyplot.set_cmap('hot')
pyplot.set_cmap('jet')
```

In interactive mode, *set_cmap()* will update the colormap post-hoc, allowing you to see which one works best for your data.

All built-in colormaps can be reversed by appending `_r`: For instance, `gray_r` is the reverse of `gray`.

There are several common color schemes used in visualization:

Sequential schemes

for unipolar data that progresses from low to high

Diverging schemes

for bipolar data that emphasizes positive or negative deviations from a central value

Cyclic schemes

for plotting values that wrap around at the endpoints, such as phase angle, wind direction, or time of day

Qualitative schemes

for nominal data that has no inherent ordering, where color is used only to distinguish categories

Matplotlib ships with 4 perceptually uniform color maps which are the recommended color maps for sequential data:

Colormap	Description
inferno	perceptually uniform shades of black-red-yellow
magma	perceptually uniform shades of black-red-white
plasma	perceptually uniform shades of blue-red-yellow
viridis	perceptually uniform shades of blue-green-yellow

The following colormaps are based on the ColorBrewer color specifications and designs developed by Cynthia Brewer:

ColorBrewer Diverging (luminance is highest at the midpoint, and decreases towards differently-colored endpoints):

Colormap	Description
BrBG	brown, white, blue-green
PiYG	pink, white, yellow-green
PRGn	purple, white, green
PuOr	orange, white, purple
RdBu	red, white, blue
RdGy	red, white, gray
RdYlBu	red, yellow, blue
RdYlGn	red, yellow, green
Spectral	red, orange, yellow, green, blue

ColorBrewer Sequential (luminance decreases monotonically):

Colormap	Description
Blues	white to dark blue
BuGn	white, light blue, dark green
BuPu	white, light blue, dark purple
GnBu	white, light green, dark blue
Greens	white to dark green
Greys	white to black (not linear)
Oranges	white, orange, dark brown
OrRd	white, orange, dark red
PuBu	white, light purple, dark blue
PuBuGn	white, light purple, dark green
PuRd	white, light purple, dark red
Purples	white to dark purple
RdPu	white, pink, dark purple
Reds	white to dark red
YlGn	light yellow, dark green
YlGnBu	light yellow, light green, dark blue
YlOrBr	light yellow, orange, dark brown
YlOrRd	light yellow, orange, dark red

ColorBrewer Qualitative:

(For plotting nominal data, *ListedColormap* is used, not *LinearSegmentedColormap*. Different sets of colors are recommended for different numbers of categories.)

- Accent
- Dark2
- Paired
- Pastel1
- Pastel2
- Set1

- Set2
- Set3

A set of colormaps derived from those of the same name provided with Matlab are also included:

Colormap	Description
autumn	sequential linearly-increasing shades of red-orange-yellow
bone	sequential increasing black-white color map with a tinge of blue, to emulate X-ray film
cool	linearly-decreasing shades of cyan-magenta
copper	sequential increasing shades of black-copper
flag	repetitive red-white-blue-black pattern (not cyclic at endpoints)
gray	sequential linearly-increasing black-to-white grayscale
hot	sequential black-red-yellow-white, to emulate blackbody radiation from an object at increasing temperatures
jet	a spectral map with dark endpoints, blue-cyan-yellow-red; based on a fluid-jet simulation by NCSA[1]
pink	sequential increasing pastel black-pink-white, meant for sepia tone colorization of photographs
prism	repetitive red-yellow-green-blue-purple-...-green pattern (not cyclic at endpoints)
spring	linearly-increasing shades of magenta-yellow
summer	sequential linearly-increasing shades of green-yellow
winter	linearly-increasing shades of blue-green

A set of palettes from the Yorick scientific visualisation package, an evolution of the GIST package, both by David H. Munro are included:

[1] Rainbow colormaps, `jet` in particular, are considered a poor choice for scientific visualization by many researchers: Rainbow Color Map (Still) Considered Harmful

Colormap	Description
gist_earth	mapmaker's colors from dark blue deep ocean to green lowlands to brown highlands to white mountains
gist_heat	sequential increasing black-red-orange-white, to emulate blackbody radiation from an iron bar as it grows hotter
gist_ncar	pseudo-spectral black-blue-green-yellow-red-purple-white colormap from National Center for Atmospheric Research[2]
gist_rainbow	runs through the colors in spectral order from red to violet at full saturation (like *hsv* but not cyclic)
gist_stern	"Stern special" color table from Interactive Data Language software

A set of cyclic color maps:

Colormap	Description
hsv	red-yellow-green-cyan-blue-magenta-red, formed by changing the hue component in the HSV color space
twilight	perceptually uniform shades of white-blue-black-red-white
twilight_shifted	perceptually uniform shades of black-blue-white-red-black

Other miscellaneous schemes:

[2] Resembles "BkBlAqGrYeOrReViWh200" from NCAR Command Language. See Color Table Gallery

Colormap	Description
afmhot	sequential black-orange-yellow-white blackbody spectrum, commonly used in atomic force microscopy
brg	blue-red-green
bwr	diverging blue-white-red
coolwarm	diverging blue-gray-red, meant to avoid issues with 3D shading, color blindness, and ordering of colors[3]
CMRmap	"Default colormaps on color images often reproduce to confusing grayscale images. The proposed colormap maintains an aesthetically pleasing color image that automatically reproduces to a monotonic grayscale with discrete, quantifiable saturation levels."[4]
cubehelix	Unlike most other color schemes cubehelix was designed by D.A. Green to be monotonically increasing in terms of perceived brightness. Also, when printed on a black and white postscript printer, the scheme results in a greyscale with monotonically increasing brightness. This color scheme is named cubehelix because the (r, g, b) values produced can be visualised as a squashed helix around the diagonal in the (r, g, b) color cube.
gnuplot	gnuplot's traditional pm3d scheme (black-blue-red-yellow)
gnuplot2	sequential color printable as gray (black-blue-violet-yellow-white)
ocean	green-blue-white
rainbow	spectral purple-blue-green-yellow-orange-red colormap with diverging luminance
seismic	diverging blue-white-red
nipy_spectral	black-purple-blue-green-yellow-red-white spectrum, originally from the Neuroimaging in Python project
terrain	mapmaker's colors, blue-green-yellow-brown-white, originally from IGOR Pro
turbo	Spectral map (purple-blue-green-yellow-orange-red) with a bright center and darker endpoints. A smoother alternative to jet.

The following colormaps are redundant and may be removed in future versions. It's recommended to use the names in the descriptions instead, which produce identical output:

[3] See Diverging Color Maps for Scientific Visualization by Kenneth Moreland.
[4] See A Color Map for Effective Black-and-White Rendering of Color-Scale Images by Carey Rappaport

Colormap	Description
gist_gray	identical to *gray*
gist_yarg	identical to *gray_r*
binary	identical to *gray_r*

18.39 matplotlib.projections

class matplotlib.projections.**ProjectionRegistry**

Bases: object

A mapping of registered projection names to projection classes.

get_projection_class(*self*, *name*)

Get a projection class from its *name*.

get_projection_names(*self*)

Return the names of all projections currently registered.

register(*self*, **projections*)

Register a new set of projections.

matplotlib.projections.**get_projection_class**(*projection=None*)

Get a projection class from its name.

If *projection* is None, a standard rectilinear projection is returned.

matplotlib.projections.**get_projection_names**()

Return the names of all projections currently registered.

matplotlib.projections.**register_projection**(*cls*)

18.40 matplotlib.projections.polar

class matplotlib.projections.polar.**InvertedPolarTransform**(*axis=None*, *use_rmin=True*, *_apply_theta_transforms=True*)

Bases: *matplotlib.transforms.Transform*

The inverse of the polar transform, mapping Cartesian coordinate space *x* and *y* back to *theta* and *r*.

Parameters

shorthand_name

[str] A string representing the "name" of the transform. The name carries no significance other than to improve the readability of `str(transform)` when DEBUG=True.

has_inverse = True

input_dims = 2

inverted(*self*)
Return the corresponding inverse transformation.

It holds `x == self.inverted().transform(self.transform(x))`.

The return value of this method should be treated as temporary. An update to *self* does not cause a corresponding update to its inverted copy.

output_dims = 2

transform_non_affine(*self*, *xy*)
Apply only the non-affine part of this transformation.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally equivalent to `transform(values)`. In affine transformations, this is always a no-op.

Parameters

values

[array] The input values as NumPy array of length *input_dims* or shape (N x *input_dims*).

Returns

array

The output values as NumPy array of length *input_dims* or shape (N x *output_dims*), depending on the input.

class matplotlib.projections.polar.PolarAffine(*scale_transform*, *limits*)
Bases: *matplotlib.transforms.Affine2DBase*

The affine part of the polar projection. Scales the output so that maximum radius rests on the edge of the axes circle.

limits is the view limit of the data. The only part of its bounds that is used is the y limits (for the radius limits). The theta range is handled by the non-affine transform.

get_matrix(*self*)
Get the matrix for the affine part of this transform.

class matplotlib.projections.polar.PolarAxes(**args*, *theta_offset=0*, *theta_direction=1*, *rlabel_position=22.5*, ***kwargs*)
Bases: matplotlib.axes._axes.Axes

A polar graph projection, where the input dimensions are *theta*, *r*.

Theta starts pointing east and goes anti-clockwise.

Build an axes in a figure.

Parameters

fig

[*Figure*] The axes is build in the *Figure* *fig*.

rect

[[left, bottom, width, height]] The axes is build in the rectangle *rect*. *rect* is in *Figure* coordinates.

sharex, sharey

[*Axes*, optional] The x or y *axis* is shared with the x or y axis in the input *Axes*.

frameon

[bool, default: True] Whether the axes frame is visible.

box_aspect

[None, or a number, optional] Sets the aspect of the axes box. See *set_box_aspect* for details.

****kwargs**

Other optional keyword arguments:

Property	Description
adjustable	{'box', 'datalim'}
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and r
alpha	float or None
anchor	2-tuple of floats or {'C', 'SW', 'S', 'SE', ...}
animated	bool
aspect	{'auto'} or num
autoscale_on	bool
autoscalex_on	bool
autoscaley_on	bool
axes_locator	Callable[[Axes, Renderer], Bbox]
axisbelow	bool or 'line'
box_aspect	None, or a number
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
contains	unknown
facecolor or fc	color
figure	*Figure*

Table 212 – continued from previous page

Property	Description
frame_on	bool
gid	str
in_layout	bool
label	object
navigate	bool
navigate_mode	unknown
path_effects	*AbstractPathEffect*
picker	None or bool or callable
position	[left, bottom, width, height] or *Bbox*
prop_cycle	unknown
rasterization_zorder	float or None
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
title	str
transform	*Transform*
url	str
visible	bool
xbound	unknown
xlabel	str
xlim	(bottom: float, top: float)
xmargin	float greater than -0.5
xscale	{"linear", "log", "symlog", "logit", ...}
xticklabels	unknown
xticks	unknown
ybound	unknown
ylabel	str
ylim	(bottom: float, top: float)
ymargin	float greater than -0.5
yscale	{"linear", "log", "symlog", "logit", ...}
yticklabels	unknown
yticks	unknown
zorder	float

Returns

Axes

The new *Axes* object.

class InvertedPolarTransform(*axis=None*, *use_rmin=True*, *_apply_theta_transforms=True*)

Bases: *matplotlib.transforms.Transform*

The inverse of the polar transform, mapping Cartesian coordinate space *x*

and *y* back to *theta* and *r*.

Parameters

shorthand_name

[str] A string representing the "name" of the transform. The name carries no significance other than to improve the readability of `str(transform)` when DEBUG=True.

`has_inverse = True`

`input_dims = 2`

`inverted`(*self*)

Return the corresponding inverse transformation.

It holds `x == self.inverted().transform(self.transform(x))`.

The return value of this method should be treated as temporary. An update to *self* does not cause a corresponding update to its inverted copy.

`output_dims = 2`

`transform_non_affine`(*self*, *xy*)

Apply only the non-affine part of this transformation.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally equivalent to `transform(values)`. In affine transformations, this is always a no-op.

Parameters

values

[array] The input values as NumPy array of length *input_dims* or shape (N x *input_dims*).

Returns

array

The output values as NumPy array of length *input_dims* or shape (N x *output_dims*), depending on the input.

`class PolarAffine`(*scale_transform*, *limits*)

Bases: *matplotlib.transforms.Affine2DBase*

The affine part of the polar projection. Scales the output so that maximum radius rests on the edge of the axes circle.

limits is the view limit of the data. The only part of its bounds that is used is the y limits (for the radius limits). The theta range is handled by the non-affine transform.

`get_matrix`(*self*)
Get the matrix for the affine part of this transform.

`class PolarTransform`(*axis=None*, *use_rmin=True*, *_apply_theta_transforms=True*)
Bases: *matplotlib.transforms.Transform*

The base polar transform. This handles projection *theta* and *r* into Cartesian coordinate space *x* and *y*, but does not perform the ultimate affine transformation into the correct position.

Parameters

shorthand_name

[str] A string representing the "name" of the transform. The name carries no significance other than to improve the readability of `str(transform)` when DEBUG=True.

`has_inverse = True`

`input_dims = 2`

`inverted`(*self*)
Return the corresponding inverse transformation.

It holds `x == self.inverted().transform(self.transform(x))`.

The return value of this method should be treated as temporary. An update to *self* does not cause a corresponding update to its inverted copy.

`output_dims = 2`

`transform_non_affine`(*self*, *tr*)
Apply only the non-affine part of this transformation.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally equivalent to `transform(values)`. In affine transformations, this is always a no-op.

Parameters

values

[array] The input values as NumPy array of length *input_dims* or shape (N x *input_dims*).

Returns

array

The output values as NumPy array of length *input_dims* or shape (N x *output_dims*), depending on the input.

transform_path_non_affine(*self*, *path*)
Apply the non-affine part of this transform to *Path* *path*, returning a new *Path*.

transform_path(path) is equivalent to transform_path_affine(transform_path_non_affine

class RadialLocator(*base*, *axes=None*)
Bases: *matplotlib.ticker.Locator*

Used to locate radius ticks.

Ensures that all ticks are strictly positive. For all other tasks, it delegates to the base *Locator* (which may be different depending on the scale of the *r*-axis.

autoscale(*self*)
[*Deprecated*]

Notes

Deprecated since version 3.2:

nonsingular(*self*, *vmin*, *vmax*)
Adjust a range as needed to avoid singularities.

This method gets called during autoscaling, with `(v0, v1)` set to the data limits on the axes if the axes contains any data, or `(-inf, +inf)` if not.

- If `v0 == v1` (possibly up to some floating point slop), this method returns an expanded interval around this value.
- If `(v0, v1) == (-inf, +inf)`, this method returns appropriate default view limits.
- Otherwise, `(v0, v1)` is returned without modification.

pan(*self*, *numsteps*)
[*Deprecated*]

Notes

Deprecated since version 3.3:

refresh(*self*)
[*Deprecated*]

Notes

Deprecated since version 3.3:

`view_limits`(*self*, *vmin*, *vmax*)
Select a scale for the range from vmin to vmax.

Subclasses should override this method to change locator behaviour.

`zoom`(*self*, *direction*)
[*Deprecated*]

Notes

Deprecated since version 3.3:

`class ThetaFormatter`
Bases: *matplotlib.ticker.Formatter*

Used to format the *theta* tick labels. Converts the native unit of radians into degrees and adds a degree symbol.

`class ThetaLocator`(*base*)
Bases: *matplotlib.ticker.Locator*

Used to locate theta ticks.

This will work the same as the base locator except in the case that the view spans the entire circle. In such cases, the previously used default locations of every 45 degrees are returned.

`autoscale`(*self*)
[*Deprecated*]

Notes

Deprecated since version 3.2:

`pan`(*self*, *numsteps*)
[*Deprecated*]

Notes

Deprecated since version 3.3:

`refresh`(*self*)
[*Deprecated*] Refresh internal information based on current limits.

Notes

Deprecated since version 3.3.

`set_axis`(*self*, *axis*)

`view_limits`(*self*, *vmin*, *vmax*)
Select a scale for the range from vmin to vmax.

Subclasses should override this method to change locator behaviour.

`zoom`(*self*, *direction*)
[*Deprecated*]

Notes

Deprecated since version 3.3:

`can_pan`(*self*)
Return *True* if this axes supports the pan/zoom button functionality.

For polar axes, this is slightly misleading. Both panning and zooming are performed by the same button. Panning is performed in azimuth while zooming is done along the radial.

`can_zoom`(*self*)
Return *True* if this axes supports the zoom box button functionality.

Polar axes do not support zoom boxes.

`cla`(*self*)
Clear the current axes.

`drag_pan`(*self*, *button*, *key*, *x*, *y*)
Called when the mouse moves during a pan operation.

Parameters

button

[*MouseButton*] The pressed mouse button.

key

[str or None] The pressed key, if any.

x, y

[float] The mouse coordinates in display coords.

Notes

This is intended to be overridden by new projection types.

`draw`(*self*, *renderer*, **args*, ***kwargs*)
Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

Parameters

renderer

[*RendererBase* subclass.]

Notes

This method is overridden in the Artist subclasses.

`end_pan`(*self*)
Called when a pan operation completes (when the mouse button is up.)

Notes

This is intended to be overridden by new projection types.

`format_coord`(*self*, *theta*, *r*)
Return a format string formatting the *x*, *y* coordinates.

`get_data_ratio`(*self*)
Return the aspect ratio of the data itself. For a polar plot, this should always be 1.0

`get_rlabel_position`(*self*)

Returns

float

The theta position of the radius labels in degrees.

`get_rmax`(*self*)

Returns

float

Outer radial limit.

`get_rmin`(*self*)

Returns

float

The inner radial limit.

`get_rorigin`(*self*)

Returns

float

`get_rsign`(*self*)

`get_theta_direction`(*self*)
Get the direction in which theta increases.

-1:

Theta increases in the clockwise direction

1:

Theta increases in the counterclockwise direction

`get_theta_offset`(*self*)
Get the offset for the location of 0 in radians.

`get_thetamax`(*self*)
Return the maximum theta limit in degrees.

`get_thetamin`(*self*)
Get the minimum theta limit in degrees.

`get_xaxis_text1_transform`(*self*, *pad*)

Returns

transform

[Transform] The transform used for drawing x-axis labels, which will add *pad_points* of padding (in points) between the axes and the label. The x-direction is in data coordinates and the y-direction is in axis corrdinates

valign

[{'center', 'top', 'bottom', 'baseline', 'center_baseline'}] The text vertical alignment.

halign

[{'center', 'left', 'right'}] The text horizontal alignment.

Notes

This transformation is primarily used by the *Axis* class, and is meant to be overridden by new kinds of projections that may need to place axis elements in different locations.

`get_xaxis_text2_transform`(*self*, *pad*)

Returns

transform

[Transform] The transform used for drawing secondary x-axis labels, which will add *pad_points* of padding (in points) between the axes and the label. The x-direction is in data coordinates and the y-direction is in axis corrdinates

valign

[{'center', 'top', 'bottom', 'baseline', 'center_baseline'}] The text vertical alignment.

halign

[{'center', 'left', 'right'}] The text horizontal alignment.

Notes

This transformation is primarily used by the *Axis* class, and is meant to be overridden by new kinds of projections that may need to place axis elements in different locations.

`get_xaxis_transform`(*self*, *which='grid'*)

Get the transformation used for drawing x-axis labels, ticks and gridlines. The x-direction is in data coordinates and the y-direction is in axis coordinates.

Note: This transformation is primarily used by the *Axis* class, and is meant to be overridden by new kinds of projections that may need to place axis elements in different locations.

`get_yaxis_text1_transform`(*self*, *pad*)

Returns

transform

[Transform] The transform used for drawing y-axis labels, which will add *pad_points* of padding (in points) between the

axes and the label. The x-direction is in axis coordinates and the y-direction is in data corrdinates

valign

[{'center', 'top', 'bottom', 'baseline', 'center_baseline'}] The text vertical alignment.

halign

[{'center', 'left', 'right'}] The text horizontal alignment.

Notes

This transformation is primarily used by the *Axis* class, and is meant to be overridden by new kinds of projections that may need to place axis elements in different locations.

`get_yaxis_text2_transform`(*self*, *pad*)

Returns

transform

[Transform] The transform used for drawing secondart y-axis labels, which will add *pad_points* of padding (in points) between the axes and the label. The x-direction is in axis coordinates and the y-direction is in data corrdinates

valign

[{'center', 'top', 'bottom', 'baseline', 'center_baseline'}] The text vertical alignment.

halign

[{'center', 'left', 'right'}] The text horizontal alignment.

Notes

This transformation is primarily used by the *Axis* class, and is meant to be overridden by new kinds of projections that may need to place axis elements in different locations.

`get_yaxis_transform`(*self*, *which='grid'*)

Get the transformation used for drawing y-axis labels, ticks and gridlines. The x-direction is in axis coordinates and the y-direction is in data coordinates.

Note: This transformation is primarily used by the *Axis* class, and is meant to be overridden by new kinds of projections that may need to place axis elements in different locations.

name = 'polar'

set_rgrids(*self*, *radii*, *labels=None*, *angle=None*, *fmt=None*, ***kwargs*)

Set the radial gridlines on a polar plot.

Parameters

radii

[tuple with floats] The radii for the radial gridlines

labels

[tuple with strings or None] The labels to use at each radial gridline. The *matplotlib.ticker.ScalarFormatter* will be used if None.

angle

[float] The angular position of the radius labels in degrees.

fmt

[str or None] Format string used in *matplotlib.ticker.FormatStrFormatter*. For example '%f'.

Returns

lines

[list of *lines.Line2D*] The radial gridlines.

labels

[list of *text.Text*] The tick labels.

Other Parameters

****kwargs**

kwargs are optional *Text* properties for the labels.

See also:

PolarAxes.set_thetagrids

Axis.get_gridlines

Axis.get_ticklabels

set_rlabel_position(*self*, *value*)
Update the theta position of the radius labels.

Parameters

value

[number] The angular position of the radius labels in degrees.

set_rlim(*self*, *bottom=None*, *top=None*, *emit=True*, *auto=False*, ***kwargs*)
See *set_ylim*.

set_rmax(*self*, *rmax*)
Set the outer radial limit.

Parameters

rmax

[float]

set_rmin(*self*, *rmin*)
Set the inner radial limit.

Parameters

rmin

[float]

set_rorigin(*self*, *rorigin*)
Update the radial origin.

Parameters

rorigin

[float]

set_rscale(*self*, **args*, ***kwargs*)

set_rticks(*self*, **args*, ***kwargs*)

set_theta_direction(*self*, *direction*)
Set the direction in which theta increases.

clockwise, -1:

Theta increases in the clockwise direction

counterclockwise, anticlockwise, 1:

Theta increases in the counterclockwise direction

set_theta_offset(*self*, *offset*)
Set the offset for the location of 0 in radians.

`set_theta_zero_location`(*self*, *loc*, *offset=0.0*)
Set the location of theta's zero.

This simply calls *set_theta_offset* with the correct value in radians.

Parameters

loc

[str] May be one of "N", "NW", "W", "SW", "S", "SE", "E", or "NE".

offset

[float, default: 0] An offset in degrees to apply from the specified *loc*. **Note:** this offset is *always* applied counter-clockwise regardless of the direction setting.

`set_thetagrids`(*self*, *angles*, *labels=None*, *fmt=None*, ***kwargs*)
Set the theta gridlines in a polar plot.

Parameters

angles

[tuple with floats, degrees] The angles of the theta gridlines.

labels

[tuple with strings or None] The labels to use at each theta gridline. The *projections.polar.ThetaFormatter* will be used if None.

fmt

[str or None] Format string used in *matplotlib.ticker.FormatStrFormatter*. For example '%f'. Note that the angle that is used is in radians.

Returns

lines

[list of *lines.Line2D*] The theta gridlines.

labels

[list of *text.Text*] The tick labels.

Other Parameters

****kwargs**

kwargs are optional *Text* properties for the labels.

See also:

PolarAxes.set_rgrids

Axis.get_gridlines

Axis.get_ticklabels

`set_thetalim`(*self*, **args*, ***kwargs*)
Set the minimum and maximum theta values.

Can take the following signatures:

- `set_thetalim(minval, maxval)`: Set the limits in radians.
- `set_thetalim(thetamin=minval, thetamax=maxval)`: Set the limits in degrees.

where minval and maxval are the minimum and maximum limits. Values are wrapped in to the range $[0, 2\pi]$ (in radians), so for example it is possible to do `set_thetalim(-np.pi / 2, np.pi / 2)` to have an axes symmetric around 0. A ValueError is raised if the absolute angle difference is larger than 2π.

`set_thetamax`(*self*, *thetamax*)
Set the maximum theta limit in degrees.

`set_thetamin`(*self*, *thetamin*)
Set the minimum theta limit in degrees.

`set_xscale`(*self*, *scale*, **args*, ***kwargs*)
Set the x-axis scale.

Parameters

value

[{"linear", "log", "symlog", "logit", ...}] The axis scale type to apply.

****kwargs**

Different keyword arguments are accepted, depending on the scale. See the respective class keyword arguments:

- *matplotlib.scale.LinearScale*
- *matplotlib.scale.LogScale*
- *matplotlib.scale.SymmetricalLogScale*
- *matplotlib.scale.LogitScale*

Notes

By default, Matplotlib supports the above mentioned scales. Additionally, custom scales may be registered using *matplotlib.scale.register_scale*. These scales can then also be used here.

`set_ylim`(*self*, *bottom=None*, *top=None*, *emit=True*, *auto=False*, *, *ymin=None*, *ymax=None*)

Set the data limits for the radial axis.

Parameters

bottom

[float, optional] The bottom limit (default: None, which leaves the bottom limit unchanged). The bottom and top ylims may be passed as the tuple (*bottom*, *top*) as the first positional argument (or as the *bottom* keyword argument).

top

[float, optional] The top limit (default: None, which leaves the top limit unchanged).

emit

[bool, default: True] Whether to notify observers of limit change.

auto

[bool or None, default: False] Whether to turn on autoscaling of the y-axis. True turns on, False turns off, None leaves unchanged.

ymin, ymax

[float, optional] These arguments are deprecated and will be removed in a future version. They are equivalent to *bottom* and *top* respectively, and it is an error to pass both *ymin* and *bottom* or *ymax* and *top*.

Returns

bottom, top

[(float, float)] The new y-axis limits in data coordinates.

`set_yscale`(*self*, **args*, ***kwargs*)

Set the y-axis scale.

Parameters

value

[{"linear", "log", "symlog", "logit", ...}] The axis scale type to apply.

****kwargs**

Different keyword arguments are accepted, depending on the scale. See the respective class keyword arguments:

- *matplotlib.scale.LinearScale*
- *matplotlib.scale.LogScale*
- *matplotlib.scale.SymmetricalLogScale*
- *matplotlib.scale.LogitScale*

Notes

By default, Matplotlib supports the above mentioned scales. Additionally, custom scales may be registered using *matplotlib.scale.register_scale*. These scales can then also be used here.

`start_pan`(*self*, *x*, *y*, *button*)
Called when a pan operation has started.

Parameters

x, y

[float] The mouse coordinates in display coords.

button

[*MouseButton*] The pressed mouse button.

Notes

This is intended to be overridden by new projection types.

`class matplotlib.projections.polar.PolarTransform`(*axis=None*, *use_rmin=True*, *_apply_theta_transforms=True*)

Bases: *matplotlib.transforms.Transform*

The base polar transform. This handles projection *theta* and *r* into Cartesian coordinate space *x* and *y*, but does not perform the ultimate affine transformation into the correct position.

Parameters

shorthand_name

[str] A string representing the "name" of the transform. The name carries no significance other than to improve the readability of `str(transform)` when DEBUG=True.

`has_inverse = True`

`input_dims = 2`

`inverted`(*self*)
Return the corresponding inverse transformation.

It holds `x == self.inverted().transform(self.transform(x))`.

The return value of this method should be treated as temporary. An update to *self* does not cause a corresponding update to its inverted copy.

`output_dims = 2`

`transform_non_affine`(*self*, *tr*)
Apply only the non-affine part of this transformation.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally equivalent to `transform(values)`. In affine transformations, this is always a no-op.

Parameters

values

[array] The input values as NumPy array of length *input_dims* or shape (N x *input_dims*).

Returns

array

The output values as NumPy array of length *input_dims* or shape (N x *output_dims*), depending on the input.

`transform_path_non_affine`(*self*, *path*)
Apply the non-affine part of this transform to *Path* *path*, returning a new *Path*.

`transform_path(path)` is equivalent to `transform_path_affine(transform_path_non_affine(va`

`class matplotlib.projections.polar.RadialAxis`(**args*, ***kwargs*)
Bases: *matplotlib.axis.YAxis*

A radial Axis.

This overrides certain properties of a *YAxis* to provide special-casing for a radial axis.

Parameters

axes

[*matplotlib.axes.Axes*] The *Axes* to which the created Axis belongs.

pickradius

[float] The acceptance radius for containment tests. See also *Axis.contains*.

`axis_name = 'radius'`
Read-only name identifying the axis.

`cla`(*self*)
Clear this axis.

`class matplotlib.projections.polar.RadialLocator`(*base*, *axes=None*)
Bases: *matplotlib.ticker.Locator*

Used to locate radius ticks.

Ensures that all ticks are strictly positive. For all other tasks, it delegates to the base *Locator* (which may be different depending on the scale of the *r*-axis.

`autoscale`(*self*)
[*Deprecated*]

Notes

Deprecated since version 3.2:

`nonsingular`(*self*, *vmin*, *vmax*)
Adjust a range as needed to avoid singularities.

This method gets called during autoscaling, with `(v0, v1)` set to the data limits on the axes if the axes contains any data, or `(-inf, +inf)` if not.

- If `v0 == v1` (possibly up to some floating point slop), this method returns an expanded interval around this value.
- If `(v0, v1) == (-inf, +inf)`, this method returns appropriate default view limits.
- Otherwise, `(v0, v1)` is returned without modification.

`pan`(*self*, *numsteps*)
[*Deprecated*]

Notes

Deprecated since version 3.3:

`refresh`(*self*)
[*Deprecated*]

Notes

Deprecated since version 3.3:

`view_limits`(*self*, *vmin*, *vmax*)
Select a scale for the range from vmin to vmax.

Subclasses should override this method to change locator behaviour.

`zoom`(*self*, *direction*)
[*Deprecated*]

Notes

Deprecated since version 3.3:

`class matplotlib.projections.polar.RadialTick`(**args*, ***kwargs*)
Bases: *matplotlib.axis.YTick*

A radial-axis tick.

This subclass of *YTick* provides radial ticks with some small modification to their re-positioning such that ticks are rotated based on axes limits. This results in ticks that are correctly perpendicular to the spine. Labels are also rotated to be perpendicular to the spine, when 'auto' rotation is enabled.

bbox is the Bound2D bounding box in display coords of the Axes loc is the tick location in data coords size is the tick size in points

`update_position`(*self*, *loc*)
Set the location of tick in data coords with scalar *loc*.

`class matplotlib.projections.polar.ThetaAxis`(**args*, ***kwargs*)
Bases: *matplotlib.axis.XAxis*

A theta Axis.

This overrides certain properties of an *XAxis* to provide special-casing for an angular axis.

Parameters

axes

[*matplotlib.axes.Axes*] The *Axes* to which the created Axis belongs.

pickradius

[float] The acceptance radius for containment tests. See also *Axis.contains*.

`axis_name = 'theta'`
Read-only name identifying the axis.

`cla`(*self*)
Clear this axis.

`class matplotlib.projections.polar.ThetaFormatter`
Bases: *matplotlib.ticker.Formatter*

Used to format the *theta* tick labels. Converts the native unit of radians into degrees and adds a degree symbol.

`class matplotlib.projections.polar.ThetaLocator`(*base*)
Bases: *matplotlib.ticker.Locator*

Used to locate theta ticks.

This will work the same as the base locator except in the case that the view spans the entire circle. In such cases, the previously used default locations of every 45 degrees are returned.

`autoscale`(*self*)
[*Deprecated*]

Notes

Deprecated since version 3.2:

`pan`(*self*, *numsteps*)
[*Deprecated*]

Notes

Deprecated since version 3.3:

`refresh`(*self*)
[*Deprecated*] Refresh internal information based on current limits.

Notes

Deprecated since version 3.3.

`set_axis`(*self*, *axis*)

`view_limits`(*self*, *vmin*, *vmax*)
Select a scale for the range from vmin to vmax.

Subclasses should override this method to change locator behaviour.

`zoom`(*self*, *direction*)
[*Deprecated*]

Notes

Deprecated since version 3.3:

`class matplotlib.projections.polar.ThetaTick`(*axes*, **args*, ***kwargs*)
Bases: *matplotlib.axis.XTick*

A theta-axis tick.

This subclass of *XTick* provides angular ticks with some small modification to their re-positioning such that ticks are rotated based on tick location. This results in ticks that are correctly perpendicular to the arc spine.

When 'auto' rotation is enabled, labels are also rotated to be parallel to the spine. The label padding is also applied here since it's not possible to use a generic axes transform to produce tick-specific padding.

bbox is the Bound2D bounding box in display coords of the Axes loc is the tick location in data coords size is the tick size in points

`update_position`(*self*, *loc*)
Set the location of tick in data coords with scalar *loc*.

18.41 matplotlib.quiver

Support for plotting vector fields.

Presently this contains Quiver and Barb. Quiver plots an arrow in the direction of the vector, with the size of the arrow related to the magnitude of the vector.

Barbs are like quiver in that they point along a vector, but the magnitude of the vector is given schematically by the presence of barbs or flags on the barb.

This will also become a home for things such as standard deviation ellipses, which can and will be derived very easily from the Quiver code.

18.41.1 Classes

Quiver(ax, *args[, scale, headwidth, ...])	Specialized PolyCollection for arrows.
QuiverKey(Q, X, Y, U, label, *[, angle, ...])	Labelled arrow for use as a quiver plot scale key.
Barbs(ax, *args[, pivot, length, barbcolor, ...])	Specialized PolyCollection for barbs.

matplotlib.quiver.Quiver

class matplotlib.quiver.**Quiver**(*ax*, **args*, *scale=None*, *headwidth=3*, *headlength=5*, *headaxislength=4.5*, *minshaft=1*, *minlength=1*, *units='width'*, *scale_units=None*, *angles='uv'*, *width=None*, *color='k'*, *pivot='tail'*, ***kw*)

Bases: *matplotlib.collections.PolyCollection*

Specialized PolyCollection for arrows.

The only API method is set_UVC(), which can be used to change the size, orientation, and color of the arrows; their locations are fixed when the class is instantiated. Possibly this method will be useful in animations.

Much of the work in this class is done in the draw() method so that as much information as possible is available about the plot. In subsequent draw() calls, recalculation is limited to things that might have changed, so there should be no performance penalty from putting the calculations in the draw() method.

Plot a 2D field of arrows.

Call signature:

```
quiver([X, Y], U, V, [C], **kw)
```

X, *Y* define the arrow locations, *U*, *V* define the arrow directions, and *C* optionally sets the color.

Arrow size

The default settings auto-scales the length of the arrows to a reasonable size. To change this behavior see the *scale* and *scale_units* parameters.

Arrow shape

The defaults give a slightly swept-back arrow; to make the head a triangle, make *headaxislength* the same as *headlength*. To make the arrow more pointed, reduce *headwidth* or increase *headlength* and *headaxislength*. To make the head smaller relative to the shaft, scale down all the head parameters. You will probably do best to leave minshaft alone.

Arrow outline

linewidths and *edgecolors* can be used to customize the arrow outlines.

Parameters

X, Y

[1D or 2D array-like, optional] The x and y coordinates of the arrow locations.

If not given, they will be generated as a uniform integer meshgrid based on the dimensions of *U* and *V*.

If *X* and *Y* are 1D but *U*, *V* are 2D, *X*, *Y* are expanded to 2D using `X, Y = np.meshgrid(X, Y)`. In this case `len(X)` and `len(Y)` must match the column and row dimensions of *U* and *V*.

U, V

[1D or 2D array-like] The x and y direction components of the arrow vectors.

They must have the same number of elements, matching the number of arrow locations. *U* and *V* may be masked. Only locations unmasked in *U*, *V*, and *C* will be drawn.

C

[1D or 2D array-like, optional] Numeric data that defines the arrow colors by colormapping via *norm* and *cmap*.

This does not support explicit colors. If you want to set colors directly, use *color* instead. The size of *C* must match the number of arrow locations.

units

[{'width', 'height', 'dots', 'inches', 'x', 'y' 'xy'}, default: 'width'] The arrow dimensions (except for *length*) are measured in multiples of this unit.

The following values are supported:

- 'width', 'height': The width or height of the axis.
- 'dots', 'inches': Pixels or inches based on the figure dpi.
- 'x', 'y', 'xy': *X*, *Y* or $\sqrt{X^2 + Y^2}$ in data units.

The arrows scale differently depending on the units. For 'x' or 'y', the arrows get larger as one zooms in; for other units, the arrow size is independent of the zoom state. For 'width or 'height', the arrow size increases with the width and height of the axes, respectively, when the window is resized; for 'dots' or 'inches', resizing does not change the arrows.

angles

[{'uv', 'xy'} or array-like, default: 'uv'] Method for determining the angle of the arrows.

- 'uv': The arrow axis aspect ratio is 1 so that if $U == V$ the orientation of the arrow on the plot is 45 degrees counter-clockwise from the horizontal axis (positive to the right).

  Use this if the arrows symbolize a quantity that is not based on X, Y data coordinates.

- 'xy': Arrows point from (x, y) to (x+u, y+v). Use this for plotting a gradient field, for example.

- Alternatively, arbitrary angles may be specified explicitly as an array of values in degrees, counter-clockwise from the horizontal axis.

  In this case U, V is only used to determine the length of the arrows.

Note: inverting a data axis will correspondingly invert the arrows only with `angles='xy'`.

scale

[float, optional] Number of data units per arrow length unit, e.g., m/s per plot width; a smaller scale parameter makes the arrow longer. Default is *None*.

If *None*, a simple autoscaling algorithm is used, based on the average vector length and the number of vectors. The arrow length unit is given by the *scale_units* parameter.

scale_units

[{'width', 'height', 'dots', 'inches', 'x', 'y', 'xy'}, optional] If the *scale* kwarg is *None*, the arrow length unit. Default is *None*.

e.g. *scale_units* is 'inches', *scale* is 2.0, and `(u, v) = (1, 0)`, then the vector will be 0.5 inches long.

If *scale_units* is 'width' or 'height', then the vector will be half the width/height of the axes.

If *scale_units* is 'x' then the vector will be 0.5 x-axis units. To plot vectors in the x-y plane, with u and v having the same units as x and y, use `angles='xy', scale_units='xy', scale=1`.

width

[float, optional] Shaft width in arrow units; default depends on choice of units, above, and number of vectors; a typical starting value is about 0.005 times the width of the plot.

headwidth

[float, default: 3] Head width as multiple of shaft width.

headlength

[float, default: 5] Head length as multiple of shaft width.

headaxislength

[float, default: 4.5] Head length at shaft intersection.

minshaft

[float, default: 1] Length below which arrow scales, in units of head length. Do not set this to less than 1, or small arrows will look terrible!

minlength

[float, default: 1] Minimum length as a multiple of shaft width; if an arrow length is less than this, plot a dot (hexagon) of this diameter instead.

pivot

[{'tail', 'mid', 'middle', 'tip'}, default: 'tail'] The part of the arrow that is anchored to the *X*, *Y* grid. The arrow rotates about this point.

'mid' is a synonym for 'middle'.

color

[color or color sequence, optional] Explicit color(s) for the arrows. If *C* has been set, *color* has no effect.

This is a synonym for the *PolyCollection* *facecolor* parameter.

Other Parameters

****kwargs**

[*PolyCollection* properties, optional] All other keyword arguments are passed on to *PolyCollection*:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a
alpha	float or None
animated	bool
antialiased or aa or antialiaseds	bool or list of bools
array	ndarray
capstyle	{'butt', 'round', 'projecting'}
clim	(vmin: float, vmax: float)
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None

Table 214 – continued from previous page

Property	Description
cmap	*Colormap* or str or None
color	color or list of rgba tuples
contains	unknown
edgecolor or ec or edgecolors	color or list of colors or 'face'
facecolor or facecolors or fc	color or list of colors
figure	*Figure*
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or dashes or linestyles or ls	str or tuple or list thereof
linewidth or linewidths or lw	float or list of floats
norm	*Normalize* or None
offset_position	unknown
offsets	array-like (N, 2) or (2,)
path_effects	*AbstractPathEffect*
picker	None or bool or callable
pickradius	unknown
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
urls	list of str or None
visible	bool
zorder	float

See also:

Axes.quiverkey

Add a key to a quiver plot.

`__init__`(*self*, *ax*, **args*, *scale=None*, *headwidth=3*, *headlength=5*, *headaxislength=4.5*, *minshaft=1*, *minlength=1*, *units='width'*, *scale_units=None*, *angles='uv'*, *width=None*, *color='k'*, *pivot='tail'*, ***kw*)

The constructor takes one required argument, an Axes instance, followed by the args and kwargs described by the following pyplot interface documentation:

Plot a 2D field of arrows.

Call signature:

```
quiver([X, Y], U, V, [C], **kw)
```

X, *Y* define the arrow locations, *U*, *V* define the arrow directions, and *C* optionally sets the color.

Arrow size

The default settings auto-scales the length of the arrows to a reasonable size. To change this behavior see the *scale* and *scale_units* parameters.

Arrow shape

The defaults give a slightly swept-back arrow; to make the head a triangle, make *headaxislength* the same as *headlength*. To make the arrow more pointed, reduce *headwidth* or increase *headlength* and *headaxislength*. To make the head smaller relative to the shaft, scale down all the head parameters. You will probably do best to leave minshaft alone.

Arrow outline

linewidths and *edgecolors* can be used to customize the arrow outlines.

Parameters

X, Y

[1D or 2D array-like, optional] The x and y coordinates of the arrow locations.

If not given, they will be generated as a uniform integer meshgrid based on the dimensions of *U* and *V*.

If *X* and *Y* are 1D but *U*, *V* are 2D, *X*, *Y* are expanded to 2D using `X, Y = np.meshgrid(X, Y)`. In this case `len(X)` and `len(Y)` must match the column and row dimensions of *U* and *V*.

U, V

[1D or 2D array-like] The x and y direction components of the arrow vectors.

They must have the same number of elements, matching the number of arrow locations. *U* and *V* may be masked. Only locations unmasked in *U*, *V*, and *C* will be drawn.

C

[1D or 2D array-like, optional] Numeric data that defines the arrow colors by colormapping via *norm* and *cmap*.

This does not support explicit colors. If you want to set colors directly, use *color* instead. The size of *C* must match the number of arrow locations.

units

[{'width', 'height', 'dots', 'inches', 'x', 'y' 'xy'}, default: 'width'] The arrow dimensions (except for *length*) are measured in multiples of this unit.

The following values are supported:

- 'width', 'height': The width or height of the axis.
- 'dots', 'inches': Pixels or inches based on the figure dpi.
- 'x', 'y', 'xy': X, Y or $\sqrt{X^2 + Y^2}$ in data units.

The arrows scale differently depending on the units. For 'x' or 'y', the arrows get larger as one zooms in; for other units, the arrow size is independent of the zoom state. For 'width or 'height', the arrow size increases with the width and height of the axes, respectively, when the window is resized; for 'dots' or 'inches', resizing does not change the arrows.

angles

[{'uv', 'xy'} or array-like, default: 'uv'] Method for determining the angle of the arrows.

- 'uv': The arrow axis aspect ratio is 1 so that if $U == V$ the orientation of the arrow on the plot is 45 degrees counter-clockwise from the horizontal axis (positive to the right).

  Use this if the arrows symbolize a quantity that is not based on X, Y data coordinates.

- 'xy': Arrows point from (x, y) to (x+u, y+v). Use this for plotting a gradient field, for example.

- Alternatively, arbitrary angles may be specified explicitly as an array of values in degrees, counter-clockwise from the horizontal axis.

  In this case U, V is only used to determine the length of the arrows.

Note: inverting a data axis will correspondingly invert the arrows only with `angles='xy'`.

scale

[float, optional] Number of data units per arrow length unit, e.g., m/s per plot width; a smaller scale parameter makes the arrow longer. Default is *None*.

If *None*, a simple autoscaling algorithm is used, based on the average vector length and the number of vectors. The arrow length unit is given by the *scale_units* parameter.

scale_units

[{'width', 'height', 'dots', 'inches', 'x', 'y', 'xy'}, optional] If the *scale* kwarg is *None*, the arrow length unit. Default is *None*.

e.g. *scale_units* is 'inches', *scale* is 2.0, and `(u, v) = (1, 0)`, then the vector will be 0.5 inches long.

If *scale_units* is 'width' or 'height', then the vector will be half the width/height of the axes.

If *scale_units* is 'x' then the vector will be 0.5 x-axis units. To plot vectors in the x-y plane, with u and v having the same units as x and y, use `angles='xy', scale_units='xy', scale=1`.

width

[float, optional] Shaft width in arrow units; default depends on choice of units, above, and number of vectors; a typical starting value is about 0.005 times the width of the plot.

headwidth

[float, default: 3] Head width as multiple of shaft width.

headlength

[float, default: 5] Head length as multiple of shaft width.

headaxislength

[float, default: 4.5] Head length at shaft intersection.

minshaft

[float, default: 1] Length below which arrow scales, in units of head length. Do not set this to less than 1, or small arrows will look terrible!

minlength

[float, default: 1] Minimum length as a multiple of shaft width; if an arrow length is less than this, plot a dot (hexagon) of this diameter instead.

pivot

[{'tail', 'mid', 'middle', 'tip'}, default: 'tail'] The part of the arrow that is anchored to the *X*, *Y* grid. The arrow rotates about this point.

'mid' is a synonym for 'middle'.

color

[color or color sequence, optional] Explicit color(s) for the arrows. If *C* has been set, *color* has no effect.

This is a synonym for the *PolyCollection* *facecolor* parameter.

Other Parameters

**kwargs

[*PolyCollection* properties, optional] All other keyword arguments are passed on to *PolyCollection*:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a
alpha	float or None
animated	bool
antialiased or aa or antialiaseds	bool or list of bools
array	ndarray
capstyle	{'butt', 'round', 'projecting'}
clim	(vmin: float, vmax: float)
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
cmap	*Colormap* or str or None
color	color or list of rgba tuples
contains	unknown
edgecolor or ec or edgecolors	color or list of colors or 'face'
facecolor or facecolors or fc	color or list of colors
figure	*Figure*
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or dashes or linestyles or ls	str or tuple or list thereof
linewidth or linewidths or lw	float or list of floats
norm	*Normalize* or None
offset_position	unknown
offsets	array-like (N, 2) or (2,)
path_effects	*AbstractPathEffect*
picker	None or bool or callable
pickradius	unknown
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
urls	list of str or None
visible	bool
zorder	float

See also:

Axes.quiverkey

Add a key to a quiver plot.

__module__ = 'matplotlib.quiver'

ax(*self*)

[*Deprecated*]

Notes

Deprecated since version 3.3:

draw(*self*, *renderer*)

Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

Parameters

renderer

[*RendererBase* subclass.]

Notes

This method is overridden in the Artist subclasses.

get_datalim(*self*, *transData*)

quiver_doc = "\nPlot a 2D field of arrows.\n\nCall signature::\n\n quiver([X, Y], U, V, [C

remove(*self*)

Remove the artist from the figure if possible.

The effect will not be visible until the figure is redrawn, e.g., with *FigureCanvasBase.draw_idle*. Call *relim* to update the axes limits if desired.

Note: *relim* will not see collections even if the collection was added to the axes with *autolim* = True.

Note: there is no support for removing the artist's legend entry.

set_UVC(*self*, *U*, *V*, *C=None*)

Examples using `matplotlib.quiver.Quiver`

- sphx_glr_gallery_images_contours_and_fields_quiver_demo.py
- sphx_glr_gallery_images_contours_and_fields_quiver_simple_demo.py

matplotlib.quiver.QuiverKey

`class matplotlib.quiver.QuiverKey(`*Q*, *X*, *Y*, *U*, *label*, ***, *angle=0*, *coordinates='axes'*, *color=None*, *labelsep=0.1*, *labelpos='N'*, *labelcolor=None*, *fontproperties=None*, ***kw*`)`

Bases: *matplotlib.artist.Artist*

Labelled arrow for use as a quiver plot scale key.

Add a key to a quiver plot.

The positioning of the key depends on *X*, *Y*, *coordinates*, and *labelpos*. If *labelpos* is 'N' or 'S', *X*, *Y* give the position of the middle of the key arrow. If *labelpos* is 'E', *X*, *Y* positions the head, and if *labelpos* is 'W', *X*, *Y* positions the tail; in either of these two cases, *X*, *Y* is somewhere in the middle of the arrow+label key object.

Parameters

Q

[*matplotlib.quiver.Quiver*] A *Quiver* object as returned by a call to *quiver()*.

X, Y

[float] The location of the key.

U

[float] The length of the key.

label

[str] The key label (e.g., length and units of the key).

angle

[float, default: 0] The angle of the key arrow, in degrees anticlockwise from the x-axis.

coordinates

[{'axes', 'figure', 'data', 'inches'}, default: 'axes'] Coordinate system and units for *X*, *Y*: 'axes' and 'figure' are normalized coordinate systems with (0, 0) in the lower left and (1, 1) in the upper right; 'data' are the axes data coordinates (used for the locations of the vectors in the quiver plot itself); 'inches' is position in the figure in inches, with (0, 0) at the lower left corner.

color

[color] Overrides face and edge colors from *Q*.

labelpos

[{'N', 'S', 'E', 'W'}] Position the label above, below, to the right, to the left of the arrow, respectively.

labelsep

[float, default: 0.1] Distance in inches between the arrow and the label.

labelcolor

[color, default: `rcParams["text.color"]` (default: `'black'`)] Label color.

fontproperties

[dict, optional] A dictionary with keyword arguments accepted by the *FontProperties* initializer: *family*, *style*, *variant*, *size*, *weight*.

****kwargs**

Any additional keyword arguments are used to override vector properties taken from *Q*.

`__init__`(*self*, *Q*, *X*, *Y*, *U*, *label*, *, *angle=0*, *coordinates='axes'*, *color=None*, *labelsep=0.1*, *labelpos='N'*, *labelcolor=None*, *fontproperties=None*, ***kw*)

Add a key to a quiver plot.

The positioning of the key depends on *X*, *Y*, *coordinates*, and *labelpos*. If *labelpos* is 'N' or 'S', *X*, *Y* give the position of the middle of the key arrow. If *labelpos* is 'E', *X*, *Y* positions the head, and if *labelpos* is 'W', *X*, *Y* positions the tail; in either of these two cases, *X*, *Y* is somewhere in the middle of the arrow+label key object.

Parameters

Q

[*matplotlib.quiver.Quiver*] A *Quiver* object as returned by a call to *quiver()*.

X, Y

[float] The location of the key.

U

[float] The length of the key.

label

[str] The key label (e.g., length and units of the key).

angle

[float, default: 0] The angle of the key arrow, in degrees anti-clockwise from the x-axis.

coordinates

[{'axes', 'figure', 'data', 'inches'}, default: 'axes'] Coordinate system and units for *X*, *Y*: 'axes' and 'figure' are normalized coordinate systems with (0, 0) in the lower left and (1, 1) in the upper right; 'data' are the axes data coordinates (used for the locations of the vectors in the quiver plot itself); 'inches' is position in the figure in inches, with (0, 0) at the lower left corner.

color

[color] Overrides face and edge colors from *Q*.

labelpos

[{'N', 'S', 'E', 'W'}] Position the label above, below, to the right, to the left of the arrow, respectively.

labelsep

[float, default: 0.1] Distance in inches between the arrow and the label.

labelcolor

[color, default: `rcParams["text.color"]` (default: `'black'`)] Label color.

fontproperties

[dict, optional] A dictionary with keyword arguments accepted by the *FontProperties* initializer: *family*, *style*, *variant*, *size*, *weight*.

****kwargs**

Any additional keyword arguments are used to override vector properties taken from *Q*.

`__module__ = 'matplotlib.quiver'`

`contains`(*self*, *mouseevent*)

Test whether the artist contains the mouse event.

Parameters

mouseevent

[*matplotlib.backend_bases.MouseEvent*]

Returns

contains

[bool] Whether any values are within the radius.

details

[dict] An artist-specific dictionary of details of the event context, such as which points are contained in the pick radius. See the individual Artist subclasses for details.

`draw`(*self*, *renderer*)
Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

Parameters

renderer

[*RendererBase* subclass.]

Notes

This method is overridden in the Artist subclasses.

`halign = {'E': 'left', 'N': 'center', 'S': 'center', 'W': 'right'}`

`pivot = {'E': 'tip', 'N': 'middle', 'S': 'middle', 'W': 'tail'}`

`property quiverkey_doc`

`remove`(*self*)
Remove the artist from the figure if possible.

The effect will not be visible until the figure is redrawn, e.g., with *FigureCanvasBase.draw_idle*. Call *relim* to update the axes limits if desired.

Note: *relim* will not see collections even if the collection was added to the axes with *autolim* = True.

Note: there is no support for removing the artist's legend entry.

`set_figure`(*self*, *fig*)
Set the *Figure* instance the artist belongs to.

Parameters

fig

[*Figure*]

```
valign = {'E': 'center', 'N': 'bottom', 'S': 'top', 'W': 'center'}
```

Examples using `matplotlib.quiver.QuiverKey`

- sphx_glr_gallery_images_contours_and_fields_quiver_demo.py

matplotlib.quiver.Barbs

class matplotlib.quiver.**Barbs**(*ax*, **args*, *pivot='tip'*, *length=7*, *barbcolor=None*, *flagcolor=None*, *sizes=None*, *fill_empty=False*, *barb_increments=None*, *rounding=True*, *flip_barb=False*, ***kw*)

Bases: *matplotlib.collections.PolyCollection*

Specialized PolyCollection for barbs.

The only API method is *set_UVC()*, which can be used to change the size, orientation, and color of the arrows. Locations are changed using the *set_offsets()* collection method. Possibly this method will be useful in animations.

There is one internal function `_find_tails()` which finds exactly what should be put on the barb given the vector magnitude. From there `_make_barbs()` is used to find the vertices of the polygon to represent the barb based on this information.

Plot a 2D field of barbs.

Call signature:

```
barbs([X, Y], U, V, [C], **kw)
```

Where X, Y define the barb locations, U, V define the barb directions, and C optionally sets the color.

All arguments may be 1D or 2D. U, V, C may be masked arrays, but masked X, Y are not supported at present.

Barbs are traditionally used in meteorology as a way to plot the speed and direction of wind observations, but can technically be used to plot any two dimensional vector quantity. As opposed to arrows, which give vector magnitude by the length of the arrow, the barbs give more quantitative information about the vector magnitude by putting slanted lines or a triangle for various increments in magnitude, as show schematically below:

```
:                  /\    \
:                 /  \    \
:                /    \    \    \
:               /      \    \    \
:              ------------------------------
```

The largest increment is given by a triangle (or "flag"). After those come full lines (barbs). The smallest increment is a half line. There is only, of course, ever at most 1 half line. If the magnitude is small and only needs a single half-line and no full lines or triangles, the half-line is offset from the end of the barb so that it can be easily distinguished from barbs with a single full line. The magnitude for the barb shown above would nominally be 65, using the standard increments of 50, 10, and 5.

See also https://en.wikipedia.org/wiki/Wind_barb.

Parameters

X, Y

[1D or 2D array-like, optional] The x and y coordinates of the barb locations. See *pivot* for how the barbs are drawn to the x, y positions.

If not given, they will be generated as a uniform integer meshgrid based on the dimensions of *U* and *V*.

If *X* and *Y* are 1D but *U*, *V* are 2D, *X*, *Y* are expanded to 2D using `X, Y = np.meshgrid(X, Y)`. In this case `len(X)` and `len(Y)` must match the column and row dimensions of *U* and *V*.

U, V

[1D or 2D array-like] The x and y components of the barb shaft.

C

[1D or 2D array-like, optional] Numeric data that defines the barb colors by colormapping via *norm* and *cmap*.

This does not support explicit colors. If you want to set colors directly, use *barbcolor* instead.

length

[float, default: 7] Length of the barb in points; the other parts of the barb are scaled against this.

pivot

[{'tip', 'middle'} or float, default: 'tip'] The part of the arrow that is anchored to the *X*, *Y* grid. The barb rotates about this point. This can also be a number, which shifts the start of the barb that many points away from grid point.

barbcolor

[color or color sequence] The color of all parts of the barb except for the flags. This parameter is analogous to the *edgecolor* parameter for polygons, which can be used instead. However this parameter will override facecolor.

flagcolor

[color or color sequence] The color of any flags on the barb. This parameter is analogous to the *facecolor* parameter for polygons, which can be used instead. However, this parameter will override facecolor. If this is not set (and *C* has not either) then *flagcolor* will be set to match *barbcolor* so that the barb has a uniform color. If *C* has been set, *flagcolor* has no effect.

sizes

[dict, optional] A dictionary of coefficients specifying the ratio of a given feature to the length of the barb. Only those values one wishes to override need to be included. These features include:

- 'spacing' - space between features (flags, full/half barbs)
- 'height' - height (distance from shaft to top) of a flag or full barb
- 'width' - width of a flag, twice the width of a full barb
- 'emptybarb' - radius of the circle used for low magnitudes

fill_empty

[bool, default: False] Whether the empty barbs (circles) that are drawn should be filled with the flag color. If they are not filled, the center is transparent.

rounding

[bool, default: True] Whether the vector magnitude should be rounded when allocating barb components. If True, the magnitude is rounded to the nearest multiple of the half-barb increment. If False, the magnitude is simply truncated to the next lowest multiple.

barb_increments

[dict, optional] A dictionary of increments specifying values to associate with different parts of the barb. Only those values one wishes to override need to be included.

- 'half' - half barbs (Default is 5)
- 'full' - full barbs (Default is 10)
- 'flag' - flags (default is 50)

flip_barb

[bool or array-like of bool, default: False] Whether the lines and flags should point opposite to normal. Normal behavior is for the barbs and lines to point right (comes from wind barbs having these features point towards low pressure in the Northern Hemisphere).

A single value is applied to all barbs. Individual barbs can be flipped by passing a bool array of the same size as *U* and *V*.

Returns

barbs

[*Barbs*]

Other Parameters

**kwargs

The barbs can further be customized using *PolyCollection* keyword arguments:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a
alpha	float or None
animated	bool
antialiased or aa or antialiaseds	bool or list of bools
array	ndarray
capstyle	{'butt', 'round', 'projecting'}
clim	(vmin: float, vmax: float)
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
cmap	*Colormap* or str or None
color	color or list of rgba tuples
contains	unknown
edgecolor or ec or edgecolors	color or list of colors or 'face'
facecolor or facecolors or fc	color or list of colors
figure	*Figure*
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or dashes or linestyles or ls	str or tuple or list thereof
linewidth or linewidths or lw	float or list of floats
norm	*Normalize* or None
offset_position	unknown
offsets	array-like (N, 2) or (2,)
path_effects	*AbstractPathEffect*
picker	None or bool or callable
pickradius	unknown

Table 216 – continued from previous page

Property	Description
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
urls	list of str or None
visible	bool
zorder	float

`__init__`(*self*, *ax*, **args*, *pivot='tip'*, *length=7*, *barbcolor=None*, *flagcolor=None*, *sizes=None*, *fill_empty=False*, *barb_increments=None*, *rounding=True*, *flip_barb=False*, ***kw*)

The constructor takes one required argument, an Axes instance, followed by the args and kwargs described by the following pyplot interface documentation:

Plot a 2D field of barbs.

Call signature:

```
barbs([X, Y], U, V, [C], **kw)
```

Where *X*, *Y* define the barb locations, *U*, *V* define the barb directions, and *C* optionally sets the color.

All arguments may be 1D or 2D. *U*, *V*, *C* may be masked arrays, but masked *X*, *Y* are not supported at present.

Barbs are traditionally used in meteorology as a way to plot the speed and direction of wind observations, but can technically be used to plot any two dimensional vector quantity. As opposed to arrows, which give vector magnitude by the length of the arrow, the barbs give more quantitative information about the vector magnitude by putting slanted lines or a triangle for various increments in magnitude, as show schematically below:

```
:                    /\    \
:                   /  \    \
:                  /    \    \    \
:                 /      \    \    \
:                ------------------------------
```

The largest increment is given by a triangle (or "flag"). After those come full lines (barbs). The smallest increment is a half line. There is only, of course, ever at most 1 half line. If the magnitude is small and only needs a single half-line and no full lines or triangles, the half-line is offset from the end of the barb so that it can be easily distinguished from barbs with a single full line. The magnitude for the barb shown above would nominally be 65, using the standard increments of 50, 10, and 5.

See also https://en.wikipedia.org/wiki/Wind_barb.

Parameters

X, Y

[1D or 2D array-like, optional] The x and y coordinates of the barb locations. See *pivot* for how the barbs are drawn to the x, y positions.

If not given, they will be generated as a uniform integer meshgrid based on the dimensions of *U* and *V*.

If *X* and *Y* are 1D but *U*, *V* are 2D, *X*, *Y* are expanded to 2D using `X, Y = np.meshgrid(X, Y)`. In this case `len(X)` and `len(Y)` must match the column and row dimensions of *U* and *V*.

U, V

[1D or 2D array-like] The x and y components of the barb shaft.

C

[1D or 2D array-like, optional] Numeric data that defines the barb colors by colormapping via *norm* and *cmap*.

This does not support explicit colors. If you want to set colors directly, use *barbcolor* instead.

length

[float, default: 7] Length of the barb in points; the other parts of the barb are scaled against this.

pivot

[{'tip', 'middle'} or float, default: 'tip'] The part of the arrow that is anchored to the *X*, *Y* grid. The barb rotates about this point. This can also be a number, which shifts the start of the barb that many points away from grid point.

barbcolor

[color or color sequence] The color of all parts of the barb except for the flags. This parameter is analogous to the *edgecolor* parameter for polygons, which can be used instead. However this parameter will override facecolor.

flagcolor

[color or color sequence] The color of any flags on the barb. This parameter is analogous to the *facecolor* parameter for polygons, which can be used instead. However, this parameter will override facecolor. If this is not set (and *C* has not either) then *flagcolor* will be set to match *barbcolor* so that the barb has a uniform color. If *C* has been set, *flagcolor* has no effect.

sizes

[dict, optional] A dictionary of coefficients specifying the ratio of a given feature to the length of the barb. Only those values one wishes to override need to be included. These features include:

- 'spacing' - space between features (flags, full/half barbs)
- 'height' - height (distance from shaft to top) of a flag or full barb
- 'width' - width of a flag, twice the width of a full barb
- 'emptybarb' - radius of the circle used for low magnitudes

fill_empty

[bool, default: False] Whether the empty barbs (circles) that are drawn should be filled with the flag color. If they are not filled, the center is transparent.

rounding

[bool, default: True] Whether the vector magnitude should be rounded when allocating barb components. If True, the magnitude is rounded to the nearest multiple of the half-barb increment. If False, the magnitude is simply truncated to the next lowest multiple.

barb_increments

[dict, optional] A dictionary of increments specifying values to associate with different parts of the barb. Only those values one wishes to override need to be included.

- 'half' - half barbs (Default is 5)
- 'full' - full barbs (Default is 10)
- 'flag' - flags (default is 50)

flip_barb

[bool or array-like of bool, default: False] Whether the lines and flags should point opposite to normal. Normal behavior is for the barbs and lines to point right (comes from wind barbs having these features point towards low pressure in the Northern Hemisphere).

A single value is applied to all barbs. Individual barbs can be flipped by passing a bool array of the same size as *U* and *V*.

Returns

barbs

[*Barbs*]

Other Parameters

**kwargs

The barbs can further be customized using *PolyCollection* keyword arguments:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a
alpha	float or None
animated	bool
antialiased or aa or antialiaseds	bool or list of bools
array	ndarray
capstyle	{'butt', 'round', 'projecting'}
clim	(vmin: float, vmax: float)
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
cmap	*Colormap* or str or None
color	color or list of rgba tuples
contains	unknown
edgecolor or ec or edgecolors	color or list of colors or 'face'
facecolor or facecolors or fc	color or list of colors
figure	*Figure*
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or dashes or linestyles or ls	str or tuple or list thereof
linewidth or linewidths or lw	float or list of floats
norm	*Normalize* or None
offset_position	unknown
offsets	array-like (N, 2) or (2,)
path_effects	*AbstractPathEffect*
picker	None or bool or callable
pickradius	unknown
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
urls	list of str or None
visible	bool

Table 217 – continued from previous page

Property	Description
zorder	float

__module__ = 'matplotlib.quiver'

barbs_doc = '\nPlot a 2D field of barbs.\n\nCall signature::\n\n barbs([X, Y], U, V, [C],

set_UVC(*self*, *U*, *V*, *C=None*)

set_offsets(*self*, *xy*)

Set the offsets for the barb polygons. This saves the offsets passed in and masks them as appropriate for the existing U/V data.

Parameters

xy

[sequence of pairs of floats]

Examples using `matplotlib.quiver.Barbs`

18.42 matplotlib.rcsetup

The rcsetup module contains the validation code for customization using Matplotlib's rc settings.

Each rc setting is assigned a function used to validate any attempted changes to that setting. The validation functions are defined in the rcsetup module, and are used to construct the rcParams global object which stores the settings and is referenced throughout Matplotlib.

The default values of the rc settings are set in the default matplotlibrc file. Any additions or deletions to the parameter set listed here should also be propagated to the `matplotlibrc.template` in Matplotlib's root source directory.

class matplotlib.rcsetup.**ValidateInStrings**(*key*, *valid*, *ignorecase=False*, *, *_deprecated_since=None*)

Bases: `object`

valid is a list of legal strings.

matplotlib.rcsetup.**cycler**(**args*, ***kwargs*)

Create a `Cycler` object much like `cycler.cycler()`, but includes input validation.

Call signatures:

```
cycler(cycler)
cycler(label=values[, label2=values2[, ...]])
cycler(label, values)
```

Form 1 copies a given `Cycler` object.

Form 2 creates a `Cycler` which cycles over one or more properties simultaneously. If multiple properties are given, their value lists must have the same length.

Form 3 creates a `Cycler` for a single property. This form exists for compatibility with the original cycler. Its use is discouraged in favor of the kwarg form, i.e. `cycler(label=values)`.

Parameters

cycler

[Cycler] Copy constructor for Cycler.

label

[str] The property key. Must be a valid *Artist* property. For example, 'color' or 'linestyle'. Aliases are allowed, such as 'c' for 'color' and 'lw' for 'linewidth'.

values

[iterable] Finite-length iterable of the property values. These values are validated and will raise a ValueError if invalid.

Returns

Cycler

A new `Cycler` for the given properties.

Examples

Creating a cycler for a single property:

```
>>> c = cycler(color=['red', 'green', 'blue'])
```

Creating a cycler for simultaneously cycling over multiple properties (e.g. red circle, green plus, blue cross):

```
>>> c = cycler(color=['red', 'green', 'blue'],
...            marker=['o', '+', 'x'])
```

matplotlib.rcsetup.**update_savefig_format**(*value*)

[*Deprecated*]

Notes

Deprecated since version 3.2:

matplotlib.rcsetup.**validate_animation_writer_path**(*p*)
[*Deprecated*]

Notes

Deprecated since version 3.2:

matplotlib.rcsetup.**validate_any**(*s*)

matplotlib.rcsetup.**validate_anylist**(*s*)

matplotlib.rcsetup.**validate_aspect**(*s*)

matplotlib.rcsetup.**validate_axisbelow**(*s*)

matplotlib.rcsetup.**validate_backend**(*s*)

matplotlib.rcsetup.**validate_bbox**(*s*)

matplotlib.rcsetup.**validate_bool**(*b*)
Convert b to `bool` or raise.

matplotlib.rcsetup.**validate_bool_maybe_none**(*b*)
[*Deprecated*] Convert b to `bool` or raise, passing through *None*.

Notes

Deprecated since version 3.3.

matplotlib.rcsetup.**validate_capstyle**(*s*)

matplotlib.rcsetup.**validate_capstylelist**(*s*)

matplotlib.rcsetup.**validate_color**(*s*)
Return a valid color arg.

matplotlib.rcsetup.**validate_color_for_prop_cycle**(*s*)

matplotlib.rcsetup.**validate_color_or_auto**(*s*)

matplotlib.rcsetup.**validate_color_or_inherit**(*s*)
Return a valid color arg.

matplotlib.rcsetup.**validate_colorlist**(*s*)
return a list of colorspecs

matplotlib.rcsetup.**validate_cycler**(*s*)
Return a Cycler object from a string repr or the object itself.

matplotlib.rcsetup.**validate_dashlist**(*s*)
return a list of floats

matplotlib.rcsetup.**validate_dpi**(*s*)
Confirm s is string 'figure' or convert s to float or raise.

matplotlib.rcsetup.**validate_fillstylelist**(*s*)

matplotlib.rcsetup.**validate_float**(*s*)

matplotlib.rcsetup.**validate_float_or_None**(*s*)

matplotlib.rcsetup.**validate_floatlist**(*s*)
return a list of floats

matplotlib.rcsetup.**validate_font_properties**(*s*)

matplotlib.rcsetup.**validate_fontsize**(*s*)

matplotlib.rcsetup.**validate_fontsize_None**(*s*)

matplotlib.rcsetup.**validate_fontsizelist**(*s*)

matplotlib.rcsetup.**validate_fonttype**(*s*)
Confirm that this is a Postscript or PDF font type that we know how to convert to.

matplotlib.rcsetup.**validate_fontweight**(*s*)

matplotlib.rcsetup.**validate_hatch**(*s*)
Validate a hatch pattern. A hatch pattern string can have any sequence of the following characters: `\ / | - + * . x o O`.

matplotlib.rcsetup.**validate_hatchlist**(*s*)
Validate a hatch pattern. A hatch pattern string can have any sequence of the following characters: `\ / | - + * . x o O`.

matplotlib.rcsetup.**validate_hinting**(*s*)
[*Deprecated*]

Notes

Deprecated since version 3.3:

matplotlib.rcsetup.**validate_hist_bins**(*s*)

matplotlib.rcsetup.**validate_int**(*s*)

matplotlib.rcsetup.**validate_int_or_None**(*s*)

matplotlib.rcsetup.**validate_joinstyle**(*s*)

matplotlib.rcsetup.**validate_joinstylelist**(*s*)

matplotlib.rcsetup.**validate_markevery**(*s*)
Validate the markevery property of a Line2D object.

Parameters

s

[None, int, float, slice, length-2 tuple of ints,] length-2 tuple of floats, list of ints

Returns

None, int, float, slice, length-2 tuple of ints,

length-2 tuple of floats, list of ints

matplotlib.rcsetup.**validate_markeverylist**(*s*)

Validate the markevery property of a Line2D object.

Parameters

s

[None, int, float, slice, length-2 tuple of ints,] length-2 tuple of floats, list of ints

Returns

None, int, float, slice, length-2 tuple of ints,

length-2 tuple of floats, list of ints

matplotlib.rcsetup.**validate_movie_writer**(*s*)

[*Deprecated*]

Notes

Deprecated since version 3.3:

matplotlib.rcsetup.**validate_nseq_float**(*n*)

[*Deprecated*]

Notes

Deprecated since version 3.3:

matplotlib.rcsetup.**validate_nseq_int**(*n*)

[*Deprecated*]

Notes

Deprecated since version 3.3:

matplotlib.rcsetup.**validate_path_exists**(*s*)
[*Deprecated*] If s is a path, return s, else False

Notes

Deprecated since version 3.2.

matplotlib.rcsetup.**validate_ps_distiller**(*s*)

matplotlib.rcsetup.**validate_sketch**(*s*)

matplotlib.rcsetup.**validate_string**(*s*)

matplotlib.rcsetup.**validate_string_or_None**(*s*)

matplotlib.rcsetup.**validate_stringlist**(*s*)
return a list of strings

matplotlib.rcsetup.**validate_webagg_address**(*s*)
[*Deprecated*]

Notes

Deprecated since version 3.3:

matplotlib.rcsetup.**validate_whiskers**(*s*)

18.43 matplotlib.sankey

Module for creating Sankey diagrams using Matplotlib.

class matplotlib.sankey.**Sankey**(*ax=None*, *scale=1.0*, *unit=''*, *format='%G'*, *gap=0.25*, *radius=0.1*, *shoulder=0.03*, *offset=0.15*, *head_angle=100*, *margin=0.4*, *tolerance=1e-06*, ***kwargs*)

Bases: `object`

Sankey diagram.

Sankey diagrams are a specific type of flow diagram, in which the width of the arrows is shown proportionally to the flow quantity. They are typically used to visualize energy or material or cost transfers between processes. Wikipedia (6/1/2011)

Create a new Sankey instance.

The optional arguments listed below are applied to all subdiagrams so that there is consistent alignment and formatting.

In order to draw a complex Sankey diagram, create an instance of *Sankey* by calling it without any kwargs:

```
sankey = Sankey()
```

Then add simple Sankey sub-diagrams:

```
sankey.add() # 1
sankey.add() # 2
#...
sankey.add() # n
```

Finally, create the full diagram:

```
sankey.finish()
```

Or, instead, simply daisy-chain those calls:

```
Sankey().add().add...  .add().finish()
```

Other Parameters

ax

[*Axes*] Axes onto which the data should be plotted. If *ax* isn't provided, new Axes will be created.

scale

[float] Scaling factor for the flows. *scale* sizes the width of the paths in order to maintain proper layout. The same scale is applied to all subdiagrams. The value should be chosen such that the product of the scale and the sum of the inputs is approximately 1.0 (and the product of the scale and the sum of the outputs is approximately -1.0).

unit

[str] The physical unit associated with the flow quantities. If *unit* is None, then none of the quantities are labeled.

format

[str] A Python number formatting string to be used in labeling the flow as a quantity (i.e., a number times a unit, where the unit is given).

gap

[float] Space between paths that break in/break away to/from the top or bottom.

radius

[float] Inner radius of the vertical paths.

shoulder

[float] Size of the shoulders of output arrows.

offset

[float] Text offset (from the dip or tip of the arrow).

head_angle

[float] Angle, in degrees, of the arrow heads (and negative of the angle of the tails).

margin

[float] Minimum space between Sankey outlines and the edge of the plot area.

tolerance

[float] Acceptable maximum of the magnitude of the sum of flows. The magnitude of the sum of connected flows cannot be greater than *tolerance*.

****kwargs**

Any additional keyword arguments will be passed to *add()*, which will create the first subdiagram.

See also:

Sankey.add

Sankey.finish

Examples

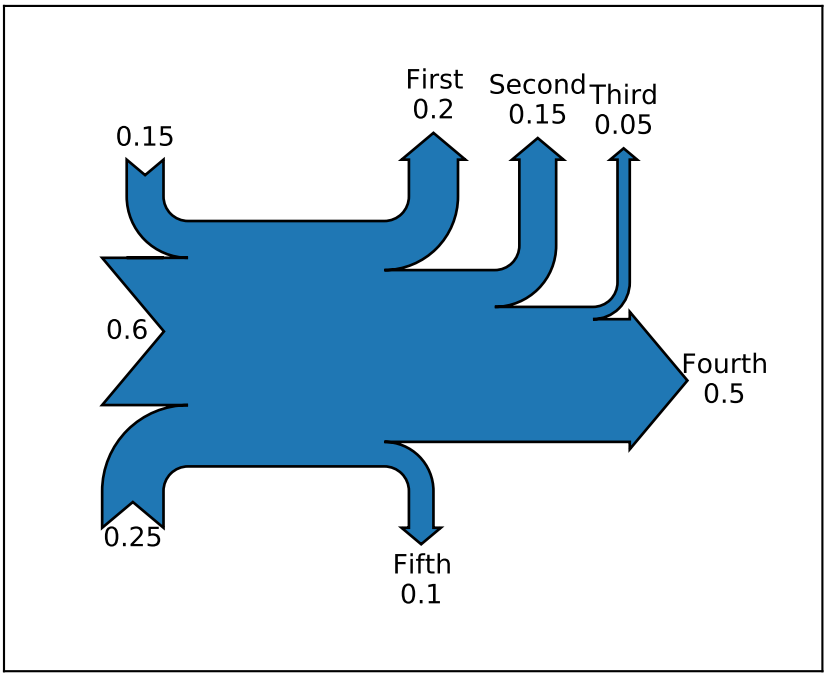
add(*self*, *patchlabel=''*, *flows=None*, *orientations=None*, *labels=''*, *trunklength=1.0*, *pathlengths=0.25*, *prior=None*, *connect=0, 0*, *rotation=0*, ***kwargs*)

Add a simple Sankey diagram with flows at the same hierarchical level.

Parameters

patchlabel

The default settings produce a diagram like this.



[str] Label to be placed at the center of the diagram. Note that *label* (not *patchlabel*) can be passed as keyword argument to create an entry in the legend.

flows

[list of float] Array of flow values. By convention, inputs are positive and outputs are negative.

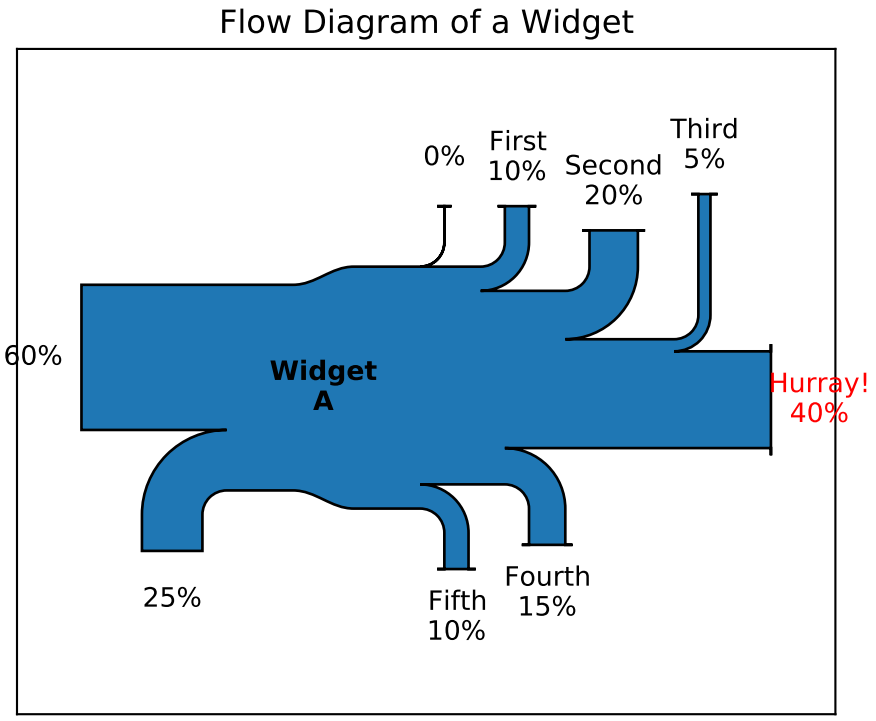
Flows are placed along the top of the diagram from the inside out in order of their index within *flows*. They are placed along the sides of the diagram from the top down and along the bottom from the outside in.

If the sum of the inputs and outputs is nonzero, the discrepancy will appear as a cubic Bezier curve along the top and bottom edges of the trunk.

orientations

[list of {-1, 0, 1}] List of orientations of the flows (or a single orientation to be used for all flows). Valid values are 0 (inputs from the left, outputs to the right), 1 (from and to the top) or -1 (from and to the bottom).

labels



[list of (str or None)] List of labels for the flows (or a single label to be used for all flows). Each label may be *None* (no label), or a labeling string. If an entry is a (possibly empty) string, then the quantity for the corresponding flow will be shown below the string. However, if the *unit* of the main diagram is None, then quantities are never shown, regardless of the value of this argument.

trunklength

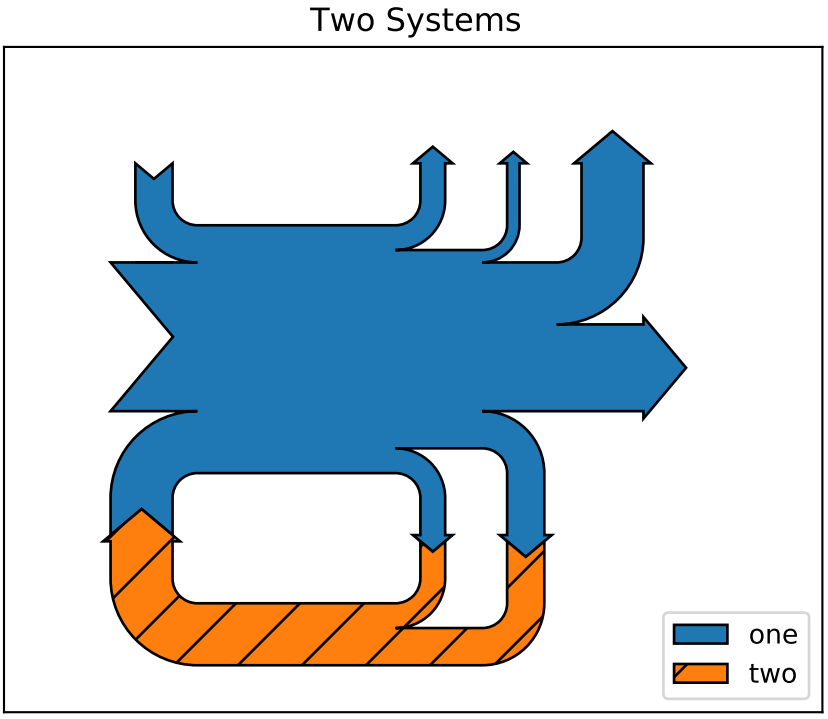
[float] Length between the bases of the input and output groups (in data-space units).

pathlengths

[list of float] List of lengths of the vertical arrows before break-in or after break-away. If a single value is given, then it will be applied to the first (inside) paths on the top and bottom, and the length of all other arrows will be justified accordingly. The *pathlengths* are not applied to the horizontal inputs and outputs.

prior

[int] Index of the prior diagram to which this diagram should



be connected.

connect

[(int, int)] A (prior, this) tuple indexing the flow of the prior diagram and the flow of this diagram which should be connected. If this is the first diagram or *prior* is *None*, *connect* will be ignored.

rotation

[float] Angle of rotation of the diagram in degrees. The interpretation of the *orientations* argument will be rotated accordingly (e.g., if *rotation* == 90, an *orientations* entry of 1 means to/from the left). *rotation* is ignored if this diagram is connected to an existing one (using *prior* and *connect*).

Returns

Sankey

The current *Sankey* instance.

Other Parameters

****kwargs**

Additional keyword arguments set *matplotlib.patches.PathPatch* properties, listed below. For example, one may want to use `fill=False` or `label="A legend entry"`.

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and retur
alpha	float or None
animated	bool
antialiased or aa	unknown
capstyle	{'butt', 'round', 'projecting'}
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color	color
contains	unknown
edgecolor or ec	color or None or 'auto'
facecolor or fc	color or None
figure	*Figure*
fill	bool
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float or None
path_effects	*AbstractPathEffect*
picker	None or bool or callable
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
visible	bool
zorder	float

See also:

Sankey.finish

`finish`(*self*)

Adjust the axes and return a list of information about the Sankey subdiagram(s).

Return value is a list of subdiagrams represented with the following fields:

Field	Description
patch	Sankey outline (an instance of *PathPatch*)
flows	values of the flows (positive for input, negative for output)
angles	list of angles of the arrows [deg/90] For example, if the diagram has not been rotated, an input to the top side will have an angle of 3 (DOWN), and an output from the top side will have an angle of 1 (UP). If a flow has been skipped (because its magnitude is less than *tolerance*), then its angle will be *None*.
tips	array in which each row is an [x, y] pair indicating the positions of the tips (or "dips") of the flow paths If the magnitude of a flow is less the *tolerance* for the instance of *Sankey*, the flow is skipped and its tip will be at the center of the diagram.
text	*Text* instance for the label of the diagram
texts	list of *Text* instances for the labels of flows

See also:

Sankey.add

18.44 matplotlib.scale

Scales define the distribution of data values on an axis, e.g. a log scaling.

They are attached to an *Axis* and hold a *Transform*, which is responsible for the actual data transformation.

See also *axes.Axes.set_xscale* and the scales examples in the documentation.

class matplotlib.scale.**FuncScale**(*axis*, *functions*)

Bases: *matplotlib.scale.ScaleBase*

Provide an arbitrary scale with user-supplied function for the axis.

Parameters

axis

[*Axis*] The axis for the scale.

functions

[(callable, callable)] two-tuple of the forward and inverse functions for the scale. The forward function must be monotonic.

Both functions must have the signature:

```
def forward(values: array-like) -> array-like
```

`get_transform`(*self*)
Return the *FuncTransform* associated with this scale.

`name = 'function'`

`set_default_locators_and_formatters`(*self*, *axis*)
Set the locators and formatters of *axis* to instances suitable for this scale.

`class matplotlib.scale.FuncScaleLog`(*axis*, *functions*, *base=10*)
Bases: *matplotlib.scale.LogScale*

Provide an arbitrary scale with user-supplied function for the axis and then put on a logarithmic axes.

Parameters

axis

[*matplotlib.axis.Axis*] The axis for the scale.

functions

[(callable, callable)] two-tuple of the forward and inverse functions for the scale. The forward function must be monotonic.

Both functions must have the signature:

```
def forward(values: array-like) -> array-like
```

base

[float, default: 10] Logarithmic base of the scale.

`property base`

`get_transform`(*self*)
Return the *Transform* associated with this scale.

`name = 'functionlog'`

`class matplotlib.scale.FuncTransform`(*forward*, *inverse*)
Bases: *matplotlib.transforms.Transform*

A simple transform that takes and arbitrary function for the forward and inverse transform.

Parameters

forward

[callable] The forward function for the transform. This function must have an inverse and, for best behavior, be monotonic. It must have the signature:

```
def forward(values: array-like) -> array-like
```

inverse

[callable] The inverse of the forward function. Signature as `forward`.

`has_inverse = True`

`input_dims = 1`

`inverted`(*self*)

Return the corresponding inverse transformation.

It holds `x == self.inverted().transform(self.transform(x))`.

The return value of this method should be treated as temporary. An update to *self* does not cause a corresponding update to its inverted copy.

`is_separable = True`

`output_dims = 1`

`transform_non_affine`(*self*, *values*)

Apply only the non-affine part of this transformation.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally equivalent to `transform(values)`. In affine transformations, this is always a no-op.

Parameters

values

[array] The input values as NumPy array of length *input_dims* or shape (N x *input_dims*).

Returns

array

The output values as NumPy array of length *input_dims* or shape (N x *output_dims*), depending on the input.

`class matplotlib.scale.InvertedLogTransform`(*base*)

Bases: *matplotlib.transforms.Transform*

Parameters

shorthand_name

[str] A string representing the "name" of the transform. The name carries no significance other than to improve the readability of `str(transform)` when DEBUG=True.

has_inverse = True

input_dims = 1

inverted(*self*)

Return the corresponding inverse transformation.

It holds `x == self.inverted().transform(self.transform(x))`.

The return value of this method should be treated as temporary. An update to *self* does not cause a corresponding update to its inverted copy.

is_separable = True

output_dims = 1

transform_non_affine(*self*, *a*)

Apply only the non-affine part of this transformation.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally equivalent to `transform(values)`. In affine transformations, this is always a no-op.

Parameters

values

[array] The input values as NumPy array of length *input_dims* or shape (N x *input_dims*).

Returns

array

The output values as NumPy array of length *input_dims* or shape (N x *output_dims*), depending on the input.

class matplotlib.scale.InvertedSymmetricalLogTransform(*base*, *linthresh*, *linscale*)

Bases: *matplotlib.transforms.Transform*

Parameters

shorthand_name

[str] A string representing the "name" of the transform. The name carries no significance other than to improve the readability of `str(transform)` when DEBUG=True.

has_inverse = True

input_dims = 1

`inverted`(*self*)
Return the corresponding inverse transformation.

It holds `x == self.inverted().transform(self.transform(x))`.

The return value of this method should be treated as temporary. An update to *self* does not cause a corresponding update to its inverted copy.

`is_separable = True`

`output_dims = 1`

`transform_non_affine`(*self*, *a*)
Apply only the non-affine part of this transformation.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally equivalent to `transform(values)`. In affine transformations, this is always a no-op.

Parameters

values

[array] The input values as NumPy array of length *input_dims* or shape (N x *input_dims*).

Returns

array

The output values as NumPy array of length *input_dims* or shape (N x *output_dims*), depending on the input.

`class matplotlib.scale.LinearScale`(*axis*, ***kwargs*)
Bases: *matplotlib.scale.ScaleBase*

The default linear scale.

`get_transform`(*self*)
Return the transform for linear scaling, which is just the *IdentityTransform*.

`name = 'linear'`

`set_default_locators_and_formatters`(*self*, *axis*)
Set the locators and formatters of *axis* to instances suitable for this scale.

`class matplotlib.scale.LogScale`(*axis*, ***kwargs*)
Bases: *matplotlib.scale.ScaleBase*

A standard logarithmic scale. Care is taken to only plot positive values.

Parameters

axis

[*Axis*] The axis for the scale.

base

[float, default: 10] The base of the logarithm.

nonpositive

[{'clip', 'mask'}, default: 'clip'] Determines the behavior for non-positive values. They can either be masked as invalid, or clipped to a very small positive number.

subs

[sequence of int, default: None] Where to place the subticks between each major tick. For example, in a log10 scale, `[2, 3, 4, 5, 6, 7, 8, 9]` will place 8 logarithmically spaced minor ticks between each major tick.

property InvertedLogTransform

property LogTransform

property base

get_transform(*self*)
Return the *LogTransform* associated with this scale.

limit_range_for_scale(*self*, *vmin*, *vmax*, *minpos*)
Limit the domain to positive values.

name = 'log'

set_default_locators_and_formatters(*self*, *axis*)
Set the locators and formatters of *axis* to instances suitable for this scale.

class matplotlib.scale.LogTransform(**kwargs)
Bases: *matplotlib.transforms.Transform*

Parameters

shorthand_name

[str] A string representing the "name" of the transform. The name carries no significance other than to improve the readability of `str(transform)` when DEBUG=True.

has_inverse = True

input_dims = 1

inverted(*self*)
Return the corresponding inverse transformation.

It holds `x == self.inverted().transform(self.transform(x))`.

The return value of this method should be treated as temporary. An update to *self* does not cause a corresponding update to its inverted copy.

is_separable = True

output_dims = 1

transform_non_affine(*self*, *a*)

Apply only the non-affine part of this transformation.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally equivalent to `transform(values)`. In affine transformations, this is always a no-op.

Parameters

values

[array] The input values as NumPy array of length *input_dims* or shape (N x *input_dims*).

Returns

array

The output values as NumPy array of length *input_dims* or shape (N x *output_dims*), depending on the input.

class matplotlib.scale.LogisticTransform(***kwargs*)

Bases: *matplotlib.transforms.Transform*

Parameters

shorthand_name

[str] A string representing the "name" of the transform. The name carries no significance other than to improve the readability of `str(transform)` when DEBUG=True.

has_inverse = True

input_dims = 1

inverted(*self*)

Return the corresponding inverse transformation.

It holds `x == self.inverted().transform(self.transform(x))`.

The return value of this method should be treated as temporary. An update to *self* does not cause a corresponding update to its inverted copy.

is_separable = True

output_dims = 1

transform_non_affine(*self*, *a*)

logistic transform (base 10)

class matplotlib.scale.LogitScale(**kwargs)

Bases: *matplotlib.scale.ScaleBase*

Logit scale for data between zero and one, both excluded.

This scale is similar to a log scale close to zero and to one, and almost linear around 0.5. It maps the interval]0, 1[onto]-infty, +infty[.

Parameters

axis

[*matplotlib.axis.Axis*] Currently unused.

nonpositive

[{'mask', 'clip'}] Determines the behavior for values beyond the open interval]0, 1[. They can either be masked as invalid, or clipped to a number very close to 0 or 1.

use_overline

[bool, default: False] Indicate the usage of survival notation (overline{x}) in place of standard notation (1-x) for probability close to one.

one_half

[str, default: r"frac{1}{2}"] The string used for ticks formatter to represent 1/2.

get_transform(*self*)

Return the *LogitTransform* associated with this scale.

limit_range_for_scale(*self*, *vmin*, *vmax*, *minpos*)

Limit the domain to values between 0 and 1 (excluded).

name = 'logit'

set_default_locators_and_formatters(*self*, *axis*)

Set the locators and formatters of *axis* to instances suitable for this scale.

class matplotlib.scale.LogitTransform(**kwargs)

Bases: *matplotlib.transforms.Transform*

Parameters

shorthand_name

[str] A string representing the "name" of the transform. The name carries no significance other than to improve the readability of `str(transform)` when DEBUG=True.

has_inverse = True

input_dims = 1

`inverted`(*self*)
Return the corresponding inverse transformation.

It holds `x == self.inverted().transform(self.transform(x))`.

The return value of this method should be treated as temporary. An update to *self* does not cause a corresponding update to its inverted copy.

`is_separable = True`

`output_dims = 1`

`transform_non_affine`(*self*, *a*)
logit transform (base 10), masked or clipped

`class matplotlib.scale.ScaleBase`(*axis*, ***kwargs*)
Bases: `object`

The base class for all scales.

Scales are separable transformations, working on a single dimension.

Any subclasses will want to override:

- `name`
- *get_transform()*
- *set_default_locators_and_formatters()*

And optionally:

- *limit_range_for_scale()*

Construct a new scale.

Notes

The following note is for scale implementors.

For back-compatibility reasons, scales take an *Axis* object as first argument. However, this argument should not be used: a single scale object should be usable by multiple *Axis*es at the same time.

`get_transform`(*self*)
Return the *Transform* object associated with this scale.

`limit_range_for_scale`(*self*, *vmin*, *vmax*, *minpos*)
Return the range *vmin*, *vmax*, restricted to the domain supported by this scale (if any).

minpos should be the minimum positive value in the data. This is used by log scales to determine a minimum value.

`set_default_locators_and_formatters`(*self*, *axis*)
Set the locators and formatters of *axis* to instances suitable for this scale.

class matplotlib.scale.SymmetricalLogScale(*axis*, ***kwargs*)

Bases: *matplotlib.scale.ScaleBase*

The symmetrical logarithmic scale is logarithmic in both the positive and negative directions from the origin.

Since the values close to zero tend toward infinity, there is a need to have a range around zero that is linear. The parameter *linthresh* allows the user to specify the size of this range (-*linthresh*, *linthresh*).

Parameters

base

[float, default: 10] The base of the logarithm.

linthresh

[float, default: 2] Defines the range `(-x, x)`, within which the plot is linear. This avoids having the plot go to infinity around zero.

subs

[sequence of int] Where to place the subticks between each major tick. For example, in a log10 scale: `[2, 3, 4, 5, 6, 7, 8, 9]` will place 8 logarithmically spaced minor ticks between each major tick.

linscale

[float, optional] This allows the linear range `(-linthresh, linthresh)` to be stretched relative to the logarithmic range. Its value is the number of decades to use for each half of the linear range. For example, when *linscale* == 1.0 (the default), the space used for the positive and negative halves of the linear range will be equal to one decade in the logarithmic range.

Construct a new scale.

Notes

The following note is for scale implementors.

For back-compatibility reasons, scales take an *Axis* object as first argument. However, this argument should not be used: a single scale object should be usable by multiple *Axis*es at the same time.

property InvertedSymmetricalLogTransform

property SymmetricalLogTransform

property base

`get_transform`(*self*)
Return the *SymmetricalLogTransform* associated with this scale.

`property linscale`

`property linthresh`

`name = 'symlog'`

`set_default_locators_and_formatters`(*self*, *axis*)
Set the locators and formatters of *axis* to instances suitable for this scale.

`class matplotlib.scale.SymmetricalLogTransform`(*base*, *linthresh*, *linscale*)
Bases: *matplotlib.transforms.Transform*

Parameters

shorthand_name

[str] A string representing the "name" of the transform. The name carries no significance other than to improve the readability of `str(transform)` when DEBUG=True.

`has_inverse = True`

`input_dims = 1`

`inverted`(*self*)
Return the corresponding inverse transformation.

It holds `x == self.inverted().transform(self.transform(x))`.

The return value of this method should be treated as temporary. An update to *self* does not cause a corresponding update to its inverted copy.

`is_separable = True`

`output_dims = 1`

`transform_non_affine`(*self*, *a*)
Apply only the non-affine part of this transformation.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally equivalent to `transform(values)`. In affine transformations, this is always a no-op.

Parameters

values

[array] The input values as NumPy array of length *input_dims* or shape (N x *input_dims*).

Returns

array

The output values as NumPy array of length *input_dims* or shape (N x *output_dims*), depending on the input.

matplotlib.scale.**get_scale_names**()
Return the names of the available scales.

matplotlib.scale.**register_scale**(*scale_class*)
Register a new kind of scale.

Parameters

scale_class

[subclass of *ScaleBase*] The scale to register.

matplotlib.scale.**scale_factory**(*scale*, *axis*, ***kwargs*)
Return a scale class by name.

Parameters

scale

[{'function', 'functionlog', 'linear', 'log', 'logit', 'symlog'}]

axis

[*matplotlib.axis.Axis*]

18.45 matplotlib.sphinxext.plot_directive

18.45.1 A directive for including a Matplotlib plot in a Sphinx document

By default, in HTML output, `plot` will include a .png file with a link to a high-res .png and .pdf. In LaTeX output, it will include a .pdf.

The source code for the plot may be included in one of three ways:

1. **A path to a source file** as the argument to the directive:

```
.. plot:: path/to/plot.py
```

When a path to a source file is given, the content of the directive may optionally contain a caption for the plot:

```
.. plot:: path/to/plot.py

   The plot's caption.
```

Additionally, one may specify the name of a function to call (with no arguments) immediately after importing the module:

```
.. plot:: path/to/plot.py plot_function1
```

2. Included as **inline content** to the directive:

```
.. plot::

   import matplotlib.pyplot as plt
   import matplotlib.image as mpimg
   import numpy as np
   img = mpimg.imread('_static/stinkbug.png')
   imgplot = plt.imshow(img)
```

3. Using **doctest** syntax:

```
.. plot::

   A plotting example:
   >>> import matplotlib.pyplot as plt
   >>> plt.plot([1, 2, 3], [4, 5, 6])
```

Options

The `plot` directive supports the following options:

format

[{'python', 'doctest'}] The format of the input.

include-source

[bool] Whether to display the source code. The default can be changed using the `plot_include_source` variable in `conf.py`.

encoding

[str] If this source file is in a non-UTF8 or non-ASCII encoding, the encoding must be specified using the `:encoding:` option. The encoding will not be inferred using the `-*- coding -*-` metacomment.

context

[bool or str] If provided, the code will be run in the context of all previous plot directives for which the `:context:` option was specified. This only applies to inline code plot directives, not those run from files. If the `:context: reset` option is specified, the context is reset for this and future plots, and previous figures are closed prior to running the code. `:context: close-figs` keeps the context but closes previous figures before running the code.

nofigs

[bool] If specified, the code block will be run, but no figures will be inserted. This is usually useful with the `:context:` option.

Additionally, this directive supports all of the options of the `image` directive, except for *target* (since plot will add its own target). These include *alt*, *height*, *width*, *scale*, *align* and *class*.

Configuration options

The plot directive has the following configuration options:

plot_include_source

Default value for the include-source option

plot_html_show_source_link

Whether to show a link to the source in HTML.

plot_pre_code

Code that should be executed before each plot. If not specified or None it will default to a string containing:

```
import numpy as np
from matplotlib import pyplot as plt
```

plot_basedir

Base directory, to which `plot::` file names are relative to. (If None or empty, file names are relative to the directory where the file containing the directive is.)

plot_formats

File formats to generate. List of tuples or strings:

```
[(suffix, dpi), suffix, ...]
```

that determine the file format and the DPI. For entries whose DPI was omitted, sensible defaults are chosen. When passing from the command line through sphinx_build the list should be passed as suffix:dpi,suffix:dpi, ...

plot_html_show_formats

Whether to show links to the files in HTML.

plot_rcparams

A dictionary containing any non-standard rcParams that should be applied before each plot.

plot_apply_rcparams

By default, rcParams are applied when `:context:` option is not used in a plot directive. This configuration option overrides this behavior and applies rcParams before each plot.

plot_working_directory

By default, the working directory will be changed to the directory of the example, so the code can get at its data files, if any. Also its path will be added to `sys.path` so it can import any helper modules sitting beside it. This configuration option can be used to specify a central directory (also added to `sys.path`) where data files and helper modules for all code are located.

plot_template

Provide a customized template for preparing restructured text.

`class matplotlib.sphinxext.plot_directive.PlotDirective(`*name*, *arguments*, *options*, *content*, *lineno*, *content_offset*, *block_text*, *state*, *state_machine*`)`

The `.. plot::` directive, as documented in the module's docstring.

`run(`*self*`)`

Run the plot directive.

`exception matplotlib.sphinxext.plot_directive.PlotError`

`matplotlib.sphinxext.plot_directive.mark_plot_labels(`*app*, *document*`)`

To make plots referenceable, we need to move the reference from the "htmlonly" (or "latexonly") node to the actual figure node itself.

`matplotlib.sphinxext.plot_directive.out_of_date(`*original*, *derived*`)`

Return whether *derived* is out-of-date relative to *original*, both of which are full file paths.

`matplotlib.sphinxext.plot_directive.render_figures(`*code*, *code_path*, *output_dir*, *output_base*, *context*, *function_name*, *config*, *context_reset=False*, *close_figs=False*`)`

Run a pyplot script and save the images in *output_dir*.

Save the images under *output_dir* with file names derived from *output_base*

`matplotlib.sphinxext.plot_directive.run_code(`*code*, *code_path*, *ns=None*, *function_name=None*`)`

Import a Python module from a path, and run the function given by name, if function_name is not None.

`matplotlib.sphinxext.plot_directive.split_code_at_show(`*text*`)`

Split code at plt.show().

`matplotlib.sphinxext.plot_directive.unescape_doctest(`*text*`)`

Extract code from a piece of text, which contains either Python code or doctests.

18.46 matplotlib.spines

class matplotlib.spines.**Spine**(*axes*, *spine_type*, *path*, ***kwargs*)

Bases: *matplotlib.patches.Patch*

An axis spine -- the line noting the data area boundaries.

Spines are the lines connecting the axis tick marks and noting the boundaries of the data area. They can be placed at arbitrary positions. See *set_position* for more information.

The default position is `('outward', 0)`.

Spines are subclasses of *Patch*, and inherit much of their behavior.

Spines draw a line, a circle, or an arc depending if *set_patch_line*, *set_patch_circle*, or *set_patch_arc* has been called. Line-like is the default.

Parameters

axes

[*Axes*] The *Axes* instance containing the spine.

spine_type

[str] The spine type.

path

[*Path*] The *Path* instance used to draw the spine.

Other Parameters

****kwargs**

Valid keyword arguments are:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and retur
alpha	float or None
animated	bool
antialiased or aa	unknown
capstyle	{'butt', 'round', 'projecting'}
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color	color
contains	unknown
edgecolor or ec	color or None or 'auto'
facecolor or fc	color or None

Table 219 – continued from previous page

Property	Description
figure	*Figure*
fill	bool
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float or None
path_effects	*AbstractPathEffect*
picker	None or bool or callable
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
visible	bool
zorder	float

classmethod arc_spine(*axes*, *spine_type*, *center*, *radius*, *theta1*, *theta2*, ***kwargs*)

Create and return an arc *Spine*.

classmethod circular_spine(*axes*, *center*, *radius*, ***kwargs*)

Create and return a circular *Spine*.

cla(*self*)

Clear the current spine.

draw(*self*, *renderer*)

Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

Parameters

renderer

[*RendererBase* subclass.]

Notes

This method is overridden in the Artist subclasses.

`get_bounds`(*self*)
Get the bounds of the spine.

`get_patch_transform`(*self*)
Return the *Transform* instance mapping patch coordinates to data coordinates.

For example, one may define a patch of a circle which represents a radius of 5 by providing coordinates for a unit circle, and a transform which scales the coordinates (the patch coordinate) by 5.

`get_path`(*self*)
Return the path of this patch.

`get_position`(*self*)
Return the spine position.

`get_smart_bounds`(*self*)
[*Deprecated*] Return whether the spine has smart bounds.

Notes

Deprecated since version 3.2.

`get_spine_transform`(*self*)
Return the spine transform.

`get_window_extent`(*self*, *renderer=None*)
Return the window extent of the spines in display space, including padding for ticks (but not their labels)

See also:

matplotlib.axes.Axes.get_tightbbox

matplotlib.axes.Axes.get_window_extent

`classmethod linear_spine`(*axes*, *spine_type*, ***kwargs*)
Create and return a linear *Spine*.

`register_axis`(*self*, *axis*)
Register an axis.

An axis should be registered with its corresponding spine from the Axes instance. This allows the spine to clear any axis properties when needed.

`set_bounds`(*self*, *low=None*, *high=None*)
Set the spine bounds.

Parameters

low

[float or None, optional] The lower spine bound. Passing *None* leaves the limit unchanged.

The bounds may also be passed as the tuple (*low*, *high*) as the first positional argument.

high

[float or None, optional] The higher spine bound. Passing *None* leaves the limit unchanged.

`set_color`(*self*, *c*)
Set the edgecolor.

Parameters

c

[color]

Notes

This method does not modify the facecolor (which defaults to "none"), unlike the *Patch.set_color* method defined in the parent class. Use *Patch.set_facecolor* to set the facecolor.

`set_patch_arc`(*self*, *center*, *radius*, *theta1*, *theta2*)
Set the spine to be arc-like.

`set_patch_circle`(*self*, *center*, *radius*)
Set the spine to be circular.

`set_patch_line`(*self*)
Set the spine to be linear.

`set_position`(*self*, *position*)
Set the position of the spine.

Spine position is specified by a 2 tuple of (position type, amount). The position types are:

- 'outward': place the spine out from the data area by the specified number of points. (Negative values place the spine inwards.)
- 'axes': place the spine at the specified Axes coordinate (0 to 1).
- 'data': place the spine at the specified data coordinate.

Additionally, shorthand notations define a special positions:

- 'center' -> ('axes', 0.5)

- 'zero' -> ('data', 0.0)

`set_smart_bounds`(*self*, *value*)
[*Deprecated*] Set the spine and associated axis to have smart bounds.

Notes

Deprecated since version 3.2.

18.47 matplotlib.style

Styles are predefined sets of *rcParams* that define the visual appearance of a plot.

Customizing Matplotlib with style sheets and rcParams describes the mechanism and usage of styles.

The /gallery/style_sheets/style_sheets_reference gives an overview of the builtin styles.

`matplotlib.style.context`(*style*, *after_reset=False*)
Context manager for using style settings temporarily.

Parameters

style

[str, dict, Path or list] A style specification. Valid options are:

str	The name of a style or a path/URL to a style file. For a list of available style names, see `style.available`.
dict	Dictionary with valid key/value pairs for *matplotlib.rcParams*.
Path	A path-like object which is a path to a style file.
list	A list of style specifiers (str, Path or dict) applied from first to last in the list.

after_reset

[bool] If True, apply style after resetting settings to their defaults; otherwise, apply style on top of the current settings.

`matplotlib.style.reload_library`()
Reload the style library.

`matplotlib.style.use`(*style*)
Use Matplotlib style settings from a style specification.

The style name of 'default' is reserved for reverting back to the default style settings.

Note: This updates the *rcParams* with the settings from the style. *rcParams* not defined in the style are kept.

Parameters

style

[str, dict, Path or list] A style specification. Valid options are:

str	The name of a style or a path/URL to a style file. For a list of available style names, see `style.available`.
dict	Dictionary with valid key/value pairs for *matplotlib.rcParams*.
Path	A path-like object which is a path to a style file.
list	A list of style specifiers (str, Path or dict) applied from first to last in the list.

`matplotlib.style.library`
A dict mapping from style name to *RcParams* defining that style.

This is meant to be read-only. Use *reload_library* to update.

`matplotlib.style.available`
List of the names of the available styles.

This is meant to be read-only. Use *reload_library* to update.

18.48 matplotlib.table

Tables drawing.

Use the factory function *table* to create a ready-made table from texts. If you need more control, use the *Table* class and its methods.

The table consists of a grid of cells, which are indexed by (row, column). The cell (0, 0) is positioned at the top left.

Thanks to John Gill for providing the class and table.

`class matplotlib.table.Cell(`*xy*, *width*, *height*, *edgecolor='k'*, *facecolor='w'*, *fill=True*, *text=''*, *loc=None*, *fontproperties=None*, *, *visible_edges='closed'*`)`
Bases: *matplotlib.patches.Rectangle*

A cell is a *Rectangle* with some associated *Text*.

As a user, you'll most likely not creates cells yourself. Instead, you should use either the *table* factory function or *Table.add_cell*.

Parameters

xy

[2-tuple] The position of the bottom left corner of the cell.

width

[float] The cell width.

height

[float] The cell height.

edgecolor

[color] The color of the cell border.

facecolor

[color] The cell facecolor.

fill

[bool] Whether the cell background is filled.

text

[str] The cell text.

loc

[{'left', 'center', 'right'}, default: 'right'] The alignment of the text within the cell.

fontproperties

[dict] A dict defining the font properties of the text. Supported keys and values are the keyword arguments accepted by *FontProperties*.

visible_edges

[str, default: 'closed'] The cell edges to be drawn with a line: a substring of 'BRTL' (bottom, right, top, left), or one of 'open' (no edges drawn), 'closed' (all edges drawn), 'horizontal' (bottom and top), 'vertical' (right and left).

`PAD = 0.1`
Padding between text and rectangle.

`auto_set_font_size`(*self*, *renderer*)
Shrink font size until the text fits into the cell width.

`draw`(*self*, *renderer*)
Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

Parameters

renderer

[*RendererBase* subclass.]

Notes

This method is overridden in the Artist subclasses.

`get_fontsize`(*self*)
Return the cell fontsize.

`get_path`(*self*)
Return a *Path* for the *visible_edges*.

`get_required_width`(*self*, *renderer*)
Return the minimal required width for the cell.

`get_text`(*self*)
Return the cell *Text* instance.

`get_text_bounds`(*self*, *renderer*)
Return the text bounds as *(x, y, width, height)* in table coordinates.

`set_figure`(*self*, *fig*)
Set the *Figure* instance the artist belongs to.

Parameters

fig

[*Figure*]

`set_fontsize`(*self*, *size*)
Set the text fontsize.

`set_text_props`(*self*, ***kwargs*)
Update the text properties.

Valid keyword arguments are:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array an
alpha	float or None
animated	bool
backgroundcolor	color
bbox	dict with properties for *patches.FancyBboxPatch*
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None

Tabl

Property	Description
color or c	color
contains	unknown
figure	*Figure*
fontfamily or family	{FONTNAME, 'serif', 'sans-serif', 'cursive', 'fantasy', '
fontproperties or font or font_properties	*font_manager.FontProperties* or `str` or `pathlib.Path`
fontsize or size	float or {'xx-small', 'x-small', 'small', 'medium', 'large',
fontstretch or stretch	{a numeric value in range 0-1000, 'ultra-condensed',
fontstyle or style	{'normal', 'italic', 'oblique'}
fontvariant or variant	{'normal', 'small-caps'}
fontweight or weight	{a numeric value in range 0-1000, 'ultralight', 'light',
gid	str
horizontalalignment or ha	{'center', 'right', 'left'}
in_layout	bool
label	object
linespacing	float (multiple of font size)
multialignment or ma	{'left', 'right', 'center'}
path_effects	*AbstractPathEffect*
picker	None or bool or callable
position	(float, float)
rasterized	bool or None
rotation	float or {'vertical', 'horizontal'}
rotation_mode	{None, 'default', 'anchor'}
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
text	object
transform	*Transform*
url	str
usetex	bool or None
verticalalignment or va	{'center', 'top', 'bottom', 'baseline', 'center_baseline'}
visible	bool
wrap	bool
x	float
y	float
zorder	float

`set_transform`(*self*, *trans*)

Set the artist transform.

Parameters

t

[*Transform*]

property visible_edges
The cell edges to be drawn with a line.

Reading this property returns a substring of 'BRTL' (bottom, right, top, left').

When setting this property, you can use a substring of 'BRTL' or one of {'open', 'closed', 'horizontal', 'vertical'}.

matplotlib.table.CustomCell
alias of *matplotlib.table.Cell*

class matplotlib.table.Table(*ax*, *loc=None*, *bbox=None*, ***kwargs*)
Bases: *matplotlib.artist.Artist*

A table of cells.

The table consists of a grid of cells, which are indexed by (row, column).

For a simple table, you'll have a full grid of cells with indices from (0, 0) to (num_rows-1, num_cols-1), in which the cell (0, 0) is positioned at the top left. However, you can also add cells with negative indices. You don't have to add a cell to every grid position, so you can create tables that have holes.

Note: You'll usually not create an empty table from scratch. Instead use *table* to create a table from data.

Parameters

ax

[*matplotlib.axes.Axes*] The *Axes* to plot the table into.

loc

[str] The position of the cell with respect to *ax*. This must be one of the *codes*.

bbox

[*Bbox* or None] A bounding box to draw the table into. If this is not *None*, this overrides *loc*.

Other Parameters

****kwargs**

Artist properties.

AXESPAD = 0.02
The border between the Axes and the table edge in Axes units.

FONTSIZE = 10

add_cell(*self*, *row*, *col*, **args*, ***kwargs*)
Create a cell and add it to the table.

Parameters

row

[int] Row index.

col

[int] Column index.

***args, ****kwargs**

All other parameters are passed on to *Cell*.

Returns

Cell

The created cell.

`auto_set_column_width`(*self*, *col*)
Automatically set the widths of given columns to optimal sizes.

Parameters

col

[int or sequence of ints] The indices of the columns to autoscale.

`auto_set_font_size`(*self*, *value=True*)
Automatically set font size.

`codes = {'best': 0, 'bottom': 17, 'bottom left': 12, 'bottom right': 13, 'center': 9, 'c`
Possible values where to place the table relative to the Axes.

`contains`(*self*, *mouseevent*)
Test whether the artist contains the mouse event.

Parameters

mouseevent

[*matplotlib.backend_bases.MouseEvent*]

Returns

contains

[bool] Whether any values are within the radius.

details

[dict] An artist-specific dictionary of details of the event context, such as which points are contained in the pick radius. See the individual Artist subclasses for details.

`draw`(*self*, *renderer*)
Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

Parameters

renderer

[*RendererBase* subclass.]

Notes

This method is overridden in the Artist subclasses.

`property edges`
The default value of *visible_edges* for newly added cells using *add_cell*.

Notes

This setting does currently only affect newly created cells using *add_cell*.

To change existing cells, you have to set their edges explicitly:

```
for c in tab.get_celld().values():
    c.visible_edges = 'horizontal'
```

`get_celld`(*self*)
Return a dict of cells in the table mapping *(row, column)* to *Cell*s.

Notes

You can also directly index into the Table object to access individual cells:

```
cell = table[row, col]
```

`get_children`(*self*)
Return the Artists contained by the table.

`get_window_extent`(*self*, *renderer*)
Return the bounding box of the table in window coords.

`scale`(*self*, *xscale*, *yscale*)
Scale column widths by *xscale* and row heights by *yscale*.

`set_fontsize`(*self*, *size*)
Set the font size, in points, of the cell text.

Parameters

size

[float]

Notes

As long as auto font size has not been disabled, the value will be clipped such that the text fits horizontally into the cell.

You can disable this behavior using *auto_set_font_size*.

```
>>> the_table.auto_set_font_size(False)
>>> the_table.set_fontsize(20)
```

However, there is no automatic scaling of the row height so that the text may exceed the cell boundary.

matplotlib.table.**table**(*ax*, *cellText=None*, *cellColours=None*, *cellLoc='right'*, *colWidths=None*, *rowLabels=None*, *rowColours=None*, *rowLoc='left'*, *colLabels=None*, *colColours=None*, *colLoc='center'*, *loc='bottom'*, *bbox=None*, *edges='closed'*, ***kwargs*)

Add a table to an *Axes*.

At least one of *cellText* or *cellColours* must be specified. These parameters must be 2D lists, in which the outer lists define the rows and the inner list define the column values per row. Each row must have the same number of elements.

The table can optionally have row and column headers, which are configured using *rowLabels*, *rowColours*, *rowLoc* and *colLabels*, *colColours*, *colLoc* respectively.

For finer grained control over tables, use the *Table* class and add it to the axes with *Axes.add_table*.

Parameters

cellText

[2D list of str, optional] The texts to place into the table cells.

Note: Line breaks in the strings are currently not accounted for and will result in the text exceeding the cell boundaries.

cellColours

[2D list of colors, optional] The background colors of the cells.

cellLoc

[{'left', 'center', 'right'}, default: 'right'] The alignment of the text within the cells.

colWidths

[list of float, optional] The column widths in units of the axes. If not given, all columns will have a width of *1 / ncols*.

rowLabels

[list of str, optional] The text of the row header cells.

rowColours

[list of colors, optional] The colors of the row header cells.

rowLoc

[{'left', 'center', 'right'}, default: 'left'] The text alignment of the row header cells.

colLabels

[list of str, optional] The text of the column header cells.

colColours

[list of colors, optional] The colors of the column header cells.

colLoc

[{'left', 'center', 'right'}, default: 'left'] The text alignment of the column header cells.

loc

[str, optional] The position of the cell with respect to *ax*. This must be one of the *codes*.

bbox

[*Bbox*, optional] A bounding box to draw the table into. If this is not *None*, this overrides *loc*.

edges

[substring of 'BRTL' or {'open', 'closed', 'horizontal', 'vertical'}] The cell edges to be drawn with a line. See also *visible_edges*.

Returns

Table

The created table.

Other Parameters

****kwargs**

Table properties.

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array
alpha	float or None
animated	bool
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
contains	unknown
figure	*Figure*
fontsize	float
gid	str
in_layout	bool
label	object
path_effects	*AbstractPathEffect*
picker	None or bool or callable
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
visible	bool
zorder	float

18.49 matplotlib.testing

18.49.1 matplotlib.testing

Helper functions for testing.

matplotlib.testing.**is_called_from_pytest**()

[*Deprecated*] Whether we are in a pytest run.

Notes

Deprecated since version 3.2.

matplotlib.testing.**set_font_settings_for_testing**()

matplotlib.testing.**set_reproducibility_for_testing**()

matplotlib.testing.**setup**()

18.49.2 matplotlib.testing.compare

Utilities for comparing image results.

matplotlib.testing.compare.**comparable_formats**()

Return the list of file formats that *compare_images* can compare on this system.

Returns

list of str

E.g. `['png', 'pdf', 'svg', 'eps']`.

matplotlib.testing.compare.**compare_images**(*expected*, *actual*, *tol*, *in_decorator=False*)

Compare two "image" files checking differences within a tolerance.

The two given filenames may point to files which are convertible to PNG via the `converter` dictionary. The underlying RMS is calculated with the `calculate_rms` function.

Parameters

expected

[str] The filename of the expected image.

actual

[str] The filename of the actual image.

tol

[float] The tolerance (a color value difference, where 255 is the maximal difference). The test fails if the average pixel difference is greater than this value.

in_decorator

[bool] Determines the output format. If called from image_comparison decorator, this should be True. (default=False)

Returns

None or dict or str

Return *None* if the images are equal within the given tolerance.

If the images differ, the return value depends on *in_decorator*. If *in_decorator* is true, a dict with the following entries is returned:

- *rms*: The RMS of the image difference.
- *expected*: The filename of the expected image.
- *actual*: The filename of the actual image.

- *diff_image*: The filename of the difference image.
- *tol*: The comparison tolerance.

Otherwise, a human-readable multi-line string representation of this information is returned.

Examples

```
img1 = "./baseline/plot.png"
img2 = "./output/plot.png"
compare_images(img1, img2, 0.001)
```

18.49.3 matplotlib.testing.decorators

class matplotlib.testing.decorators.**CleanupTestCase**(*methodName='runTest'*)

Bases: unittest.case.TestCase

A wrapper for unittest.TestCase that includes cleanup operations.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

classmethod setUpClass()

Hook method for setting up class fixture before running tests in the class.

classmethod tearDownClass()

Hook method for deconstructing the class fixture after running all tests in the class.

matplotlib.testing.decorators.**check_figures_equal**(**, extensions='png', 'pdf', 'svg', tol=0*)

Decorator for test cases that generate and compare two figures.

The decorated function must take two keyword arguments, *fig_test* and *fig_ref*, and draw the test and reference images on them. After the function returns, the figures are saved and compared.

This decorator should be preferred over *image_comparison* when possible in order to keep the size of the test suite from ballooning.

Parameters

extensions

[list, default: ["png", "pdf", "svg"]] The extensions to test.

tol

[float] The RMS threshold above which the test is considered failed.

Examples

Check that calling *Axes.plot* with a single argument plots it against `[0, 1, 2, ...]`:

```
@check_figures_equal()
def test_plot(fig_test, fig_ref):
    fig_test.subplots().plot([1, 3, 5])
    fig_ref.subplots().plot([0, 1, 2], [1, 3, 5])
```

matplotlib.testing.decorators.**check_freetype_version**(*ver*)

matplotlib.testing.decorators.**cleanup**(*style=None*)

A decorator to ensure that any global state is reset before running a test.

Parameters

style

[str, dict, or list, optional] The style(s) to apply. Defaults to `["classic", "_classic_test_patch"]`.

matplotlib.testing.decorators.**image_comparison**(*baseline_images*, *extensions=None*, *tol=0*, *freetype_version=None*, *remove_text=False*, *savefig_kwarg=None*, *style='classic'*, *'_classic_test_patch'*)

Compare images generated by the test with those specified in *baseline_images*, which must correspond, else an `ImageComparisonFailure` exception will be raised.

Parameters

baseline_images

[list or None] A list of strings specifying the names of the images generated by calls to *Figure.savefig*.

If *None*, the test function must use the `baseline_images` fixture, either as a parameter or with `pytest.mark.usefixtures`. This value is only allowed when using pytest.

extensions

[None or list of str] The list of extensions to test, e.g. `['png', 'pdf']`.

If *None*, defaults to all supported extensions: png, pdf, and svg.

When testing a single extension, it can be directly included in the names passed to *baseline_images*. In that case, *extensions* must not be set.

In order to keep the size of the test suite from ballooning, we only include the `svg` or `pdf` outputs if the test is explicitly exercising a feature dependent on that backend (see also the *check_figures_equal* decorator for that purpose).

tol

[float, default: 0] The RMS threshold above which the test is considered failed.

Due to expected small differences in floating-point calculations, on 32-bit systems an additional 0.06 is added to this threshold.

freetype_version

[str or tuple] The expected freetype version or range of versions for this test to pass.

remove_text

[bool] Remove the title and tick text from the figure before comparison. This is useful to make the baseline images independent of variations in text rendering between different versions of FreeType.

This does not remove other, more deliberate, text, such as legends and annotations.

savefig_kwarg

[dict] Optional arguments that are passed to the savefig method.

style

[str, dict, or list] The optional style(s) to apply to the image test. The test itself can also apply additional styles if desired. Defaults to `["classic", "_classic_test_patch"]`.

matplotlib.testing.decorators.**remove_ticks_and_titles**(*figure*)

18.49.4 matplotlib.testing.exceptions

exception matplotlib.testing.exceptions.**ImageComparisonFailure**

Bases: AssertionError

Raise this exception to mark a test as a comparison between two images.

18.50 matplotlib.text

Classes for including text in a figure.

class matplotlib.text.**Annotation**(*text*, *xy*, *xytext=None*, *xycoords='data'*, *textcoords=None*, *arrowprops=None*, *annotation_clip=None*, ***kwargs*)

Bases: *matplotlib.text.Text*, matplotlib.text._AnnotationBase

An *Annotation* is a *Text* that can refer to a specific position *xy*. Optionally an arrow pointing from the text to *xy* can be drawn.

Attributes

xy

The annotated position.

xycoords

The coordinate system for *xy*.

arrow_patch

A *FancyArrowPatch* to point from *xytext* to *xy*.

Annotate the point *xy* with text *text*.

In the simplest form, the text is placed at *xy*.

Optionally, the text can be displayed in another position *xytext*. An arrow pointing from the text to the annotated point *xy* can then be added by defining *arrowprops*.

Parameters

text

[str] The text of the annotation. *s* is a deprecated synonym for this parameter.

xy

[(float, float)] The point *(x, y)* to annotate. The coordinate system is determined by *xycoords*.

xytext

[(float, float), default: *xy*] The position *(x, y)* to place the text at. The coordinate system is determined by *textcoords*.

xycoords

[str or *Artist* or *Transform* or callable or (float, float), default: 'data'] The coordinate system that *xy* is given in. The following types of values are supported:

- One of the following strings:

Value	Description
'figure points'	Points from the lower left of the figure
'figure pixels'	Pixels from the lower left of the figure
'figure fraction'	Fraction of figure from lower left
'axes points'	Points from lower left corner of axes
'axes pixels'	Pixels from lower left corner of axes
'axes fraction'	Fraction of axes from lower left
'data'	Use the coordinate system of the object being annotated (default)
'polar'	*(theta, r)* if not native 'data' coordinates

- An *Artist*: *xy* is interpreted as a fraction of the artist's *Bbox*. E.g. *(0, 0)* would be the lower left corner of the bounding box and *(0.5, 1)* would be the center top of the bounding box.
- A *Transform* to transform *xy* to screen coordinates.
- A function with one of the following signatures:

```
def transform(renderer) -> Bbox
def transform(renderer) -> Transform
```

  where *renderer* is a *RendererBase* subclass.

  The result of the function is interpreted like the *Artist* and *Transform* cases above.

- A tuple *(xcoords, ycoords)* specifying separate coordinate systems for *x* and *y*. *xcoords* and *ycoords* must each be of one of the above described types.

See *Advanced Annotations* for more details.

textcoords

[str or *Artist* or *Transform* or callable or (float, float), default: value of *xycoords*] The coordinate system that *xytext* is given in.

All *xycoords* values are valid as well as the following strings:

Value	Description
'offset points'	Offset (in points) from the *xy* value
'offset pixels'	Offset (in pixels) from the *xy* value

arrowprops

[dict, optional] The properties used to draw a *FancyArrowPatch* arrow between the positions *xy* and *xytext*.

If *arrowprops* does not contain the key 'arrowstyle' the allowed keys are:

Key	Description
width	The width of the arrow in points
headwidth	The width of the base of the arrow head in points
headlength	The length of the arrow head in points
shrink	Fraction of total length to shrink from both ends
?	Any key to *matplotlib.patches.FancyArrowPatch*

If *arrowprops* contains the key 'arrowstyle' the above keys are forbidden. The allowed values of `'arrowstyle'` are:

Name	Attrs
`'-'`	None
`'->'`	head_length=0.4,head_width=0.2
`'-['`	widthB=1.0,lengthB=0.2,angleB=None
`'|-|'`	widthA=1.0,widthB=1.0
`'-|>'`	head_length=0.4,head_width=0.2
`'<-'`	head_length=0.4,head_width=0.2
`'<->'`	head_length=0.4,head_width=0.2
`'<|-'`	head_length=0.4,head_width=0.2
`'<|-|>'`	head_length=0.4,head_width=0.2
`'fancy'`	head_length=0.4,head_width=0.4,tail_width=0.4
`'simple'`	head_length=0.5,head_width=0.5,tail_width=0.2
`'wedge'`	tail_width=0.3,shrink_factor=0.5

Valid keys for *FancyArrowPatch* are:

Key	Description
arrowstyle	the arrow style
connectionstyle	the connection style
relpos	default is (0.5, 0.5)
patchA	default is bounding box of the text
patchB	default is None
shrinkA	default is 2 points
shrinkB	default is 2 points
mutation_scale	default is text size (in points)
mutation_aspect	default is 1.
?	any key for *matplotlib.patches.PathPatch*

Defaults to None, i.e. no arrow is drawn.

annotation_clip

[bool or None, default: None] Whether to draw the annotation when the annotation point *xy* is outside the axes area.

- If *True*, the annotation will only be drawn when *xy* is within the axes.
- If *False*, the annotation will always be drawn.
- If *None*, the annotation will only be drawn when *xy* is within the axes and *xycoords* is 'data'.

****kwargs**

Additional kwargs are passed to *Text*.

Returns

Annotation

See also:

Advanced Annotations

`property anncoords`
The coordinate system to use for *Annotation.xyann*.

`contains`(*self*, *event*)
Return whether the mouse event occurred inside the axis-aligned bounding-box of the text.

`draw`(*self*, *renderer*)
Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

Parameters

renderer

[*RendererBase* subclass.]

Notes

This method is overridden in the Artist subclasses.

`get_anncoords`(*self*)
Return the coordinate system to use for *Annotation.xyann*.

See also *xycoords* in *Annotation*.

`get_window_extent`(*self*, *renderer=None*)
Return the *Bbox* bounding the text and arrow, in display units.

Parameters

renderer

[Renderer, optional] A renderer is needed to compute the bounding box. If the artist has already been drawn, the renderer is cached; thus, it is only necessary to pass this argument when calling *get_window_extent* before the first *draw*. In practice, it is usually easier to trigger a draw first (e.g. by saving the figure).

`set_anncoords`(*self*, *coords*)
Set the coordinate system to use for *Annotation.xyann*.

See also *xycoords* in *Annotation*.

`set_figure`(*self*, *fig*)
Set the *Figure* instance the artist belongs to.

Parameters

fig

[*Figure*]

`update_positions`(*self*, *renderer*)
Update the pixel positions of the annotation text and the arrow patch.

`property xyann`
The text position.

See also *xytext* in *Annotation*.

`property xycoords`

`class matplotlib.text.OffsetFrom`(*artist*, *ref_coord*, *unit='points'*)
Bases: `object`

Callable helper class for working with *Annotation*.

Parameters

artist

[*Artist* or *BboxBase* or *Transform*] The object to compute the offset from.

ref_coord

[(float, float)] If *artist* is an *Artist* or *BboxBase*, this values is the location to of the offset origin in fractions of the *artist* bounding box.

If *artist* is a transform, the offset origin is the transform applied to this value.

unit

[{'points, 'pixels'}, default: 'points'] The screen units to use (pixels or points) for the offset input.

`get_unit`(*self*)
Return the unit for input to the transform used by `__call__`.

`set_unit`(*self*, *unit*)
Set the unit for input to the transform used by `__call__`.

Parameters

unit

[{'points', 'pixels'}]

`class matplotlib.text.Text`(*x=0*, *y=0*, *text=''*, *color=None*, *verticalalignment='baseline'*, *horizontalalignment='left'*, *multialignment=None*, *fontproperties=None*, *rotation=None*, *linespacing=None*, *rotation_mode=None*, *usetex=None*, *wrap=False*, ***kwargs*)

Bases: *matplotlib.artist.Artist*

Handle storing and drawing of text in window or data coordinates.

Create a *Text* instance at *x*, *y* with string *text*.

Valid keyword arguments are:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array an
alpha	float or None
animated	bool
backgroundcolor	color
bbox	dict with properties for *patches.FancyBboxPatch*
clip_box	*Bbox*

Tabl

Property	Description
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color or c	color
contains	unknown
figure	*Figure*
fontfamily or family	{FONTNAME, 'serif', 'sans-serif', 'cursive', 'fantasy', '
fontproperties or font or font_properties	*font_manager.FontProperties* or `str` or `pathlib.Path`
fontsize or size	float or {'xx-small', 'x-small', 'small', 'medium', 'large',
fontstretch or stretch	{a numeric value in range 0-1000, 'ultra-condensed',
fontstyle or style	{'normal', 'italic', 'oblique'}
fontvariant or variant	{'normal', 'small-caps'}
fontweight or weight	{a numeric value in range 0-1000, 'ultralight', 'light',
gid	str
horizontalalignment or ha	{'center', 'right', 'left'}
in_layout	bool
label	object
linespacing	float (multiple of font size)
multialignment or ma	{'left', 'right', 'center'}
path_effects	*AbstractPathEffect*
picker	None or bool or callable
position	(float, float)
rasterized	bool or None
rotation	float or {'vertical', 'horizontal'}
rotation_mode	{None, 'default', 'anchor'}
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
text	object
transform	*Transform*
url	str
usetex	bool or None
verticalalignment or va	{'center', 'top', 'bottom', 'baseline', 'center_baseline'}
visible	bool
wrap	bool
x	float
y	float
zorder	float

`contains`(*self*, *mouseevent*)
: Return whether the mouse event occurred inside the axis-aligned bounding-box of the text.

`draw`(*self*, *renderer*)
: Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

Parameters

renderer

[*RendererBase* subclass.]

Notes

This method is overridden in the Artist subclasses.

`get_bbox_patch`(*self*)
Return the bbox Patch, or None if the *patches.FancyBboxPatch* is not made.

`get_c`(*self*)
Alias for *get_color*.

`get_color`(*self*)
Return the color of the text.

`get_family`(*self*)
Alias for *get_fontfamily*.

`get_font`(*self*)
Alias for *get_fontproperties*.

`get_font_properties`(*self*)
Alias for *get_fontproperties*.

`get_fontfamily`(*self*)
Return the list of font families used for font lookup.

See also:

font_manager.FontProperties.get_family

`get_fontname`(*self*)
Return the font name as a string.

See also:

font_manager.FontProperties.get_name

`get_fontproperties`(*self*)
Return the *font_manager.FontProperties*.

`get_fontsize`(*self*)
Return the font size as an integer.

See also:

font_manager.FontProperties.get_size_in_points

get_fontstyle(*self*)
Return the font style as a string.

See also:

font_manager.FontProperties.get_style

get_fontvariant(*self*)
Return the font variant as a string.

See also:

font_manager.FontProperties.get_variant

get_fontweight(*self*)
Return the font weight as a string or a number.

See also:

font_manager.FontProperties.get_weight

get_ha(*self*)
Alias for *get_horizontalalignment*.

get_horizontalalignment(*self*)
Return the horizontal alignment as a string. Will be one of 'left', 'center' or 'right'.

get_name(*self*)
Alias for *get_fontname*.

get_position(*self*)
Return the (x, y) position of the text.

get_prop_tup(*self*, *renderer=None*)
Return a hashable tuple of properties.

Not intended to be human readable, but useful for backends who want to cache derived information about text (e.g., layouts) and need to know if the text has changed.

get_rotation(*self*)
Return the text angle in degrees between 0 and 360.

get_rotation_mode(*self*)
Return the text rotation mode.

get_size(*self*)
Alias for *get_fontsize*.

get_stretch(*self*)
Return the font stretch as a string or a number.

See also:

font_manager.FontProperties.get_stretch

get_style(*self*)
Alias for *get_fontstyle*.

get_text(*self*)
Return the text string.

get_unitless_position(*self*)
Return the (x, y) unitless position of the text.

get_usetex(*self*)
Return whether this *Text* object uses TeX for rendering.

get_va(*self*)
Alias for *get_verticalalignment*.

get_variant(*self*)
Alias for *get_fontvariant*.

get_verticalalignment(*self*)
Return the vertical alignment as a string. Will be one of 'top', 'center', 'bottom' or 'baseline'.

get_weight(*self*)
Alias for *get_fontweight*.

get_window_extent(*self*, *renderer=None*, *dpi=None*)
Return the *Bbox* bounding the text, in display units.

In addition to being used internally, this is useful for specifying clickable regions in a png file on a web page.

Parameters

renderer

[Renderer, optional] A renderer is needed to compute the bounding box. If the artist has already been drawn, the renderer is cached; thus, it is only necessary to pass this argument when calling *get_window_extent* before the first *draw*. In practice, it is usually easier to trigger a draw first (e.g. by saving the figure).

dpi

[float, optional] The dpi value for computing the bbox, defaults to `self.figure.dpi` (*not* the renderer dpi); should be set e.g. if to match regions with a figure saved with a custom dpi value.

`get_wrap`(*self*)
Return whether the text can be wrapped.

`set_backgroundcolor`(*self*, *color*)
Set the background color of the text by updating the bbox.

Parameters

color

[color]

See also:

set_bbox

To change the position of the bounding box

`set_bbox`(*self*, *rectprops*)
Draw a bounding box around self.

Parameters

rectprops

[dict with properties for *patches.FancyBboxPatch*] The default boxstyle is 'square'. The mutation scale of the *patches.FancyBboxPatch* is set to the fontsize.

Examples

```
t.set_bbox(dict(facecolor='red', alpha=0.5))
```

`set_c`(*self*, *color*)
Alias for *set_color*.

`set_clip_box`(*self*, *clipbox*)
Set the artist's clip *Bbox*.

Parameters

clipbox

[*Bbox*]

`set_clip_on`(*self*, *b*)
Set whether the artist uses clipping.

When False artists will be visible outside of the axes which can lead to unexpected results.

Parameters

b

[bool]

set_clip_path(*self*, *path*, *transform=None*)
Set the artist's clip path.

Parameters

path

[*Patch* or *Path* or *TransformedPath* or None] The clip path. If given a *Path*, *transform* must be provided as well. If *None*, a previously set clip path is removed.

transform

[*Transform*, optional] Only used if *path* is a *Path*, in which case the given *Path* is converted to a *TransformedPath* using *transform*.

Notes

For efficiency, if *path* is a *Rectangle* this method will set the clipping box to the corresponding rectangle and set the clipping path to `None`.

For technical reasons (support of *set*), a tuple (*path*, *transform*) is also accepted as a single positional parameter.

set_color(*self*, *color*)
Set the foreground color of the text

Parameters

color

[color]

set_family(*self*, *fontname*)
Alias for *set_fontfamily*.

set_font(*self*, *fp*)
Alias for *set_fontproperties*.

set_font_properties(*self*, *fp*)
Alias for *set_fontproperties*.

set_fontfamily(*self*, *fontname*)
Set the font family. May be either a single string, or a list of strings in decreasing priority. Each string may be either a real font name or a generic font class name. If the latter, the specific font names will be looked up in the corresponding rcParams.

If a *Text* instance is constructed with `fontfamily=None`, then the font is set to `rcParams["font.family"]` (default: `['sans-serif']`), and the same is done when *set_fontfamily()* is called on an existing *Text* instance.

Parameters

fontname

[{FONTNAME, 'serif', 'sans-serif', 'cursive', 'fantasy', 'monospace'}]

See also:

font_manager.FontProperties.set_family

`set_fontname`(*self*, *fontname*)
Alias for *set_family*.

One-way alias only: the getter differs.

Parameters

fontname

[{FONTNAME, 'serif', 'sans-serif', 'cursive', 'fantasy', 'monospace'}]

See also:

font_manager.FontProperties.set_family

`set_fontproperties`(*self*, *fp*)
Set the font properties that control the text.

Parameters

fp

[*font_manager.FontProperties* or `str` or `pathlib.Path`] If a `str`, it is interpreted as a fontconfig pattern parsed by *FontProperties*. If a `pathlib.Path`, it is interpreted as the absolute path to a font file.

`set_fontsize`(*self*, *fontsize*)
Set the font size.

Parameters

fontsize

[float or {'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large'}] If float, the fontsize in points. The string values denote sizes relative to the default font size.

See also:

font_manager.FontProperties.set_size

`set_fontstretch`(*self*, *stretch*)
Set the font stretch (horizontal condensation or expansion).

Parameters

stretch

[{a numeric value in range 0-1000, 'ultra-condensed', 'extra-condensed', 'condensed', 'semi-condensed', 'normal', 'semi-expanded', 'expanded', 'extra-expanded', 'ultra-expanded'}]

See also:

font_manager.FontProperties.set_stretch

`set_fontstyle`(*self*, *fontstyle*)
Set the font style.

Parameters

fontstyle

[{'normal', 'italic', 'oblique'}]

See also:

font_manager.FontProperties.set_style

`set_fontvariant`(*self*, *variant*)
Set the font variant.

Parameters

variant

[{'normal', 'small-caps'}]

See also:

font_manager.FontProperties.set_variant

`set_fontweight`(*self*, *weight*)
Set the font weight.

Parameters

weight

[{a numeric value in range 0-1000, 'ultralight', 'light', 'normal', 'regular', 'book', 'medium', 'roman', 'semibold', 'demibold', 'demi', 'bold', 'heavy', 'extra bold', 'black'}]

See also:

font_manager.FontProperties.set_weight

`set_ha`(*self*, *align*)
Alias for *set_horizontalalignment*.

`set_horizontalalignment`(*self*, *align*)
Set the horizontal alignment to one of

Parameters

align

[{'center', 'right', 'left'}]

`set_linespacing`(*self*, *spacing*)
Set the line spacing as a multiple of the font size.

The default line spacing is 1.2.

Parameters

spacing

[float (multiple of font size)]

`set_ma`(*self*, *align*)
Alias for *set_multialignment*.

`set_multialignment`(*self*, *align*)
Set the text alignment for multiline texts.

The layout of the bounding box of all the lines is determined by the horizontalalignment and verticalalignment properties. This property controls the alignment of the text lines within that box.

Parameters

align

[{'left', 'right', 'center'}]

`set_name`(*self*, *fontname*)
Alias for *set_fontname*.

set_position(*self*, *xy*)
Set the (*x*, *y*) position of the text.

Parameters

xy

[(float, float)]

set_rotation(*self*, *s*)
Set the rotation of the text.

Parameters

s

[float or {'vertical', 'horizontal'}] The rotation angle in degrees in mathematically positive direction (counterclockwise). 'horizontal' equals 0, 'vertical' equals 90.

set_rotation_mode(*self*, *m*)
Set text rotation mode.

Parameters

m

[{None, 'default', 'anchor'}] If `None` or `"default"`, the text will be first rotated, then aligned according to their horizontal and vertical alignments. If `"anchor"`, then alignment occurs before rotation.

set_size(*self*, *fontsize*)
Alias for *set_fontsize*.

set_stretch(*self*, *stretch*)
Alias for *set_fontstretch*.

set_style(*self*, *fontstyle*)
Alias for *set_fontstyle*.

set_text(*self*, *s*)
Set the text string *s*.

It may contain newlines (`\n`) or math in LaTeX syntax.

Parameters

s

[object] Any object gets converted to its `str` representation, except for `None` which is converted to an empty string.

set_usetex(*self*, *usetex*)

Parameters

usetex

[bool or None] Whether to render using TeX, `None` means to use `rcParams["text.usetex"]` (default: `False`).

`set_va`(*self*, *align*)
Alias for *set_verticalalignment*.

`set_variant`(*self*, *variant*)
Alias for *set_fontvariant*.

`set_verticalalignment`(*self*, *align*)
Set the vertical alignment.

Parameters

align

[{'center', 'top', 'bottom', 'baseline', 'center_baseline'}]

`set_weight`(*self*, *weight*)
Alias for *set_fontweight*.

`set_wrap`(*self*, *wrap*)
Set whether the text can be wrapped.

Parameters

wrap

[bool]

`set_x`(*self*, *x*)
Set the *x* position of the text.

Parameters

x

[float]

`set_y`(*self*, *y*)
Set the *y* position of the text.

Parameters

y

[float]

`update`(*self*, *kwargs*)
Update this artist's properties from the dict *props*.

Parameters

props

[dict]

`update_bbox_position_size`(*self*, *renderer*)
Update the location and the size of the bbox.

This method should be used when the position and size of the bbox needs to be updated before actually drawing the bbox.

`update_from`(*self*, *other*)
Copy properties from *other* to *self*.

`zorder = 3`

`matplotlib.text.get_rotation`(*rotation*)
Return *rotation* normalized to an angle between 0 and 360 degrees.

Parameters

rotation

[float or {None, 'horizontal', 'vertical'}] Rotation angle in degrees. *None* and 'horizontal' equal 0, 'vertical' equals 90.

Returns

float

18.51 matplotlib.texmanager

Support for embedded TeX expressions in Matplotlib via dvipng and dvips for the raster and PostScript backends. The tex and dvipng/dvips information is cached in ~/.matplotlib/tex.cache for reuse between sessions.

Requirements:

- latex
- *Agg backends: dvipng>=1.6
- PS backend: psfrag, dvips, and Ghostscript>=8.60

Backends:

- *Agg
- PS
- PDF

For raster output, you can get RGBA numpy arrays from TeX expressions as follows:

```
texmanager = TexManager()
s = ('\TeX\ is Number '
     '$\displaystyle\sum_{n=1}^\infty\frac{-e^{i\pi}}{2^n}$!')
Z = texmanager.get_rgba(s, fontsize=12, dpi=80, rgb=(1, 0, 0))
```

To enable tex rendering of all text in your matplotlib figure, set `rcParams["text.usetex"]` (default: `False`) to True.

class matplotlib.texmanager.**TexManager**

Bases: `object`

Convert strings to dvi files using TeX, caching the results to a directory.

Repeated calls to this constructor always return the same instance.

property cachedir

property cursive

font_families = ('serif', 'sans-serif', 'cursive', 'monospace')

font_family = 'serif'

font_info = {'avant garde': ('pag', '\\usepackage{avant}'), 'bookman': ('pbk', '\\renewco

get_basefile(*self*, *tex*, *fontsize*, *dpi=None*)
Return a filename based on a hash of the string, fontsize, and dpi.

get_custom_preamble(*self*)
Return a string containing user additions to the tex preamble.

get_font_config(*self*)

get_font_preamble(*self*)
Return a string containing font configuration for the tex preamble.

get_grey(*self*, *tex*, *fontsize=None*, *dpi=None*)
Return the alpha channel.

get_rgba(*self*, *tex*, *fontsize=None*, *dpi=None*, *rgb=0, 0, 0*)
Return latex's rendering of the tex string as an rgba array.

get_text_width_height_descent(*self*, *tex*, *fontsize*, *renderer=None*)
Return width, height and descent of the text.

grey_arrayd = {('$+1$', 'sans-serifcmrcmsspzccmttd41d8cd98f00b204e9800998ecf8427e', 20.0,

make_dvi(*self*, *tex*, *fontsize*)
Generate a dvi file containing latex's layout of tex string.

Return the file name.

make_dvi_preview(*self*, *tex*, *fontsize*)
[*Deprecated*] Generate a dvi file containing latex's layout of tex string.

It calls make_tex_preview() method and store the size information (width, height, descent) in a separate file.

Return the file name.

Notes

Deprecated since version 3.3.

`make_png`(*self*, *tex*, *fontsize*, *dpi*)
Generate a png file containing latex's rendering of tex string.

Return the file name.

`make_tex`(*self*, *tex*, *fontsize*)
Generate a tex file to render the tex string at a specific font size.

Return the file name.

`make_tex_preview`(*self*, *tex*, *fontsize*)
[*Deprecated*] Generate a tex file to render the tex string at a specific font size.

It uses the preview.sty to determine the dimension (width, height, descent) of the output.

Return the file name.

Notes

Deprecated since version 3.3.

`property monospace`

`property rgba_arrayd`

`property sans_serif`

`property serif`

`texcache = '/root/.cache/matplotlib/tex.cache'`

18.52 matplotlib.textpath

`class matplotlib.textpath.TextPath`(*xy*, *s*, *size=None*, *prop=None*, *_interpolation_steps=1*, *usetex=False*)
Bases: *matplotlib.path.Path*

Create a path from the text.

Create a path from the text. Note that it simply is a path, not an artist. You need to use the *PathPatch* (or other artists) to draw this path onto the canvas.

Parameters

xy

[tuple or array of two float values] Position of the text. For no offset, use `xy=(0, 0)`.

s

[str] The text to convert to a path.

size

[float, optional] Font size in points. Defaults to the size specified via the font properties *prop*.

prop

[*matplotlib.font_manager.FontProperties*, optional] Font property. If not provided, will use a default `FontProperties` with parameters from the *rcParams*.

_interpolation_steps

[int, optional] (Currently ignored)

usetex

[bool, default: False] Whether to use tex rendering.

Examples

The following creates a path from the string "ABC" with Helvetica font face; and another path from the latex fraction 1/2:

```
from matplotlib.textpath import TextPath
from matplotlib.font_manager import FontProperties

fp = FontProperties(family="Helvetica", style="italic")
path1 = TextPath((12, 12), "ABC", size=12, prop=fp)
path2 = TextPath((0, 0), r"$\frac{1}{2}$", size=12, usetex=True)
```

Also see /gallery/text_labels_and_annotations/demo_text_path.

`property codes`
Return the codes

`get_size`(*self*)
Get the text size.

`set_size`(*self*, *size*)
Set the text size.

`property vertices`
Return the cached path after updating it if necessary.

class matplotlib.textpath.TextToPath

Bases: object

A class that converts strings to paths.

DPI = 72

FONT_SCALE = 100.0

get_glyphs_mathtext(*self*, *prop*, *s*, *glyph_map=None*, *return_new_glyphs_only=False*)

Parse mathtext string *s* and convert it to a (vertices, codes) pair.

get_glyphs_tex(*self*, *prop*, *s*, *glyph_map=None*, *return_new_glyphs_only=False*)

Convert the string *s* to vertices and codes using usetex mode.

get_glyphs_with_font(*self*, *font*, *s*, *glyph_map=None*, *return_new_glyphs_only=False*)

Convert string *s* to vertices and codes using the provided ttf font.

get_texmanager(*self*)

Return the cached *TexManager* instance.

get_text_path(*self*, *prop*, *s*, *ismath=False*)

Convert text *s* to path (a tuple of vertices and codes for matplotlib.path.Path).

Parameters

prop

[*FontProperties*] The font properties for the text.

s

[str] The text to be converted.

ismath

[{False, True, "TeX"}] If True, use mathtext parser. If "TeX", use tex for rendering.

Returns

verts

[list] A list of numpy arrays containing the x and y coordinates of the vertices.

codes

[list] A list of path codes.

Examples

Create a list of vertices and codes from a text, and create a *Path* from those:

```
from matplotlib.path import Path
from matplotlib.textpath import TextToPath
from matplotlib.font_manager import FontProperties

fp = FontProperties(family="Humor Sans", style="italic")
verts, codes = TextToPath().get_text_path(fp, "ABC")
path = Path(verts, codes, closed=False)
```

Also see *TextPath* for a more direct way to create a path from a text.

get_text_width_height_descent(*self*, *s*, *prop*, *ismath*)

18.53 matplotlib.ticker

18.53.1 Tick locating and formatting

This module contains classes for configuring tick locating and formatting. Generic tick locators and formatters are provided, as well as domain specific custom ones.

Although the locators know nothing about major or minor ticks, they are used by the Axis class to support major and minor tick locating and formatting.

Tick locating

The Locator class is the base class for all tick locators. The locators handle autoscaling of the view limits based on the data limits, and the choosing of tick locations. A useful semi-automatic tick locator is *MultipleLocator*. It is initialized with a base, e.g., 10, and it picks axis limits and ticks that are multiples of that base.

The Locator subclasses defined here are

AutoLocator

MaxNLocator with simple defaults. This is the default tick locator for most plotting.

MaxNLocator

Finds up to a max number of intervals with ticks at nice locations.

LinearLocator

Space ticks evenly from min to max.

LogLocator

Space ticks logarithmically from min to max.

MultipleLocator

Ticks and range are a multiple of base; either integer or float.

FixedLocator

Tick locations are fixed.

IndexLocator

Locator for index plots (e.g., where `x = range(len(y))`).

NullLocator

No ticks.

SymmetricalLogLocator

Locator for use with with the symlog norm; works like *LogLocator* for the part outside of the threshold and adds 0 if inside the limits.

LogitLocator

Locator for logit scaling.

OldAutoLocator

Choose a *MultipleLocator* and dynamically reassign it for intelligent ticking during navigation.

AutoMinorLocator

Locator for minor ticks when the axis is linear and the major ticks are uniformly spaced. Subdivides the major tick interval into a specified number of minor intervals, defaulting to 4 or 5 depending on the major interval.

There are a number of locators specialized for date locations - see the *dates* module.

You can define your own locator by deriving from Locator. You must override the `__call__` method, which returns a sequence of locations, and you will probably want to override the autoscale method to set the view limits from the data limits.

If you want to override the default locator, use one of the above or a custom locator and pass it to the x or y axis instance. The relevant methods are:

```
ax.xaxis.set_major_locator(xmajor_locator)
ax.xaxis.set_minor_locator(xminor_locator)
ax.yaxis.set_major_locator(ymajor_locator)
ax.yaxis.set_minor_locator(yminor_locator)
```

The default minor locator is *NullLocator*, i.e., no minor ticks on by default.

Tick formatting

Tick formatting is controlled by classes derived from Formatter. The formatter operates on a single tick value and returns a string to the axis.

NullFormatter

No labels on the ticks.

IndexFormatter

Set the strings from a list of labels.

FixedFormatter

Set the strings manually for the labels.

FuncFormatter

User defined function sets the labels.

StrMethodFormatter

Use string `format` method.

FormatStrFormatter

Use an old-style sprintf format string.

ScalarFormatter

Default formatter for scalars: autopick the format string.

LogFormatter

Formatter for log axes.

LogFormatterExponent

Format values for log axis using `exponent = log_base(value)`.

LogFormatterMathtext

Format values for log axis using `exponent = log_base(value)` using Math text.

LogFormatterSciNotation

Format values for log axis using scientific notation.

LogitFormatter

Probability formatter.

EngFormatter

Format labels in engineering notation.

PercentFormatter

Format labels as a percentage.

You can derive your own formatter from the Formatter base class by simply overriding the `__call__` method. The formatter class has access to the axis view and data limits.

To control the major and minor tick label formats, use one of the following methods:

```
ax.xaxis.set_major_formatter(xmajor_formatter)
ax.xaxis.set_minor_formatter(xminor_formatter)
ax.yaxis.set_major_formatter(ymajor_formatter)
ax.yaxis.set_minor_formatter(yminor_formatter)
```

In addition to a *Formatter* instance, *set_major_formatter* and *set_minor_formatter* also accept a `str` or function. `str` input will be internally replaced with an autogenerated *StrMethodFormatter* with the input `str`. For function input, a *FuncFormatter* with the input function will be generated and used.

See /gallery/ticks_and_spines/major_minor_demo for an example of setting major and minor ticks. See the *matplotlib.dates* module for more information and examples of using date locators and formatters.

class matplotlib.ticker.**AutoLocator**

Bases: *matplotlib.ticker.MaxNLocator*

Dynamically find major tick positions. This is actually a subclass of *MaxNLocator*, with parameters *nbins* = '*auto*' and *steps* = *[1, 2, 2.5, 5, 10]*.

To know the values of the non-public parameters, please have a look to the defaults of *MaxNLocator*.

class matplotlib.ticker.**AutoMinorLocator**(*n=None*)

Bases: *matplotlib.ticker.Locator*

Dynamically find minor tick positions based on the positions of major ticks. The scale must be linear with major ticks evenly spaced.

n is the number of subdivisions of the interval between major ticks; e.g., n=2 will place a single minor tick midway between major ticks.

If *n* is omitted or None, it will be set to 5 or 4.

tick_values(*self*, *vmin*, *vmax*)

Return the values of the located ticks given **vmin** and **vmax**.

Note: To get tick locations with the vmin and vmax values defined automatically for the associated `axis` simply call the Locator instance:

```
>>> print(type(loc))
<type 'Locator'>
>>> print(loc())
[1, 2, 3, 4]
```

---

class matplotlib.ticker.**EngFormatter**(*unit=''*, *places=None*, *sep=' '*, ***, *usetex=None*, *useMathText=None*)

Bases: *matplotlib.ticker.Formatter*

Format axis values using engineering prefixes to represent powers of 1000, plus a specified unit, e.g., 10 MHz instead of 1e7.

Parameters

unit

[str, default: ""] Unit symbol to use, suitable for use with single-letter representations of powers of 1000. For example, 'Hz' or 'm'.

places

[int, default: None] Precision with which to display the number, specified in digits after the decimal point (there will be between one and three digits before the decimal point). If it is None, the formatting falls back to the floating point format '%g', which displays up to 6 *significant* digits, i.e. the equivalent value for *places* varies between 0 and 5 (inclusive).

sep

[str, default: " "] Separator used between the value and the prefix/unit. For example, one get '3.14 mV' if `sep` is " " (default) and '3.14mV' if `sep` is "". Besides the default behavior, some other useful options may be:

- `sep=""` to append directly the prefix/unit to the value;
- `sep="\N{THIN SPACE}"` (`U+2009`);
- `sep="\N{NARROW NO-BREAK SPACE}"` (`U+202F`);
- `sep="\N{NO-BREAK SPACE}"` (`U+00A0`).

usetex

[bool, default: `rcParams["text.usetex"]` (default: `False`)] To enable/disable the use of TeX's math mode for rendering the numbers in the formatter.

useMathText

[bool, default: `rcParams["axes.formatter.use_mathtext"]` (default: `False`)] To enable/disable the use mathtext for rendering the numbers in the formatter.

`ENG_PREFIXES = {-24: 'y', -21: 'z', -18: 'a', -15: 'f', -12: 'p', -9: 'n', -6: 'µ', -3:`

`format_eng`(*self*, *num*)

Format a number in engineering notation, appending a letter representing the power of 1000 of the original number. Some examples:

```
>>> format_eng(0)       # for self.places = 0
'0'
```

```
>>> format_eng(1000000) # for self.places = 1
'1.0 M'
```

```
>>> format_eng("-1e-6") # for self.places = 2
'-1.00 µ'
```

get_useMathText(*self*)

get_usetex(*self*)

set_useMathText(*self*, *val*)

set_usetex(*self*, *val*)

property useMathText

property usetex

class matplotlib.ticker.**FixedFormatter**(*seq*)

Bases: *matplotlib.ticker.Formatter*

Return fixed strings for tick labels based only on position, not value.

Note: *FixedFormatter* should only be used together with *FixedLocator*. Otherwise, the labels may end up in unexpected positions.

Set the sequence *seq* of strings that will be used for labels.

get_offset(*self*)

set_offset_string(*self*, *ofs*)

class matplotlib.ticker.**FixedLocator**(*locs*, *nbins=None*)

Bases: *matplotlib.ticker.Locator*

Tick locations are fixed. If nbins is not None, the array of possible positions will be subsampled to keep the number of ticks <= nbins +1. The subsampling will be done so as to include the smallest absolute value; for example, if zero is included in the array of possibilities, then it is guaranteed to be one of the chosen ticks.

set_params(*self*, *nbins=None*)

Set parameters within this locator.

tick_values(*self*, *vmin*, *vmax*)

Return the locations of the ticks.

Note: Because the values are fixed, vmin and vmax are not used in this method.

---

class matplotlib.ticker.**FormatStrFormatter**(*fmt*)

Bases: *matplotlib.ticker.Formatter*

Use an old-style ('%' operator) format string to format the tick.

The format string should have a single variable format (%) in it. It will be applied to the value (not the position) of the tick.

class matplotlib.ticker.**Formatter**

Bases: *matplotlib.ticker.TickHelper*

Create a string based on a tick value and location.

static fix_minus(*s*)

Some classes may want to replace a hyphen for minus with the proper unicode symbol (U+2212) for typographical correctness. This is a helper method to perform such a replacement when it is enabled via `rcParams["axes.unicode_minus"]` (default: `True`).

format_data(*self*, *value*)

Return the full string representation of the value with the position unspecified.

format_data_short(*self*, *value*)

Return a short string version of the tick value.

Defaults to the position-independent long value.

format_ticks(*self*, *values*)

Return the tick labels for all the ticks at once.

get_offset(*self*)

locs = []

set_locs(*self*, *locs*)

Set the locations of the ticks.

This method is called before computing the tick labels because some formatters need to know all tick locations to do so.

class matplotlib.ticker.**FuncFormatter**(*func*)

Bases: *matplotlib.ticker.Formatter*

Use a user-defined function for formatting.

The function should take in two inputs (a tick value `x` and a position `pos`), and return a string containing the corresponding tick label.

get_offset(*self*)

set_offset_string(*self*, *ofs*)

class matplotlib.ticker.**IndexFormatter**(***kwargs*)

Bases: *matplotlib.ticker.Formatter*

[*Deprecated*] Format the position x to the nearest i-th label where `i = int(x + 0.5)`. Positions where `i < 0` or `i > len(list)` have no tick labels.

Parameters

labels

[list] List of labels.

Notes

Deprecated since version 3.3.

class matplotlib.ticker.**IndexLocator**(*base*, *offset*)

Bases: *matplotlib.ticker.Locator*

Place a tick on every multiple of some base number of points plotted, e.g., on every 5th point. It is assumed that you are doing index plotting; i.e., the axis is 0, len(data). This is mainly useful for x ticks.

Place ticks every *base* data point, starting at *offset*.

set_params(*self*, *base=None*, *offset=None*)

Set parameters within this locator

tick_values(*self*, *vmin*, *vmax*)

Return the values of the located ticks given **vmin** and **vmax**.

Note: To get tick locations with the vmin and vmax values defined automatically for the associated `axis` simply call the Locator instance:

```
>>> print(type(loc))
<type 'Locator'>
>>> print(loc())
[1, 2, 3, 4]
```

---

class matplotlib.ticker.**LinearLocator**(*numticks=None*, *presets=None*)

Bases: *matplotlib.ticker.Locator*

Determine the tick locations

The first time this function is called it will try to set the number of ticks to make a nice tick partitioning. Thereafter the number of ticks will be fixed so that interactive navigation will be nice

Use presets to set locs based on lom. A dict mapping vmin, vmax->locs

property numticks

set_params(*self*, *numticks=None*, *presets=None*)

Set parameters within this locator.

tick_values(*self*, *vmin*, *vmax*)

Return the values of the located ticks given **vmin** and **vmax**.

Note: To get tick locations with the vmin and vmax values defined automatically for the associated `axis` simply call the Locator instance:

```
>>> print(type(loc))
<type 'Locator'>
>>> print(loc())
[1, 2, 3, 4]
```

`view_limits`(*self*, *vmin*, *vmax*)
Try to choose the view limits intelligently.

`class matplotlib.ticker.Locator`
Bases: *matplotlib.ticker.TickHelper*

Determine the tick locations;

Note that the same locator should not be used across multiple *Axis* because the locator stores references to the Axis data and view limits.

`MAXTICKS = 1000`

`autoscale`(*self*)
[*Deprecated*] Autoscale the view limits.

Notes

Deprecated since version 3.2.

`nonsingular`(*self*, *v0*, *v1*)
Adjust a range as needed to avoid singularities.

This method gets called during autoscaling, with `(v0, v1)` set to the data limits on the axes if the axes contains any data, or `(-inf, +inf)` if not.

- If `v0 == v1` (possibly up to some floating point slop), this method returns an expanded interval around this value.
- If `(v0, v1) == (-inf, +inf)`, this method returns appropriate default view limits.
- Otherwise, `(v0, v1)` is returned without modification.

`pan`(*self*, *numsteps*)
[*Deprecated*] Pan numticks (can be positive or negative)

Notes

Deprecated since version 3.3.

`raise_if_exceeds`(*self*, *locs*)
Log at WARNING level if *locs* is longer than *Locator.MAXTICKS*.

This is intended to be called immediately before returning *locs* from `__call__` to inform users in case their Locator returns a huge number of ticks, causing Matplotlib to run out of memory.

The "strange" name of this method dates back to when it would raise an exception instead of emitting a log.

`refresh`(*self*)
[*Deprecated*] Refresh internal information based on current limits.

Notes

Deprecated since version 3.3.

`set_params`(*self*, ***kwargs*)
Do nothing, and raise a warning. Any locator class not supporting the set_params() function will call this.

`tick_values`(*self*, *vmin*, *vmax*)
Return the values of the located ticks given **vmin** and **vmax**.

Note: To get tick locations with the vmin and vmax values defined automatically for the associated `axis` simply call the Locator instance:

```
>>> print(type(loc))
<type 'Locator'>
>>> print(loc())
[1, 2, 3, 4]
```

`view_limits`(*self*, *vmin*, *vmax*)
Select a scale for the range from vmin to vmax.

Subclasses should override this method to change locator behaviour.

`zoom`(*self*, *direction*)
[*Deprecated*] Zoom in/out on axis; if direction is >0 zoom in, else zoom out.

Notes

Deprecated since version 3.3.

class matplotlib.ticker.**LogFormatter**(*base=10.0*, *labelOnlyBase=False*, *minor_thresholds=None*, *linthresh=None*)

Bases: *matplotlib.ticker.Formatter*

Base class for formatting ticks on a log or symlog scale.

It may be instantiated directly, or subclassed.

Parameters

base

[float, default: 10.] Base of the logarithm used in all calculations.

labelOnlyBase

[bool, default: False] If True, label ticks only at integer powers of base. This is normally True for major ticks and False for minor ticks.

minor_thresholds

[(subset, all), default: (1, 0.4)] If labelOnlyBase is False, these two numbers control the labeling of ticks that are not at integer powers of base; normally these are the minor ticks. The controlling parameter is the log of the axis data range. In the typical case where base is 10 it is the number of decades spanned by the axis, so we can call it 'numdec'. If `numdec <= all`, all minor ticks will be labeled. If `all < numdec <= subset`, then only a subset of minor ticks will be labeled, so as to avoid crowding. If `numdec > subset` then no minor ticks will be labeled.

linthresh

[None or float, default: None] If a symmetric log scale is in use, its `linthresh` parameter must be supplied here.

Notes

The *set_locs* method must be called to enable the subsetting logic controlled by the `minor_thresholds` parameter.

In some cases such as the colorbar, there is no distinction between major and minor ticks; the tick locations might be set manually, or by a locator that puts ticks at integer powers of base and at intermediate locations. For this situation, disable the minor_thresholds logic by using `minor_thresholds=(np.inf, np.inf)`, so that all ticks will be labeled.

To disable labeling of minor ticks when 'labelOnlyBase' is False, use `minor_thresholds=(0, 0)`. This is the default for the "classic" style.

Examples

To label a subset of minor ticks when the view limits span up to 2 decades, and all of the ticks when zoomed in to 0.5 decades or less, use `minor_thresholds=(2, 0.5)`.

To label all minor ticks when the view limits span up to 1.5 decades, use `minor_thresholds=(1.5, 1.5)`.

`base`(*self*, *base*)
Change the *base* for labeling.

> **Warning:** Should always match the base used for *LogLocator*

`format_data`(*self*, *value*)
Return the full string representation of the value with the position unspecified.

`format_data_short`(*self*, *value*)
Return a short string version of the tick value.

Defaults to the position-independent long value.

`label_minor`(*self*, *labelOnlyBase*)
Switch minor tick labeling on or off.

Parameters

labelOnlyBase

[bool] If True, label ticks only at integer powers of base.

`set_locs`(*self*, *locs=None*)
Use axis view limits to control which ticks are labeled.

The *locs* parameter is ignored in the present algorithm.

`class matplotlib.ticker.LogFormatterExponent`(*base=10.0*, *labelOnlyBase=False*, *minor_thresholds=None*, *linthresh=None*)
Bases: *matplotlib.ticker.LogFormatter*

Format values for log axis using `exponent = log_base(value)`.

class matplotlib.ticker.**LogFormatterMathtext**(*base=10.0*, *labelOnlyBase=False*, *minor_thresholds=None*, *linthresh=None*)

Bases: *matplotlib.ticker.LogFormatter*

Format values for log axis using `exponent = log_base(value)`.

class matplotlib.ticker.**LogFormatterSciNotation**(*base=10.0*, *labelOnlyBase=False*, *minor_thresholds=None*, *linthresh=None*)

Bases: *matplotlib.ticker.LogFormatterMathtext*

Format values following scientific notation in a logarithmic axis.

class matplotlib.ticker.**LogLocator**(*base=10.0*, *subs=1.0*, *numdecs=4*, *numticks=None*)

Bases: *matplotlib.ticker.Locator*

Determine the tick locations for log axes

Place ticks on the locations : subs[j] * base**i

Parameters

subs

[None or str or sequence of float, default: (1.0,)] Gives the multiples of integer powers of the base at which to place ticks. The default places ticks only at integer powers of the base. The permitted string values are `'auto'` and `'all'`, both of which use an algorithm based on the axis view limits to determine whether and how to put ticks between integer powers of the base. With `'auto'`, ticks are placed only between integer powers; with `'all'`, the integer powers are included. A value of None is equivalent to `'auto'`.

base(*self*, *base*)

Set the log base (major tick every `base**i`, i integer).

nonsingular(*self*, *vmin*, *vmax*)

Adjust a range as needed to avoid singularities.

This method gets called during autoscaling, with `(v0, v1)` set to the data limits on the axes if the axes contains any data, or `(-inf, +inf)` if not.

- If `v0 == v1` (possibly up to some floating point slop), this method returns an expanded interval around this value.
- If `(v0, v1) == (-inf, +inf)`, this method returns appropriate default view limits.
- Otherwise, `(v0, v1)` is returned without modification.

`set_params`(*self*, *base=None*, *subs=None*, *numdecs=None*, *numticks=None*)
Set parameters within this locator.

`subs`(*self*, *subs*)
Set the minor ticks for the log scaling every `base**i*subs[j]`.

`tick_values`(*self*, *vmin*, *vmax*)
Return the values of the located ticks given **vmin** and **vmax**.

Note: To get tick locations with the vmin and vmax values defined automatically for the associated `axis` simply call the Locator instance:

```
>>> print(type(loc))
<type 'Locator'>
>>> print(loc())
[1, 2, 3, 4]
```

`view_limits`(*self*, *vmin*, *vmax*)
Try to choose the view limits intelligently.

class matplotlib.ticker.**LogitFormatter**(*, *use_overline=False*, *one_half='\x0crac{1}{2}'*, *minor=False*, *minor_threshold=25*, *minor_number=6*)
Bases: *matplotlib.ticker.Formatter*

Probability formatter (using Math text).

Parameters

use_overline

[bool, default: False] If x > 1/2, with x = 1-v, indicate if x should be displayed as $overline{v}$. The default is to display $1-v$.

one_half

[str, default: r"frac{1}{2}"] The string used to represent 1/2.

minor

[bool, default: False] Indicate if the formatter is formatting minor ticks or not. Basically minor ticks are not labelled, except when only few ticks are provided, ticks with most space with neighbor ticks are labelled. See other parameters to change the default behavior.

minor_threshold

[int, default: 25] Maximum number of locs for labelling some minor ticks. This parameter have no effect if minor is False.

minor_number

[int, default: 6] Number of ticks which are labelled when the number of ticks is below the threshold.

`format_data_short`(*self*, *value*)
Return a short string version of the tick value.

Defaults to the position-independent long value.

`set_locs`(*self*, *locs*)
Set the locations of the ticks.

This method is called before computing the tick labels because some formatters need to know all tick locations to do so.

`set_minor_number`(*self*, *minor_number*)
Set the number of minor ticks to label when some minor ticks are labelled.

Parameters

minor_number

[int] Number of ticks which are labelled when the number of ticks is below the threshold.

`set_minor_threshold`(*self*, *minor_threshold*)
Set the threshold for labelling minors ticks.

Parameters

minor_threshold

[int] Maximum number of locations for labelling some minor ticks. This parameter have no effect if minor is False.

`set_one_half`(*self*, *one_half*)
Set the way one half is displayed.

one_half

[str, default: r"frac{1}{2}"] The string used to represent 1/2.

`use_overline`(*self*, *use_overline*)
Switch display mode with overline for labelling p>1/2.

Parameters

use_overline

[bool, default: False] If x > 1/2, with x = 1-v, indicate if x should be displayed as $overline{v}$. The default is to display $1-v$.

class matplotlib.ticker.LogitLocator(*minor=False*, ***, *nbins='auto'*)

Bases: *matplotlib.ticker.MaxNLocator*

Determine the tick locations for logit axes

Place ticks on the logit locations

Parameters

nbins

[int or 'auto', optional] Number of ticks. Only used if minor is False.

minor

[bool, default: False] Indicate if this locator is for minor ticks or not.

property minor

nonsingular(*self*, *vmin*, *vmax*)

Adjust a range as needed to avoid singularities.

This method gets called during autoscaling, with `(v0, v1)` set to the data limits on the axes if the axes contains any data, or `(-inf, +inf)` if not.

- If `v0 == v1` (possibly up to some floating point slop), this method returns an expanded interval around this value.
- If `(v0, v1) == (-inf, +inf)`, this method returns appropriate default view limits.
- Otherwise, `(v0, v1)` is returned without modification.

set_params(*self*, *minor=None*, ***kwargs*)

Set parameters within this locator.

tick_values(*self*, *vmin*, *vmax*)

Return the values of the located ticks given **vmin** and **vmax**.

Note: To get tick locations with the vmin and vmax values defined automatically for the associated `axis` simply call the Locator instance:

```
>>> print(type(loc))
<type 'Locator'>
>>> print(loc())
[1, 2, 3, 4]
```

---

class matplotlib.ticker.MaxNLocator(**args*, ***kwargs*)

Bases: *matplotlib.ticker.Locator*

Find nice tick locations with no more than N being within the view limits. Locations beyond the limits are added to support autoscaling.

Parameters

nbins

[int or 'auto', default: 10] Maximum number of intervals; one less than max number of ticks. If the string 'auto', the number of bins will be automatically determined based on the length of the axis.

steps

[array-like, optional] Sequence of nice numbers starting with 1 and ending with 10; e.g., [1, 2, 4, 5, 10], where the values are acceptable tick multiples. i.e. for the example, 20, 40, 60 would be an acceptable set of ticks, as would 0.4, 0.6, 0.8, because they are multiples of 2. However, 30, 60, 90 would not be allowed because 3 does not appear in the list of steps.

integer

[bool, default: False] If True, ticks will take only integer values, provided at least *min_n_ticks* integers are found within the view limits.

symmetric

[bool, default: False] If True, autoscaling will result in a range symmetric about zero.

prune

[{'lower', 'upper', 'both', None}, default: None] Remove edge ticks -- useful for stacked or ganged plots where the upper tick of one axes overlaps with the lower tick of the axes above it, primarily when `rcParams["axes.autolimit_mode"]` (default: `'data'`) is `'round_numbers'`. If `prune=='lower'`, the smallest tick will be removed. If `prune == 'upper'`, the largest tick will be removed. If `prune == 'both'`, the largest and smallest ticks will be removed. If *prune* is *None*, no ticks will be removed.

min_n_ticks

[int, default: 2] Relax *nbins* and *integer* constraints if necessary to obtain this minimum number of ticks.

`default_params = {'integer': False, 'min_n_ticks': 2, 'nbins': 10, 'prune': None, 'steps`

`set_params`(*self*, ***kwargs*)

Set parameters for this locator.

Parameters

nbins

[int or 'auto', optional] see *MaxNLocator*

steps

[array-like, optional] see *MaxNLocator*

integer

[bool, optional] see *MaxNLocator*

symmetric

[bool, optional] see *MaxNLocator*

prune

[{'lower', 'upper', 'both', None}, optional] see *MaxNLocator*

min_n_ticks

[int, optional] see *MaxNLocator*

`tick_values`(*self*, *vmin*, *vmax*)
Return the values of the located ticks given **vmin** and **vmax**.

Note: To get tick locations with the vmin and vmax values defined automatically for the associated `axis` simply call the Locator instance:

```
>>> print(type(loc))
<type 'Locator'>
>>> print(loc())
[1, 2, 3, 4]
```

`view_limits`(*self*, *dmin*, *dmax*)
Select a scale for the range from vmin to vmax.

Subclasses should override this method to change locator behaviour.

`class matplotlib.ticker.MultipleLocator`(*base=1.0*)
Bases: *matplotlib.ticker.Locator*

Set a tick on each integer multiple of a base within the view interval.

`set_params`(*self*, *base*)
Set parameters within this locator.

`tick_values`(*self*, *vmin*, *vmax*)
Return the values of the located ticks given **vmin** and **vmax**.

Note: To get tick locations with the vmin and vmax values defined automatically for the associated `axis` simply call the Locator instance:

```
>>> print(type(loc))
<type 'Locator'>
>>> print(loc())
[1, 2, 3, 4]
```

`view_limits`(*self*, *dmin*, *dmax*)
Set the view limits to the nearest multiples of base that contain the data.

`class matplotlib.ticker.NullFormatter`
Bases: *matplotlib.ticker.Formatter*

Always return the empty string.

`class matplotlib.ticker.NullLocator`
Bases: *matplotlib.ticker.Locator*

No ticks

`tick_values`(*self*, *vmin*, *vmax*)
Return the locations of the ticks.

Note: Because the values are Null, vmin and vmax are not used in this method.

`class matplotlib.ticker.OldAutoLocator`(**args*, ***kwargs*)
Bases: *matplotlib.ticker.Locator*

[*Deprecated*] On autoscale this class picks the best MultipleLocator to set the view limits and the tick locs.

Notes

Deprecated since version 3.3.

`get_locator`(*self*, *d*)
Pick the best locator based on a distance *d*.

`tick_values`(*self*, *vmin*, *vmax*)
Return the values of the located ticks given **vmin** and **vmax**.

Note: To get tick locations with the vmin and vmax values defined automatically for the associated `axis` simply call the Locator instance:

```
>>> print(type(loc))
<type 'Locator'>
>>> print(loc())
[1, 2, 3, 4]
```

`view_limits`(*self*, *vmin*, *vmax*)
Select a scale for the range from vmin to vmax.

Subclasses should override this method to change locator behaviour.

class matplotlib.ticker.**OldScalarFormatter**(**args*, ***kwargs*)

Bases: *matplotlib.ticker.Formatter*

[*Deprecated*] Tick location is a plain old number.

Notes

Deprecated since version 3.3.

class matplotlib.ticker.**PercentFormatter**(*xmax=100*, *decimals=None*, *symbol='%'*, *is_latex=False*)

Bases: *matplotlib.ticker.Formatter*

Format numbers as a percentage.

Parameters

xmax

[float] Determines how the number is converted into a percentage. *xmax* is the data value that corresponds to 100%. Percentages are computed as `x / xmax * 100`. So if the data is already scaled to be percentages, *xmax* will be 100. Another common situation is where *xmax* is 1.0.

decimals

[None or int] The number of decimal places to place after the point. If *None* (the default), the number will be computed automatically.

symbol

[str or None] A string that will be appended to the label. It may be *None* or empty to indicate that no symbol should be used. LaTeX special characters are escaped in *symbol* whenever latex mode is enabled, unless *is_latex* is *True*.

is_latex

[bool] If *False*, reserved LaTeX characters in *symbol* will be escaped.

convert_to_pct(*self*, *x*)

format_pct(*self*, *x*, *display_range*)

Format the number as a percentage number with the correct number of decimals and adds the percent symbol, if any.

If `self.decimals` is `None`, the number of digits after the decimal point is set based on the *display_range* of the axis as follows:

display_range	decimals	sample
>50	0	`x = 34.5` => 35%
>5	1	`x = 34.5` => 34.5%
>0.5	2	`x = 34.5` => 34.50%
...	...	...

This method will not be very good for tiny axis ranges or extremely large ones. It assumes that the values on the chart are percentages displayed on a reasonable scale.

`property symbol`

The configured percent symbol as a string.

If LaTeX is enabled via `rcParams["text.usetex"]` (default: `False`), the special characters `{'#', '$', '%', '&', '~', '_', '^', '\', '{', '}'}` are automatically escaped in the string.

`class matplotlib.ticker.ScalarFormatter(`*useOffset=None*, *useMathText=None*, *useLocale=None*`)`

Bases: *matplotlib.ticker.Formatter*

Format tick values as a number.

Parameters

useOffset

[bool or float, default: `rcParams["axes.formatter.useoffset"]` (default: `True`)] Whether to use offset notation. See *set_useOffset*.

useMathText

[bool, default: `rcParams["axes.formatter.use_mathtext"]` (default: `False`)] Whether to use fancy math formatting. See *set_useMathText*.

useLocale

[bool, default: `rcParams["axes.formatter.use_locale"]` (default: `False`).] Whether to use locale settings for decimal sign and positive sign. See *set_useLocale*.

Notes

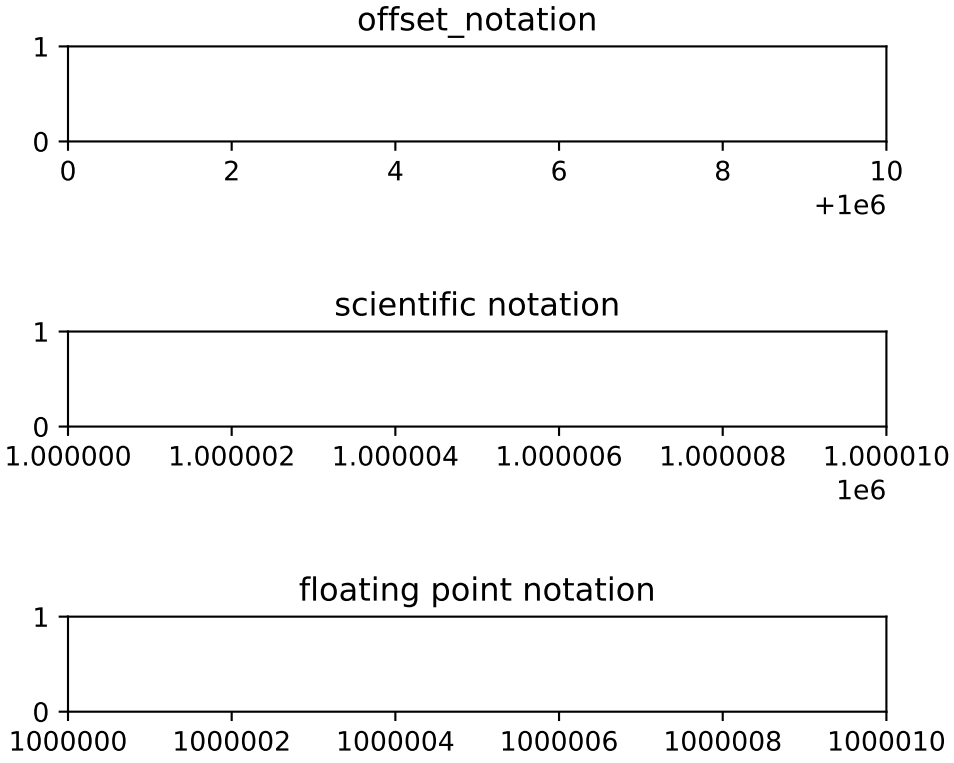
In addition to the parameters above, the formatting of scientific vs. floating point representation can be configured via *set_scientific* and *set_powerlimits*).

Offset notation and scientific notation

Offset notation and scientific notation look quite similar at first sight. Both split some information from the formatted tick values and display it at the end of the axis.

- The scientific notation splits up the order of magnitude, i.e. a multiplicative scaling factor, e.g. `1e6`.
- The offset notation separates an additive constant, e.g. `+1e6`. The offset notation label is always prefixed with a + or – sign and is thus distinguishable from the order of magnitude label.

The following plot with x limits `1_000_000` to `1_000_010` illustrates the different formatting. Note the labels at the right edge of the x axis.



`format_data`(*self*, *value*)
: Return the full string representation of the value with the position unspecified.

`format_data_short`(*self*, *value*)
: Return a short string version of the tick value.

  Defaults to the position-independent long value.

`get_offset`(*self*)
: Return scientific notation, plus offset.

`get_useLocale`(*self*)
: Return whether locale settings are used for formatting.

  See also:

ScalarFormatter.set_useLocale

`get_useMathText`(*self*)
Return whether to use fancy math formatting.

See also:

ScalarFormatter.set_useMathText

`get_useOffset`(*self*)
Return whether automatic mode for offset notation is active.

This returns True if `set_useOffset(True)`; it returns False if an explicit offset was set, e.g. `set_useOffset(1000)`.

See also:

ScalarFormatter.set_useOffset

`set_locs`(*self*, *locs*)
Set the locations of the ticks.

This method is called before computing the tick labels because some formatters need to know all tick locations to do so.

`set_powerlimits`(*self*, *lims*)
Set size thresholds for scientific notation.

Parameters

lims

[(int, int)] A tuple *(min_exp, max_exp)* containing the powers of 10 that determine the switchover threshold. For a number representable as $a \times 10^{\text{exp}}$ with $1 <= |a| < 10$, scientific notation will be used if `exp <= min_exp` or `exp >= max_exp`.

The default limits are controlled by `rcParams["axes.formatter.limits"]` (default: `[-5, 6]`).

In particular numbers with *exp* equal to the thresholds are written in scientific notation.

Typically, *min_exp* will be negative and *max_exp* will be positive.

For example, `formatter.set_powerlimits((-3, 4))` will provide the following formatting: $1 \times 10^{-3}, 9.9 \times 10^{-3}, 0.01, 9999, 1 \times 10^{4}$.

See also:

ScalarFormatter.set_scientific

`set_scientific`(*self*, *b*)
Turn scientific notation on or off.

See also:

ScalarFormatter.set_powerlimits

`set_useLocale`(*self*, *val*)
Set whether to use locale settings for decimal sign and positive sign.

Parameters

val

[bool or None] *None* resets to `rcParams["axes.formatter.use_locale"]` (default: `False`).

`set_useMathText`(*self*, *val*)
Set whether to use fancy math formatting.

If active, scientific notation is formatted as 1.2×10^3.

Parameters

val

[bool or None] *None* resets to `rcParams["axes.formatter.use_mathtext"]` (default: `False`).

`set_useOffset`(*self*, *val*)
Set whether to use offset notation.

When formatting a set numbers whose value is large compared to their range, the formatter can separate an additive constant. This can shorten the formatted numbers so that they are less likely to overlap when drawn on an axis.

Parameters

val

[bool or float]

- If False, do not use offset notation.
- If True (=automatic mode), use offset notation if it can make the residual numbers significantly shorter. The exact behavior is controlled by `rcParams["axes.formatter.offset_threshold"]` (default: `4`).
- If a number, force an offset of the given value.

Examples

With active offset notation, the values

`100_000, 100_002, 100_004, 100_006, 100_008`

will be formatted as `0, 2, 4, 6, 8` plus an offset `+1e5`, which is written to the edge of the axis.

property useLocale
Return whether locale settings are used for formatting.

See also:

ScalarFormatter.set_useLocale

property useMathText
Return whether to use fancy math formatting.

See also:

ScalarFormatter.set_useMathText

property useOffset
Return whether automatic mode for offset notation is active.

This returns True if `set_useOffset(True)`; it returns False if an explicit offset was set, e.g. `set_useOffset(1000)`.

See also:

ScalarFormatter.set_useOffset

class matplotlib.ticker.**StrMethodFormatter**(*fmt*)
Bases: *matplotlib.ticker.Formatter*

Use a new-style format string (as used by `str.format`) to format the tick.

The field used for the tick value must be labeled *x* and the field used for the tick position must be labeled *pos*.

class matplotlib.ticker.**SymmetricalLogLocator**(*transform=None*, *subs=None*, *linthresh=None*, *base=None*)
Bases: *matplotlib.ticker.Locator*

Determine the tick locations for symmetric log axes.

Parameters

transform

[*SymmetricalLogTransform*, optional] If set, defines the *base* and *linthresh* of the symlog transform.

base, linthresh

[float, optional] The *base* and *linthresh* of the symlog transform, as documented for *SymmetricalLogScale*. These parameters are only used if *transform* is not set.

subs

[sequence of float, default: [1]] The multiples of integer powers of the base where ticks are placed, i.e., ticks are placed at `[sub * base**i for i in ... for sub in subs]`.

Notes

Either *transform*, or both *base* and *linthresh*, must be given.

`set_params`(*self*, *subs=None*, *numticks=None*)
Set parameters within this locator.

`tick_values`(*self*, *vmin*, *vmax*)
Return the values of the located ticks given **vmin** and **vmax**.

Note: To get tick locations with the vmin and vmax values defined automatically for the associated `axis` simply call the Locator instance:

```
>>> print(type(loc))
<type 'Locator'>
>>> print(loc())
[1, 2, 3, 4]
```

`view_limits`(*self*, *vmin*, *vmax*)
Try to choose the view limits intelligently.

`class matplotlib.ticker.TickHelper`
Bases: `object`

`axis = None`

`create_dummy_axis`(*self*, ***kwargs*)

`set_axis`(*self*, *axis*)

`set_bounds`(*self*, *vmin*, *vmax*)

`set_data_interval`(*self*, *vmin*, *vmax*)

`set_view_interval`(*self*, *vmin*, *vmax*)

18.54 matplotlib.tight_layout

Routines to adjust subplot params so that subplots are nicely fit in the figure. In doing so, only axis labels, tick labels, axes titles and offsetboxes that are anchored to axes are currently considered.

Internally, this module assumes that the margins (left_margin, etc.) which are differences between ax.get_tightbbox and ax.bbox are independent of axes position. This may fail if Axes.adjustable is datalim. Also, This will fail for some cases (for example, left or right margin is affected by xlabel).

matplotlib.tight_layout.**auto_adjust_subplotpars**(*fig*, *renderer*, *nrows_ncols*, *num1num2_list*, *subplot_list*, *ax_bbox_list=None*, *pad=1.08*, *h_pad=None*, *w_pad=None*, *rect=None*)

Return a dict of subplot parameters to adjust spacing between subplots or `None` if resulting axes would have zero height or width.

Note that this function ignores geometry information of subplot itself, but uses what is given by the *nrows_ncols* and *num1num2_list* parameters. Also, the results could be incorrect if some subplots have `adjustable=datalim`.

Parameters

nrows_ncols

[Tuple[int, int]] Number of rows and number of columns of the grid.

num1num2_list

[List[int]] List of numbers specifying the area occupied by the subplot

subplot_list

[list of subplots] List of subplots that will be used to calculate optimal subplot_params.

pad

[float] Padding between the figure edge and the edges of subplots, as a fraction of the font size.

h_pad, w_pad

[float] Padding (height/width) between edges of adjacent subplots, as a fraction of the font size. Defaults to *pad*.

rect

[Tuple[float, float, float, float]] [left, bottom, right, top] in normalized (0, 1) figure coordinates.

matplotlib.tight_layout.**get_renderer**(*fig*)

matplotlib.tight_layout.**get_subplotspec_list**(*axes_list*, *grid_spec=None*)

Return a list of subplotspec from the given list of axes.

For an instance of axes that does not support subplotspec, None is inserted in the list.

If grid_spec is given, None is inserted for those not from the given grid_spec.

matplotlib.tight_layout.**get_tight_layout_figure**(*fig*, *axes_list*, *subplotspec_list*, *renderer*, *pad=1.08*, *h_pad=None*, *w_pad=None*, *rect=None*)

Return subplot parameters for tight-layouted-figure with specified padding.

Parameters

fig

[Figure]

axes_list

[list of Axes]

subplotspec_list

[list of *SubplotSpec*] The subplotspecs of each axes.

renderer

[renderer]

pad

[float] Padding between the figure edge and the edges of subplots, as a fraction of the font size.

h_pad, w_pad

[float] Padding (height/width) between edges of adjacent subplots. Defaults to *pad*.

rect

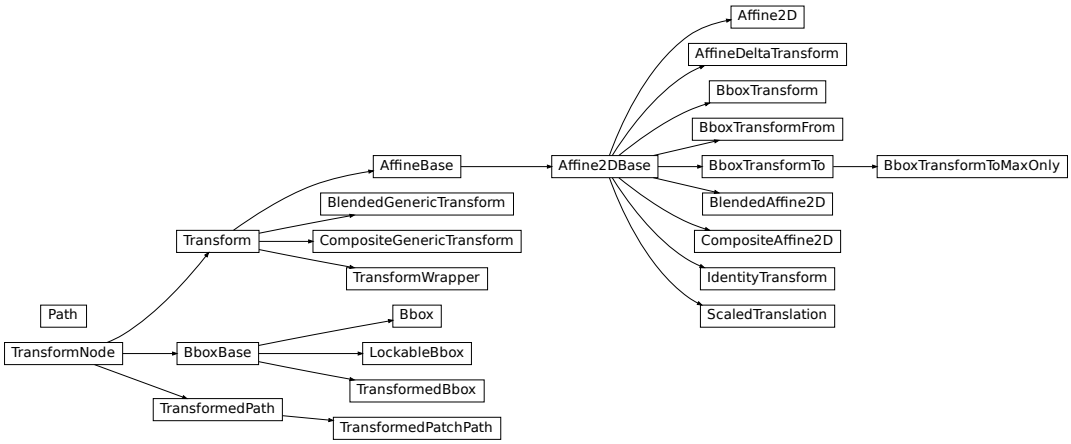
[Tuple[float, float, float, float], optional] (left, bottom, right, top) rectangle in normalized figure coordinates that the whole subplots area (including labels) will fit into. Defaults to using the entire figure.

Returns

subplotspec or None

subplotspec kwargs to be passed to *Figure.subplots_adjust* or None if tight_layout could not be accomplished.

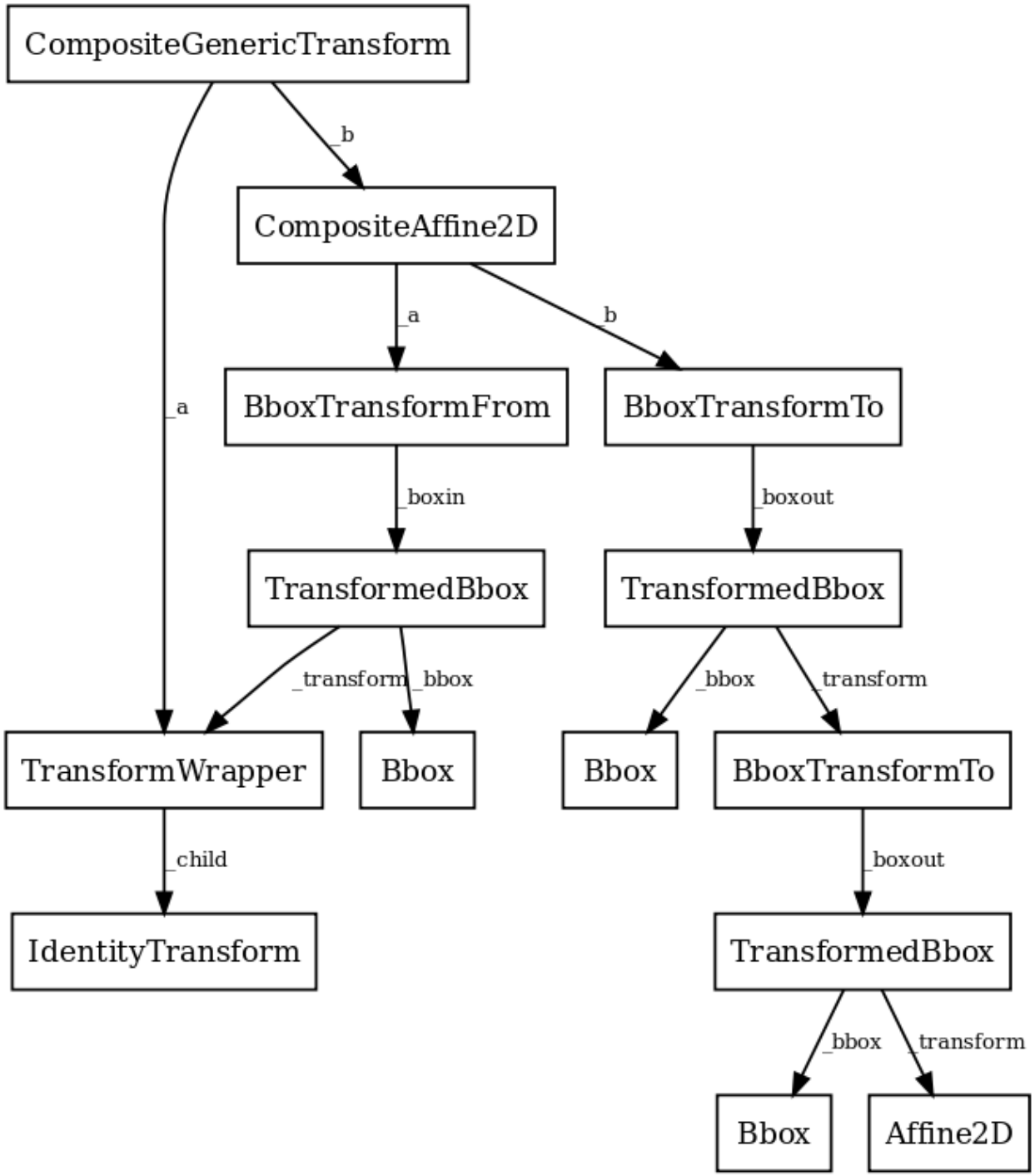
18.55 matplotlib.transforms



Matplotlib includes a framework for arbitrary geometric transformations that is used determine the final position of all elements drawn on the canvas.

Transforms are composed into trees of *TransformNode* objects whose actual value depends on their children. When the contents of children change, their parents are automatically invalidated. The next time an invalidated transform is accessed, it is recomputed to reflect those changes. This invalidation/caching approach prevents unnecessary recomputations of transforms, and contributes to better interactive performance.

For example, here is a graph of the transform tree used to plot data to the graph:



The framework can be used for both affine and non-affine transformations. However, for speed, we want use the backend renderers to perform affine transformations whenever possible. Therefore, it is possible to perform just the affine or non-affine part of a transformation on a set of data. The affine is always assumed to occur after the non-affine. For any transform:

```
full transform == non-affine part + affine part
```

The backends are not expected to handle non-affine transformations themselves.

class matplotlib.transforms.**Affine2D**(*matrix=None*, ***kwargs*)

Bases: *matplotlib.transforms.Affine2DBase*

A mutable 2D affine transformation.

Initialize an Affine transform from a 3x3 numpy float array:

```
a c e
b d f
0 0 1
```

If *matrix* is None, initialize with the identity transform.

__init__(*self*, *matrix=None*, ***kwargs*)

Initialize an Affine transform from a 3x3 numpy float array:

```
a c e
b d f
0 0 1
```

If *matrix* is None, initialize with the identity transform.

__module__ = 'matplotlib.transforms'

__str__(*self*)

Return str(self).

clear(*self*)

Reset the underlying matrix to the identity transform.

static from_values(*a*, *b*, *c*, *d*, *e*, *f*)

Create a new Affine2D instance from the given values:

```
a c e
b d f
0 0 1
```

.

get_matrix(*self*)

Get the underlying transformation matrix as a 3x3 numpy array:

```
a c e
b d f
0 0 1
```

.

static identity()

Return a new *Affine2D* object that is the identity transform.

Unless this transform will be mutated later on, consider using the faster *IdentityTransform* class instead.

rotate(*self*, *theta*)
Add a rotation (in radians) to this transform in place.

Returns *self*, so this method can easily be chained with more calls to *rotate()*, *rotate_deg()*, *translate()* and *scale()*.

rotate_around(*self*, *x*, *y*, *theta*)
Add a rotation (in radians) around the point (x, y) in place.

Returns *self*, so this method can easily be chained with more calls to *rotate()*, *rotate_deg()*, *translate()* and *scale()*.

rotate_deg(*self*, *degrees*)
Add a rotation (in degrees) to this transform in place.

Returns *self*, so this method can easily be chained with more calls to *rotate()*, *rotate_deg()*, *translate()* and *scale()*.

rotate_deg_around(*self*, *x*, *y*, *degrees*)
Add a rotation (in degrees) around the point (x, y) in place.

Returns *self*, so this method can easily be chained with more calls to *rotate()*, *rotate_deg()*, *translate()* and *scale()*.

scale(*self*, *sx*, *sy=None*)
Add a scale in place.

If *sy* is None, the same scale is applied in both the *x*- and *y*-directions.

Returns *self*, so this method can easily be chained with more calls to *rotate()*, *rotate_deg()*, *translate()* and *scale()*.

set(*self*, *other*)
Set this transformation from the frozen copy of another *Affine2DBase* object.

set_matrix(*self*, *mtx*)
Set the underlying transformation matrix from a 3x3 numpy array:

```
a c e
b d f
0 0 1
```

.

skew(*self*, *xShear*, *yShear*)
Add a skew in place.

xShear and *yShear* are the shear angles along the *x*- and *y*-axes, respectively, in radians.

Returns *self*, so this method can easily be chained with more calls to *rotate()*, *rotate_deg()*, *translate()* and *scale()*.

skew_deg(*self*, *xShear*, *yShear*)
Add a skew in place.

xShear and *yShear* are the shear angles along the *x*- and *y*-axes, respectively, in degrees.

Returns *self*, so this method can easily be chained with more calls to *rotate()*, *rotate_deg()*, *translate()* and *scale()*.

translate(*self*, *tx*, *ty*)
Add a translation in place.

Returns *self*, so this method can easily be chained with more calls to *rotate()*, *rotate_deg()*, *translate()* and *scale()*.

class matplotlib.transforms.**Affine2DBase**(**args*, ***kwargs*)
Bases: *matplotlib.transforms.AffineBase*

The base class of all 2D affine transformations.

2D affine transformations are performed using a 3x3 numpy array:

```
a c e
b d f
0 0 1
```

This class provides the read-only interface. For a mutable 2D affine transformation, use *Affine2D*.

Subclasses of this class will generally only need to override a constructor and `get_matrix()` that generates a custom 3x3 matrix.

Parameters

shorthand_name

[str] A string representing the "name" of the transform. The name carries no significance other than to improve the readability of `str(transform)` when DEBUG=True.

__module__ = 'matplotlib.transforms'

frozen(*self*)
Return a frozen copy of this transform node. The frozen copy will not be updated when its children change. Useful for storing a previously known state of a transform where `copy.deepcopy()` might normally be used.

has_inverse = True

input_dims = 2

inverted(*self*)
Return the corresponding inverse transformation.

It holds `x == self.inverted().transform(self.transform(x))`.

The return value of this method should be treated as temporary. An update to *self* does not cause a corresponding update to its inverted copy.

`property is_separable`
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

`static matrix_from_values`(*a*, *b*, *c*, *d*, *e*, *f*)
[*Deprecated*] Create a new transformation matrix as a 3x3 numpy array of the form:

```
a c e
b d f
0 0 1
```

Notes

Deprecated since version 3.2.

`output_dims = 2`

`to_values`(*self*)
Return the values of the matrix as an `(a, b, c, d, e, f)` tuple.

`transform_affine`(*self*, *points*)
Apply only the affine part of this transformation on the given array of values.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally a no-op. In affine transformations, this is equivalent to `transform(values)`.

Parameters

values

[array] The input values as NumPy array of length *input_dims* or shape (N x *input_dims*).

Returns

array

The output values as NumPy array of length *input_dims* or shape (N x *output_dims*), depending on the input.

`class matplotlib.transforms.AffineBase`(**args*, ***kwargs*)
Bases: *matplotlib.transforms.Transform*

The base class of all affine transformations of any number of dimensions.

Parameters

shorthand_name

[str] A string representing the "name" of the transform. The name carries no significance other than to improve the readability of `str(transform)` when DEBUG=True.

`__array__`(*self*, **args*, ***kwargs*)
Array interface to get at this Transform's affine matrix.

`__eq__`(*self*, *other*)
Return self==value.

`__hash__ = None`

`__init__`(*self*, **args*, ***kwargs*)

Parameters

shorthand_name

[str] A string representing the "name" of the transform. The name carries no significance other than to improve the readability of `str(transform)` when DEBUG=True.

`__module__ = 'matplotlib.transforms'`

`get_affine`(*self*)
Get the affine part of this transform.

`is_affine = True`

`transform`(*self*, *values*)
Apply this transformation on the given array of *values*.

Parameters

values

[array] The input values as NumPy array of length `input_dims` or shape (N x `input_dims`).

Returns

array

The output values as NumPy array of length `input_dims` or shape (N x `output_dims`), depending on the input.

`transform_affine`(*self*, *values*)
Apply only the affine part of this transformation on the given array of values.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`

In non-affine transformations, this is generally a no-op. In affine transformations, this is equivalent to `transform(values)`.

Parameters

values

[array] The input values as NumPy array of length `input_dims` or shape (N x `input_dims`).

Returns

array

The output values as NumPy array of length `input_dims` or shape (N x `output_dims`), depending on the input.

`transform_non_affine`(*self*, *points*)

Apply only the non-affine part of this transformation.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`

In non-affine transformations, this is generally equivalent to `transform(values)`. In affine transformations, this is always a no-op.

Parameters

values

[array] The input values as NumPy array of length `input_dims` or shape (N x `input_dims`).

Returns

array

The output values as NumPy array of length `input_dims` or shape (N x `output_dims`), depending on the input.

`transform_path`(*self*, *path*)

Apply the transform to *Path* *path*, returning a new *Path*.

In some cases, this transform may insert curves into the path that began as line segments.

`transform_path_affine`(*self*, *path*)

Apply the affine part of this transform to *Path* *path*, returning a new *Path*.

`transform_path(path)` is equivalent to `transform_path_affine(transform_path_non_affine(va`

`transform_path_non_affine`(*self*, *path*)

Apply the non-affine part of this transform to *Path* *path*, returning a new *Path*.

`transform_path(path)` is equivalent to `transform_path_affine(transform_path_non_affine(va`

class matplotlib.transforms.**AffineDeltaTransform**(*transform*, ***kwargs*)

Bases: *matplotlib.transforms.Affine2DBase*

A transform wrapper for transforming displacements between pairs of points.

This class is intended to be used to transform displacements ("position deltas") between pairs of points (e.g., as the `offset_transform` of *Collection*s): given a transform `t` such that `t = AffineDeltaTransform(t) + offset`, `AffineDeltaTransform` satisfies `AffineDeltaTransform(a - b) == AffineDeltaTransform(a) - AffineDeltaTransform(b)`.

This is implemented by forcing the offset components of the transform matrix to zero.

This class is experimental as of 3.3, and the API may change.

Parameters

shorthand_name

[str] A string representing the "name" of the transform. The name carries no significance other than to improve the readability of `str(transform)` when DEBUG=True.

__init__(*self*, *transform*, ***kwargs*)

Parameters

shorthand_name

[str] A string representing the "name" of the transform. The name carries no significance other than to improve the readability of `str(transform)` when DEBUG=True.

__module__ = 'matplotlib.transforms'

__str__(*self*)

Return str(self).

get_matrix(*self*)

Get the matrix for the affine part of this transform.

class matplotlib.transforms.**Bbox**(*points*, ***kwargs*)

Bases: *matplotlib.transforms.BboxBase*

A mutable bounding box.

Examples

Create from known bounds

The default constructor takes the boundary "points" `[[xmin, ymin], [xmax, ymax]]`.

```
>>> Bbox([[1, 1], [3, 7]])
Bbox([[1.0, 1.0], [3.0, 7.0]])
```

Alternatively, a Bbox can be created from the flattened points array, the so-called "extents" `(xmin, ymin, xmax, ymax)`

```
>>> Bbox.from_extents(1, 1, 3, 7)
Bbox([[1.0, 1.0], [3.0, 7.0]])
```

or from the "bounds" `(xmin, ymin, width, height)`.

```
>>> Bbox.from_bounds(1, 1, 2, 6)
Bbox([[1.0, 1.0], [3.0, 7.0]])
```

Create from collections of points

The "empty" object for accumulating Bboxs is the null bbox, which is a stand-in for the empty set.

```
>>> Bbox.null()
Bbox([[inf, inf], [-inf, -inf]])
```

Adding points to the null bbox will give you the bbox of those points.

```
>>> box = Bbox.null()
>>> box.update_from_data_xy([[1, 1]])
>>> box
Bbox([[1.0, 1.0], [1.0, 1.0]])
>>> box.update_from_data_xy([[2, 3], [3, 2]], ignore=False)
>>> box
Bbox([[1.0, 1.0], [3.0, 3.0]])
```

Setting `ignore=True` is equivalent to starting over from a null bbox.

```
>>> box.update_from_data_xy([[1, 1]], ignore=True)
>>> box
Bbox([[1.0, 1.0], [1.0, 1.0]])
```

Warning: It is recommended to always specify `ignore` explicitly. If not, the default value of `ignore` can be changed at any time by code with access to your Bbox, for example using the method *ignore*.

Properties of the ``null`` bbox

Note: The current behavior of *Bbox.null()* may be surprising as it does not have all of the properties of the "empty set", and as such does not behave like a "zero" object in the mathematical sense. We may change that in the future (with a deprecation period).

The null bbox is the identity for intersections

```
>>> Bbox.intersection(Bbox([[1, 1], [3, 7]]), Bbox.null())
Bbox([[1.0, 1.0], [3.0, 7.0]])
```

except with itself, where it returns the full space.

```
>>> Bbox.intersection(Bbox.null(), Bbox.null())
Bbox([[-inf, -inf], [inf, inf]])
```

A union containing null will always return the full space (not the other set!)

```
>>> Bbox.union([Bbox([[0, 0], [0, 0]]), Bbox.null()])
Bbox([[-inf, -inf], [inf, inf]])
```

Parameters

points

[ndarray] A 2x2 numpy array of the form `[[x0, y0], [x1, y1]]`.

`__format__`(*self*, *fmt*)
Default object formatter.

`__init__`(*self*, *points*, ***kwargs*)

Parameters

points

[ndarray] A 2x2 numpy array of the form `[[x0, y0], [x1, y1]]`.

`__module__ = 'matplotlib.transforms'`

`__repr__`(*self*)
Return repr(self).

`__str__`(*self*)
Return str(self).

`property bounds`
Return (*x0*, *y0*, `width`, `height`).

`static from_bounds`(*x0*, *y0*, *width*, *height*)
Create a new *Bbox* from *x0*, *y0*, *width* and *height*.

width and *height* may be negative.

static from_extents(*args)
Create a new Bbox from *left*, *bottom*, *right* and *top*.

The *y*-axis increases upwards.

get_points(self)
Get the points of the bounding box directly as a numpy array of the form: `[[x0, y0], [x1, y1]]`.

ignore(self, value)
Set whether the existing bounds of the box should be ignored by subsequent calls to *update_from_data_xy()*.

value

[bool]

- When `True`, subsequent calls to *update_from_data_xy()* will ignore the existing bounds of the *Bbox*.
- When `False`, subsequent calls to *update_from_data_xy()* will include the existing bounds of the *Bbox*.

property intervalx
The pair of *x* coordinates that define the bounding box.

This is not guaranteed to be sorted from left to right.

property intervaly
The pair of *y* coordinates that define the bounding box.

This is not guaranteed to be sorted from bottom to top.

property minpos

property minposx

property minposy

mutated(self)
Return whether the bbox has changed since init.

mutatedx(self)
Return whether the x-limits have changed since init.

mutatedy(self)
Return whether the y-limits have changed since init.

static null()
Create a new null *Bbox* from (inf, inf) to (-inf, -inf).

property p0
The first pair of (*x*, *y*) coordinates that define the bounding box.

This is not guaranteed to be the bottom-left corner (for that, use `min`).

property p1
The second pair of (*x*, *y*) coordinates that define the bounding box.

This is not guaranteed to be the top-right corner (for that, use `max`).

`set`(*self*, *other*)
Set this bounding box from the "frozen" bounds of another *Bbox*.

`set_points`(*self*, *points*)
Set the points of the bounding box directly from a numpy array of the form: `[[x0, y0], [x1, y1]]`. No error checking is performed, as this method is mainly for internal use.

`static unit`()
Create a new unit *Bbox* from (0, 0) to (1, 1).

`update_from_data_xy`(*self*, *xy*, *ignore=None*, *updatex=True*, *updatey=True*)
Update the bounds of the *Bbox* based on the passed in data. After updating, the bounds will have positive *width* and *height*; *x0* and *y0* will be the minimal values.

Parameters

xy

[ndarray] A numpy array of 2D points.

ignore

[bool, optional]

- When `True`, ignore the existing bounds of the *Bbox*.
- When `False`, include the existing bounds of the *Bbox*.
- When `None`, use the last value passed to *ignore()*.

updatex, updatey

[bool, default: True] When `True`, update the x/y values.

`update_from_path`(*self*, *path*, *ignore=None*, *updatex=True*, *updatey=True*)
Update the bounds of the *Bbox* to contain the vertices of the provided path. After updating, the bounds will have positive *width* and *height*; *x0* and *y0* will be the minimal values.

Parameters

path

[*Path*]

ignore

[bool, optional]

- when `True`, ignore the existing bounds of the *Bbox*.
- when `False`, include the existing bounds of the *Bbox*.
- when `None`, use the last value passed to *ignore()*.

updatex, updatey

[bool, default: True] When `True`, update the x/y values.

property x0
The first of the pair of *x* coordinates that define the bounding box.

This is not guaranteed to be less than *x1* (for that, use `xmin`).

property x1
The second of the pair of *x* coordinates that define the bounding box.

This is not guaranteed to be greater than *x0* (for that, use `xmax`).

property y0
The first of the pair of *y* coordinates that define the bounding box.

This is not guaranteed to be less than *y1* (for that, use `ymin`).

property y1
The second of the pair of *y* coordinates that define the bounding box.

This is not guaranteed to be greater than *y0* (for that, use `ymax`).

class matplotlib.transforms.BboxBase(*shorthand_name=None*)
Bases: *matplotlib.transforms.TransformNode*

The base class of all bounding boxes.

This class is immutable; *Bbox* is a mutable subclass.

The canonical representation is as two points, with no restrictions on their ordering. Convenience properties are provided to get the left, bottom, right and top edges and width and height, but these are not stored explicitly.

Parameters

shorthand_name

[str] A string representing the "name" of the transform. The name carries no significance other than to improve the readability of `str(transform)` when DEBUG=True.

__array__(*self*, **args*, ***kwargs*)

__module__ = 'matplotlib.transforms'

anchored(*self*, *c*, *container=None*)
Return a copy of the *Bbox* shifted to position *c* within *container*.

Parameters

c

[(float, float) or str] May be either:

- A sequence (*cx*, *cy*) where *cx* and *cy* range from 0 to 1, where 0 is left or bottom and 1 is right or top
- a string: - 'C' for centered - 'S' for bottom-center - 'SE' for bottom-left - 'E' for left - etc.

container

[*Bbox*, optional] The box within which the *Bbox* is positioned; it defaults to the initial *Bbox*.

`property bounds`
Return (*x0*, *y0*, *width*, *height*).

`coefs = {'C': (0.5, 0.5), 'E': (1.0, 0.5), 'N': (0.5, 1.0), 'NE': (1.0, 1.0), 'NW': (0, 1.`

`contains`(*self*, *x*, *y*)
Return whether `(x, y)` is in the bounding box or on its edge.

`containsx`(*self*, *x*)
Return whether *x* is in the closed (*x0*, *x1*) interval.

`containsy`(*self*, *y*)
Return whether *y* is in the closed (*y0*, *y1*) interval.

`corners`(*self*)
Return the corners of this rectangle as an array of points.

Specifically, this returns the array `[[x0, y0], [x0, y1], [x1, y0], [x1, y1]]`.

`count_contains`(*self*, *vertices*)
Count the number of vertices contained in the *Bbox*. Any vertices with a non-finite x or y value are ignored.

Parameters

vertices

[Nx2 Numpy array.]

`count_overlaps`(*self*, *bboxes*)
Count the number of bounding boxes that overlap this one.

Parameters

bboxes

[sequence of *BboxBase*]

`expanded`(*self*, *sw*, *sh*)
Construct a *Bbox* by expanding this one around its center by the factors *sw* and *sh*.

property extents
: Return (*x0*, *y0*, *x1*, *y1*).

frozen(*self*)
: The base class for anything that participates in the transform tree and needs to invalidate its parents or be invalidated. This includes classes that are not really transforms, such as bounding boxes, since some transforms depend on bounding boxes to compute their values.

fully_contains(*self*, *x*, *y*)
: Return whether `x, y` is in the bounding box, but not on its edge.

fully_containsx(*self*, *x*)
: Return whether *x* is in the open (*x0*, *x1*) interval.

fully_containsy(*self*, *y*)
: Return whether *y* is in the open (*y0*, *y1*) interval.

fully_overlaps(*self*, *other*)
: Return whether this bounding box overlaps with the other bounding box, not including the edges.

Parameters

other

[*BboxBase*]

get_points(*self*)

property height
: The (signed) height of the bounding box.

static intersection(*bbox1*, *bbox2*)
: Return the intersection of *bbox1* and *bbox2* if they intersect, or None if they don't.

property intervalx
: The pair of *x* coordinates that define the bounding box.

  This is not guaranteed to be sorted from left to right.

property intervaly
: The pair of *y* coordinates that define the bounding box.

  This is not guaranteed to be sorted from bottom to top.

inverse_transformed(*self*, *transform*)
: [*Deprecated*] Construct a *Bbox* by statically transforming this one by the inverse of *transform*.

Notes

Deprecated since version 3.3.

`is_affine = True`

`is_bbox = True`

`is_unit`(*self*)
[*Deprecated*] Return whether this is the unit box (from (0, 0) to (1, 1)).

Notes

Deprecated since version 3.2.

`property max`
The top-right corner of the bounding box.

`property min`
The bottom-left corner of the bounding box.

`overlaps`(*self*, *other*)
Return whether this bounding box overlaps with the other bounding box.

Parameters

other

[*BboxBase*]

`property p0`
The first pair of (*x*, *y*) coordinates that define the bounding box.

This is not guaranteed to be the bottom-left corner (for that, use *min*).

`property p1`
The second pair of (*x*, *y*) coordinates that define the bounding box.

This is not guaranteed to be the top-right corner (for that, use *max*).

`padded`(*self*, *p*)
Construct a *Bbox* by padding this one on all four sides by *p*.

`rotated`(*self*, *radians*)
Return the axes-aligned bounding box that bounds the result of rotating this *Bbox* by an angle of *radians*.

`shrunk`(*self*, *mx*, *my*)
Return a copy of the *Bbox*, shrunk by the factor *mx* in the *x* direction and the factor *my* in the *y* direction. The lower left corner of the box remains unchanged. Normally *mx* and *my* will be less than 1, but this is not enforced.

`shrunk_to_aspect`(*self*, *box_aspect*, *container=None*, *fig_aspect=1.0*)
Return a copy of the *Bbox*, shrunk so that it is as large as it can be while having the desired aspect ratio, *box_aspect*. If the box coordinates are relative (i.e. fractions of a larger box such as a figure) then the physical aspect ratio of that figure is specified with *fig_aspect*, so that *box_aspect* can also be given as a ratio of the absolute dimensions, not the relative dimensions.

`property size`
The (signed) width and height of the bounding box.

`splitx`(*self*, **args*)
Return a list of new *Bbox* objects formed by splitting the original one with vertical lines at fractional positions given by *args*.

`splity`(*self*, **args*)
Return a list of new *Bbox* objects formed by splitting the original one with horizontal lines at fractional positions given by *args*.

`transformed`(*self*, *transform*)
Construct a *Bbox* by statically transforming this one by *transform*.

`translated`(*self*, *tx*, *ty*)
Construct a *Bbox* by translating this one by *tx* and *ty*.

`static union`(*bboxes*)
Return a *Bbox* that contains all of the given *bboxes*.

`property width`
The (signed) width of the bounding box.

`property x0`
The first of the pair of *x* coordinates that define the bounding box.

This is not guaranteed to be less than *x1* (for that, use *xmin*).

`property x1`
The second of the pair of *x* coordinates that define the bounding box.

This is not guaranteed to be greater than *x0* (for that, use *xmax*).

`property xmax`
The right edge of the bounding box.

`property xmin`
The left edge of the bounding box.

`property y0`
The first of the pair of *y* coordinates that define the bounding box.

This is not guaranteed to be less than *y1* (for that, use *ymin*).

`property y1`
The second of the pair of *y* coordinates that define the bounding box.

This is not guaranteed to be greater than *y0* (for that, use *ymax*).

property ymax
The top edge of the bounding box.

property ymin
The bottom edge of the bounding box.

class matplotlib.transforms.BboxTransform(*boxin*, *boxout*, ***kwargs*)
Bases: *matplotlib.transforms.Affine2DBase*

BboxTransform linearly transforms points from one *Bbox* to another.

Create a new *BboxTransform* that linearly transforms points from *boxin* to *boxout*.

__init__(*self*, *boxin*, *boxout*, ***kwargs*)
Create a new *BboxTransform* that linearly transforms points from *boxin* to *boxout*.

__module__ = 'matplotlib.transforms'

__str__(*self*)
Return str(self).

get_matrix(*self*)
Get the matrix for the affine part of this transform.

is_separable = True

class matplotlib.transforms.BboxTransformFrom(*boxin*, ***kwargs*)
Bases: *matplotlib.transforms.Affine2DBase*

BboxTransformFrom linearly transforms points from a given *Bbox* to the unit bounding box.

Parameters

shorthand_name

[str] A string representing the "name" of the transform. The name carries no significance other than to improve the readability of `str(transform)` when DEBUG=True.

__init__(*self*, *boxin*, ***kwargs*)

Parameters

shorthand_name

[str] A string representing the "name" of the transform. The name carries no significance other than to improve the readability of `str(transform)` when DEBUG=True.

__module__ = 'matplotlib.transforms'

__str__(*self*)
Return str(self).

get_matrix(*self*)
Get the matrix for the affine part of this transform.

is_separable = True

class matplotlib.transforms.BboxTransformTo(*boxout*, ***kwargs*)
Bases: *matplotlib.transforms.Affine2DBase*

BboxTransformTo is a transformation that linearly transforms points from the unit bounding box to a given *Bbox*.

Create a new *BboxTransformTo* that linearly transforms points from the unit bounding box to *boxout*.

__init__(*self*, *boxout*, ***kwargs*)
Create a new *BboxTransformTo* that linearly transforms points from the unit bounding box to *boxout*.

__module__ = 'matplotlib.transforms'

__str__(*self*)
Return str(self).

get_matrix(*self*)
Get the matrix for the affine part of this transform.

is_separable = True

class matplotlib.transforms.BboxTransformToMaxOnly(*boxout*, ***kwargs*)
Bases: *matplotlib.transforms.BboxTransformTo*

BboxTransformTo is a transformation that linearly transforms points from the unit bounding box to a given *Bbox* with a fixed upper left of (0, 0).

Create a new *BboxTransformTo* that linearly transforms points from the unit bounding box to *boxout*.

__module__ = 'matplotlib.transforms'

get_matrix(*self*)
Get the matrix for the affine part of this transform.

class matplotlib.transforms.BlendedAffine2D(*x_transform*, *y_transform*, ***kwargs*)
Bases: matplotlib.transforms._BlendedMixin, *matplotlib.transforms.Affine2DBase*

A "blended" transform uses one transform for the *x*-direction, and another transform for the *y*-direction.

This version is an optimization for the case where both child transforms are of type *Affine2DBase*.

Create a new "blended" transform using *x_transform* to transform the *x*-axis and *y_transform* to transform the *y*-axis.

Both *x_transform* and *y_transform* must be 2D affine transforms.

You will generally not call this constructor directly but use the *blended_transform_factory* function instead, which can determine automatically which kind of blended transform to create.

__init__(*self*, *x_transform*, *y_transform*, ***kwargs*)
Create a new "blended" transform using *x_transform* to transform the *x*-axis and *y_transform* to transform the *y*-axis.

Both *x_transform* and *y_transform* must be 2D affine transforms.

You will generally not call this constructor directly but use the *blended_transform_factory* function instead, which can determine automatically which kind of blended transform to create.

__module__ = 'matplotlib.transforms'

get_matrix(*self*)
Get the matrix for the affine part of this transform.

is_separable = True

class matplotlib.transforms.BlendedGenericTransform(*x_transform*, *y_transform*, ***kwargs*)
Bases: matplotlib.transforms._BlendedMixin, *matplotlib.transforms.Transform*

A "blended" transform uses one transform for the *x*-direction, and another transform for the *y*-direction.

This "generic" version can handle any given child transform in the *x*- and *y*-directions.

Create a new "blended" transform using *x_transform* to transform the *x*-axis and *y_transform* to transform the *y*-axis.

You will generally not call this constructor directly but use the *blended_transform_factory* function instead, which can determine automatically which kind of blended transform to create.

__init__(*self*, *x_transform*, *y_transform*, ***kwargs*)
Create a new "blended" transform using *x_transform* to transform the *x*-axis and *y_transform* to transform the *y*-axis.

You will generally not call this constructor directly but use the *blended_transform_factory* function instead, which can determine automatically which kind of blended transform to create.

__module__ = 'matplotlib.transforms'

contains_branch(*self*, *other*)
Return whether the given transform is a sub-tree of this transform.

This routine uses transform equality to identify sub-trees, therefore in many situations it is object id which will be used.

For the case where the given transform represents the whole of this transform, returns True.

property depth
: Return the number of transforms which have been chained together to form this Transform instance.

  Note: For the special case of a Composite transform, the maximum depth of the two is returned.

frozen(*self*)
: Return a frozen copy of this transform node. The frozen copy will not be updated when its children change. Useful for storing a previously known state of a transform where `copy.deepcopy()` might normally be used.

get_affine(*self*)
: Get the affine part of this transform.

property has_inverse
: bool(x) -> bool

  Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

input_dims = 2

inverted(*self*)
: Return the corresponding inverse transformation.

  It holds `x == self.inverted().transform(self.transform(x))`.

  The return value of this method should be treated as temporary. An update to *self* does not cause a corresponding update to its inverted copy.

property is_affine
: bool(x) -> bool

  Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

is_separable = True

output_dims = 2

pass_through = True

transform_non_affine(*self*, *points*)
: Apply only the non-affine part of this transformation.

  `transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

  In non-affine transformations, this is generally equivalent to `transform(values)`. In affine transformations, this is always a no-op.

  Parameters

values

[array] The input values as NumPy array of length *input_dims* or shape (N x *input_dims*).

Returns

array

The output values as NumPy array of length *input_dims* or shape (N x *output_dims*), depending on the input.

class matplotlib.transforms.**CompositeAffine2D**(*a*, *b*, ***kwargs*)

Bases: *matplotlib.transforms.Affine2DBase*

A composite transform formed by applying transform *a* then transform *b*.

This version is an optimization that handles the case where both *a* and *b* are 2D affines.

Create a new composite transform that is the result of applying *Affine2DBase* *a* then *Affine2DBase* *b*.

You will generally not call this constructor directly but write `a + b` instead, which will automatically choose the best kind of composite transform instance to create.

__init__(*self*, *a*, *b*, ***kwargs*)

Create a new composite transform that is the result of applying *Affine2DBase* *a* then *Affine2DBase* *b*.

You will generally not call this constructor directly but write `a + b` instead, which will automatically choose the best kind of composite transform instance to create.

__module__ = 'matplotlib.transforms'

__str__(*self*)

Return str(self).

property **depth**

Return the number of transforms which have been chained together to form this Transform instance.

Note: For the special case of a Composite transform, the maximum depth of the two is returned.

---

get_matrix(*self*)

Get the matrix for the affine part of this transform.

class matplotlib.transforms.**CompositeGenericTransform**(*a*, *b*, ***kwargs*)

Bases: *matplotlib.transforms.Transform*

A composite transform formed by applying transform *a* then transform *b*.

This "generic" version can handle any two arbitrary transformations.

Create a new composite transform that is the result of applying transform *a* then transform *b*.

You will generally not call this constructor directly but write `a + b` instead, which will automatically choose the best kind of composite transform instance to create.

`__eq__`(*self*, *other*)
Return self==value.

`__hash__ = None`

`__init__`(*self*, *a*, *b*, ***kwargs*)
Create a new composite transform that is the result of applying transform *a* then transform *b*.

You will generally not call this constructor directly but write `a + b` instead, which will automatically choose the best kind of composite transform instance to create.

`__module__ = 'matplotlib.transforms'`

`__str__`(*self*)
Return str(self).

`property depth`
Return the number of transforms which have been chained together to form this Transform instance.

Note: For the special case of a Composite transform, the maximum depth of the two is returned.

`frozen`(*self*)
Return a frozen copy of this transform node. The frozen copy will not be updated when its children change. Useful for storing a previously known state of a transform where `copy.deepcopy()` might normally be used.

`get_affine`(*self*)
Get the affine part of this transform.

`property has_inverse`
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

`inverted`(*self*)
Return the corresponding inverse transformation.

It holds `x == self.inverted().transform(self.transform(x))`.

The return value of this method should be treated as temporary. An update to *self* does not cause a corresponding update to its inverted copy.

`property is_affine`
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

`property is_separable`
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

`pass_through = True`

`transform_affine`(*self*, *points*)
Apply only the affine part of this transformation on the given array of values.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally a no-op. In affine transformations, this is equivalent to `transform(values)`.

Parameters

values

[array] The input values as NumPy array of length `input_dims` or shape (N x `input_dims`).

Returns

array

The output values as NumPy array of length `input_dims` or shape (N x `output_dims`), depending on the input.

`transform_non_affine`(*self*, *points*)
Apply only the non-affine part of this transformation.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally equivalent to `transform(values)`. In affine transformations, this is always a no-op.

Parameters

values

[array] The input values as NumPy array of length `input_dims` or shape (N x `input_dims`).

Returns

array

The output values as NumPy array of length `input_dims` or shape (N x `output_dims`), depending on the input.

`transform_path_non_affine`(*self*, *path*)

Apply the non-affine part of this transform to *Path* *path*, returning a new *Path*.

`transform_path(path)` is equivalent to `transform_path_affine(transform_path_non_affine(va`

`class matplotlib.transforms.IdentityTransform`(**args*, ***kwargs*)

Bases: *matplotlib.transforms.Affine2DBase*

A special class that does one thing, the identity transform, in a fast way.

Parameters

shorthand_name

[str] A string representing the "name" of the transform. The name carries no significance other than to improve the readability of `str(transform)` when DEBUG=True.

`__module__ = 'matplotlib.transforms'`

`__str__`(*self*)

Return str(self).

`frozen`(*self*)

Return a frozen copy of this transform node. The frozen copy will not be updated when its children change. Useful for storing a previously known state of a transform where `copy.deepcopy()` might normally be used.

`get_affine`(*self*)

Get the affine part of this transform.

`get_matrix`(*self*)

Get the matrix for the affine part of this transform.

`inverted`(*self*)

Return the corresponding inverse transformation.

It holds `x == self.inverted().transform(self.transform(x))`.

The return value of this method should be treated as temporary. An update to *self* does not cause a corresponding update to its inverted copy.

`transform`(*self*, *points*)

Apply this transformation on the given array of *values*.

Parameters

values

[array] The input values as NumPy array of length `input_dims` or shape (N x `input_dims`).

Returns

array

The output values as NumPy array of length `input_dims` or shape (N x `output_dims`), depending on the input.

`transform_affine`(*self*, *points*)

Apply only the affine part of this transformation on the given array of values.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally a no-op. In affine transformations, this is equivalent to `transform(values)`.

Parameters

values

[array] The input values as NumPy array of length `input_dims` or shape (N x `input_dims`).

Returns

array

The output values as NumPy array of length `input_dims` or shape (N x `output_dims`), depending on the input.

`transform_non_affine`(*self*, *points*)

Apply only the non-affine part of this transformation.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally equivalent to `transform(values)`. In affine transformations, this is always a no-op.

Parameters

values

[array] The input values as NumPy array of length `input_dims` or shape (N x `input_dims`).

Returns

array

The output values as NumPy array of length `input_dims` or shape (N x `output_dims`), depending on the input.

`transform_path`(*self*, *path*)
Apply the transform to *Path* *path*, returning a new *Path*.

In some cases, this transform may insert curves into the path that began as line segments.

`transform_path_affine`(*self*, *path*)
Apply the affine part of this transform to *Path* *path*, returning a new *Path*.

`transform_path(path)` is equivalent to `transform_path_affine(transform_path_non_affine(va`

`transform_path_non_affine`(*self*, *path*)
Apply the non-affine part of this transform to *Path* *path*, returning a new *Path*.

`transform_path(path)` is equivalent to `transform_path_affine(transform_path_non_affine(va`

`class matplotlib.transforms.LockableBbox`(*bbox*, *x0=None*, *y0=None*, *x1=None*, *y1=None*, ***kwargs*)

Bases: *matplotlib.transforms.BboxBase*

A *Bbox* where some elements may be locked at certain values.

When the child bounding box changes, the bounds of this bbox will update accordingly with the exception of the locked elements.

Parameters

bbox

[*Bbox*] The child bounding box to wrap.

x0

[float or None] The locked value for x0, or None to leave unlocked.

y0

[float or None] The locked value for y0, or None to leave unlocked.

x1

[float or None] The locked value for x1, or None to leave unlocked.

y1

[float or None] The locked value for y1, or None to leave unlocked.

`__init__`(*self*, *bbox*, *x0=None*, *y0=None*, *x1=None*, *y1=None*, ***kwargs*)

Parameters

bbox

[*Bbox*] The child bounding box to wrap.

x0

[float or None] The locked value for x0, or None to leave unlocked.

y0

[float or None] The locked value for y0, or None to leave unlocked.

x1

[float or None] The locked value for x1, or None to leave unlocked.

y1

[float or None] The locked value for y1, or None to leave unlocked.

`__module__ = 'matplotlib.transforms'`

`__str__`(*self*)
Return str(self).

`get_points`(*self*)

`property locked_x0`
float or None: The value used for the locked x0.

`property locked_x1`
float or None: The value used for the locked x1.

`property locked_y0`
float or None: The value used for the locked y0.

`property locked_y1`
float or None: The value used for the locked y1.

`class matplotlib.transforms.ScaledTranslation`(*xt*, *yt*, *scale_trans*, ***kwargs*)
Bases: *matplotlib.transforms.Affine2DBase*

A transformation that translates by *xt* and *yt*, after *xt* and *yt* have been transformed by *scale_trans*.

Parameters

shorthand_name

[str] A string representing the "name" of the transform. The name carries no significance other than to improve the readability of `str(transform)` when DEBUG=True.

`__init__`(*self*, *xt*, *yt*, *scale_trans*, ***kwargs*)

Parameters

shorthand_name

[str] A string representing the "name" of the transform. The name carries no significance other than to improve the readability of `str(transform)` when DEBUG=True.

`__module__ = 'matplotlib.transforms'`

`__str__`(*self*)
Return str(self).

`get_matrix`(*self*)
Get the matrix for the affine part of this transform.

`class matplotlib.transforms.Transform`(*shorthand_name=None*)
Bases: *matplotlib.transforms.TransformNode*

The base class of all *TransformNode* instances that actually perform a transformation.

All non-affine transformations should be subclasses of this class. New affine transformations should be subclasses of *Affine2D*.

Subclasses of this class should override the following members (at minimum):

- *input_dims*
- *output_dims*
- *transform()*
- *inverted()* (if an inverse exists)

The following attributes may be overridden if the default is unsuitable:

- *is_separable* (defaults to True for 1d -> 1d transforms, False otherwise)
- *has_inverse* (defaults to True if *inverted()* is overridden, False otherwise)

If the transform needs to do something non-standard with *matplotlib.path.Path* objects, such as adding curves where there were once line segments, it should override:

- *transform_path()*

Parameters

shorthand_name

[str] A string representing the "name" of the transform. The name carries no significance other than to improve the readability of `str(transform)` when DEBUG=True.

`__add__`(*self*, *other*)
Compose two transforms together so that *self* is followed by *other*.

`A + B` returns a transform `C` so that `C.transform(x) == B.transform(A.transform(x))`.

`__array__`(*self*, **args*, ***kwargs*)
Array interface to get at this Transform's affine matrix.

`classmethod __init_subclass__`()
This method is called when a class is subclassed.

The default implementation does nothing. It may be overridden to extend subclasses.

`__module__ = 'matplotlib.transforms'`

`__sub__`(*self*, *other*)
Compose *self* with the inverse of *other*, cancelling identical terms if any:

```
# In general:
A - B == A + B.inverted()
# (but see note regarding frozen transforms below).

# If A "ends with" B (i.e. A == A' + B for some A') we can cancel
# out B:
(A' + B) - B == A'

# Likewise, if B "starts with" A (B = A + B'), we can cancel out A:
A - (A + B') == B'.inverted() == B'^-1
```

Cancellation (rather than naively returning `A + B.inverted()`) is important for multiple reasons:

- It avoids floating-point inaccuracies when computing the inverse of B: `B - B` is guaranteed to cancel out exactly (resulting in the identity transform), whereas `B + B.inverted()` may differ by a small epsilon.
- `B.inverted()` always returns a frozen transform: if one computes `A + B + B.inverted()` and later mutates `B`, then `B.inverted()` won't be updated and the last two terms won't cancel out anymore; on the other hand, `A + B - B` will always be equal to `A` even if `B` is mutated.

`contains_branch`(*self*, *other*)
Return whether the given transform is a sub-tree of this transform.

This routine uses transform equality to identify sub-trees, therefore in many situations it is object id which will be used.

For the case where the given transform represents the whole of this transform, returns True.

`contains_branch_seperately`(*self*, *other_transform*)
Return whether the given branch is a sub-tree of this transform on each separate dimension.

A common use for this method is to identify if a transform is a blended transform containing an axes' data transform. e.g.:

```
x_isdata, y_isdata = trans.contains_branch_seperately(ax.transData)
```

`property depth`
Return the number of transforms which have been chained together to form this Transform instance.

Note: For the special case of a Composite transform, the maximum depth of the two is returned.

`get_affine`(*self*)
Get the affine part of this transform.

`get_matrix`(*self*)
Get the matrix for the affine part of this transform.

`has_inverse = False`
True if this transform has a corresponding inverse transform.

`input_dims = None`
The number of input dimensions of this transform. Must be overridden (with integers) in the subclass.

`inverted`(*self*)
Return the corresponding inverse transformation.

It holds `x == self.inverted().transform(self.transform(x))`.

The return value of this method should be treated as temporary. An update to *self* does not cause a corresponding update to its inverted copy.

`is_separable = False`
True if this transform is separable in the x- and y- dimensions.

`output_dims = None`
The number of output dimensions of this transform. Must be overridden (with integers) in the subclass.

`transform`(*self*, *values*)
Apply this transformation on the given array of *values*.

Parameters

values

[array] The input values as NumPy array of length *input_dims* or shape (N x *input_dims*).

Returns

array

The output values as NumPy array of length *input_dims* or shape (N x *output_dims*), depending on the input.

`transform_affine`(*self*, *values*)

Apply only the affine part of this transformation on the given array of values.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally a no-op. In affine transformations, this is equivalent to `transform(values)`.

Parameters

values

[array] The input values as NumPy array of length *input_dims* or shape (N x *input_dims*).

Returns

array

The output values as NumPy array of length *input_dims* or shape (N x *output_dims*), depending on the input.

`transform_angles`(*self*, *angles*, *pts*, *radians=False*, *pushoff=1e-05*)

Transform a set of angles anchored at specific locations.

Parameters

angles

[(N,) array-like] The angles to transform.

pts

[(N, 2) array-like] The points where the angles are anchored.

radians

[bool, default: False] Whether *angles* are radians or degrees.

pushoff

[float] For each point in *pts* and angle in *angles*, the transformed angle is computed by transforming a segment of length *pushoff* starting at that point and making that angle relative

to the horizontal axis, and measuring the angle between the horizontal axis and the transformed segment.

Returns

(N,) array

`transform_bbox`(*self*, *bbox*)
Transform the given bounding box.

For smarter transforms including caching (a common requirement in Matplotlib), see *TransformedBbox*.

`transform_non_affine`(*self*, *values*)
Apply only the non-affine part of this transformation.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`

In non-affine transformations, this is generally equivalent to `transform(values)`. In affine transformations, this is always a no-op.

Parameters

values

[array] The input values as NumPy array of length *input_dims* or shape (N x *input_dims*).

Returns

array

The output values as NumPy array of length *input_dims* or shape (N x *output_dims*), depending on the input.

`transform_path`(*self*, *path*)
Apply the transform to *Path* *path*, returning a new *Path*.

In some cases, this transform may insert curves into the path that began as line segments.

`transform_path_affine`(*self*, *path*)
Apply the affine part of this transform to *Path* *path*, returning a new *Path*.

`transform_path(path)` is equivalent to `transform_path_affine(transform_path_non_affine(va`

`transform_path_non_affine`(*self*, *path*)
Apply the non-affine part of this transform to *Path* *path*, returning a new *Path*.

`transform_path(path)` is equivalent to `transform_path_affine(transform_path_non_affine(va`

transform_point(*self*, *point*)
Return a transformed point.

This function is only kept for backcompatibility; the more general *transform* method is capable of transforming both a list of points and a single point.

The point is given as a sequence of length *input_dims*. The transformed point is returned as a sequence of length *output_dims*.

class matplotlib.transforms.TransformNode(*shorthand_name=None*)
Bases: object

The base class for anything that participates in the transform tree and needs to invalidate its parents or be invalidated. This includes classes that are not really transforms, such as bounding boxes, since some transforms depend on bounding boxes to compute their values.

Parameters

shorthand_name

[str] A string representing the "name" of the transform. The name carries no significance other than to improve the readability of `str(transform)` when DEBUG=True.

INVALID = 3

INVALID_AFFINE = 2

INVALID_NON_AFFINE = 1

__copy__(*self*, **args*)

__deepcopy__(*self*, **args*)

__dict__ = mappingproxy({'__module__': 'matplotlib.transforms', '__doc__': '\n The base c

__getstate__(*self*)

__init__(*self*, *shorthand_name=None*)

Parameters

shorthand_name

[str] A string representing the "name" of the transform. The name carries no significance other than to improve the readability of `str(transform)` when DEBUG=True.

__module__ = 'matplotlib.transforms'

__setstate__(*self*, *data_dict*)

__weakref__
list of weak references to the object (if defined)

frozen(*self*)
Return a frozen copy of this transform node. The frozen copy will not be updated when its children change. Useful for storing a previously known state of a transform where `copy.deepcopy()` might normally be used.

invalidate(*self*)
Invalidate this *TransformNode* and triggers an invalidation of its ancestors. Should be called any time the transform changes.

is_affine = False

is_bbox = False

pass_through = False
If pass_through is True, all ancestors will always be invalidated, even if 'self' is already invalid.

set_children(*self*, **children*)
Set the children of the transform, to let the invalidation system know which transforms can invalidate this transform. Should be called from the constructor of any transforms that depend on other transforms.

class matplotlib.transforms.**TransformWrapper**(*child*)
Bases: *matplotlib.transforms.Transform*

A helper class that holds a single child transform and acts equivalently to it.

This is useful if a node of the transform tree must be replaced at run time with a transform of a different type. This class allows that replacement to correctly trigger invalidation.

TransformWrapper instances must have the same input and output dimensions during their entire lifetime, so the child transform may only be replaced with another child transform of the same dimensions.

child: A *Transform* instance. This child may later be replaced with *set()*.

__eq__(*self*, *other*)
Return self==value.

__hash__ = None

__init__(*self*, *child*)
child: A *Transform* instance. This child may later be replaced with *set()*.

__module__ = 'matplotlib.transforms'

__str__(*self*)
Return str(self).

frozen(*self*)
Return a frozen copy of this transform node. The frozen copy will not be updated when its children change. Useful for storing a previously known state of a transform where `copy.deepcopy()` might normally be used.

property has_inverse
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

property is_affine
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

property is_separable
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

pass_through = True

set(*self*, *child*)
Replace the current child of this transform with another one.

The new child must have the same number of input and output dimensions as the current child.

class matplotlib.transforms.TransformedBbox(*bbox*, *transform*, ***kwargs*)
Bases: *matplotlib.transforms.BboxBase*

A *Bbox* that is automatically transformed by a given transform. When either the child bounding box or transform changes, the bounds of this bbox will update accordingly.

Parameters

bbox

[*Bbox*]

transform

[*Transform*]

__init__(*self*, *bbox*, *transform*, ***kwargs*)

Parameters

bbox

[*Bbox*]

transform

[*Transform*]

__module__ = 'matplotlib.transforms'

__str__(*self*)
Return str(self).

get_points(*self*)

class matplotlib.transforms.**TransformedPatchPath**(*patch*)
Bases: *matplotlib.transforms.TransformedPath*

A *TransformedPatchPath* caches a non-affine transformed copy of the *Patch*. This cached copy is automatically updated when the non-affine part of the transform or the patch changes.

Parameters

patch

[*Patch*]

__init__(*self*, *patch*)

Parameters

patch

[*Patch*]

__module__ = 'matplotlib.transforms'

class matplotlib.transforms.**TransformedPath**(*path*, *transform*)
Bases: *matplotlib.transforms.TransformNode*

A *TransformedPath* caches a non-affine transformed copy of the *Path*. This cached copy is automatically updated when the non-affine part of the transform changes.

Note: Paths are considered immutable by this class. Any update to the path's vertices/codes will not trigger a transform recomputation.

Parameters

path

[*Path*]

transform

[*Transform*]

__init__(*self*, *path*, *transform*)

Parameters

path

[*Path*]

transform

[*Transform*]

`__module__ = 'matplotlib.transforms'`

`get_affine`(*self*)

`get_fully_transformed_path`(*self*)
Return a fully-transformed copy of the child path.

`get_transformed_path_and_affine`(*self*)
Return a copy of the child path, with the non-affine part of the transform already applied, along with the affine part of the path necessary to complete the transformation.

`get_transformed_points_and_affine`(*self*)
Return a copy of the child path, with the non-affine part of the transform already applied, along with the affine part of the path necessary to complete the transformation. Unlike *get_transformed_path_and_affine()*, no interpolation will be performed.

`matplotlib.transforms.blended_transform_factory`(*x_transform*, *y_transform*)
Create a new "blended" transform using *x_transform* to transform the *x*-axis and *y_transform* to transform the *y*-axis.

A faster version of the blended transform is returned for the case where both child transforms are affine.

`matplotlib.transforms.composite_transform_factory`(*a*, *b*)
Create a new composite transform that is the result of applying transform a then transform b.

Shortcut versions of the blended transform are provided for the case where both child transforms are affine, or one or the other is the identity transform.

Composite transforms may also be created using the '+' operator, e.g.:

```
c = a + b
```

`matplotlib.transforms.interval_contains`(*interval*, *val*)
Check, inclusively, whether an interval includes a given value.

Parameters

interval

[(float, float)] The endpoints of the interval.

val

[float] Value to check is within interval.

Returns

bool

Whether *val* is within the *interval*.

matplotlib.transforms.**interval_contains_open**(*interval*, *val*)

Check, excluding endpoints, whether an interval includes a given value.

Parameters

interval

[(float, float)] The endpoints of the interval.

val

[float] Value to check is within interval.

Returns

bool

Whether *val* is within the *interval*.

matplotlib.transforms.**nonsingular**(*vmin*, *vmax*, *expander=0.001*, *tiny=1e-15*, *increasing=True*)

Modify the endpoints of a range as needed to avoid singularities.

Parameters

vmin, vmax

[float] The initial endpoints.

expander

[float, default: 0.001] Fractional amount by which *vmin* and *vmax* are expanded if the original interval is too small, based on *tiny*.

tiny

[float, default: 1e-15] Threshold for the ratio of the interval to the maximum absolute value of its endpoints. If the interval is smaller than this, it will be expanded. This value should be around 1e-15 or larger; otherwise the interval will be approaching the double precision resolution limit.

increasing

[bool, default: True] If True, swap *vmin*, *vmax* if *vmin* > *vmax*.

Returns

vmin, vmax

[float] Endpoints, expanded and/or swapped if necessary. If either input is inf or NaN, or if both inputs are 0 or very close to zero, it returns -*expander*, *expander*.

matplotlib.transforms.**offset_copy**(*trans*, *fig=None*, *x=0.0*, *y=0.0*, *units='inches'*)

Return a new transform with an added offset.

Parameters

trans

[*Transform* subclass] Any transform, to which offset will be applied.

fig

[*Figure*, default: None] Current figure. It can be None if *units* are 'dots'.

x, y

[float, default: 0.0] The offset to apply.

units

[{'inches', 'points', 'dots'}, default: 'inches'] Units of the offset.

Returns

Transform **subclass**

Transform with applied offset.

18.56 matplotlib.tri

Unstructured triangular grid functions.

class matplotlib.tri.**Triangulation**(*x*, *y*, *triangles=None*, *mask=None*)

An unstructured triangular grid consisting of npoints points and ntri triangles. The triangles can either be specified by the user or automatically generated using a Delaunay triangulation.

Parameters

x, y

[array-like of shape (npoints)] Coordinates of grid points.

triangles

[int array-like of shape (ntri, 3), optional] For each triangle, the indices of the three points that make up the triangle, ordered in an anticlockwise manner. If not specified, the Delaunay triangulation is calculated.

mask

[bool array-like of shape (ntri), optional] Which triangles are masked out.

Notes

For a Triangulation to be valid it must not have duplicate points, triangles formed from colinear points, or overlapping triangles.

Attributes

edges

[int array of shape (nedges, 2)] Return integer array of shape (nedges, 2) containing all edges of non-masked triangles.

neighbors

[int array of shape (ntri, 3)] Return integer array of shape (ntri, 3) containing neighbor triangles.

mask

[bool array of shape (ntri, 3)] Masked out triangles.

is_delaunay

[bool] Whether the Triangulation is a calculated Delaunay triangulation (where *triangles* was not specified) or not.

`calculate_plane_coefficients`(*self*, *z*)

Calculate plane equation coefficients for all unmasked triangles from the point (x, y) coordinates and specified z-array of shape (npoints). The returned array has shape (npoints, 3) and allows z-value at (x, y) position in triangle tri to be calculated using `z = array[tri, 0] * x + array[tri, 1] * y + array[tri, 2]`.

`property edges`

Return integer array of shape (nedges, 2) containing all edges of non-masked triangles.

Each row defines an edge by it's start point index and end point index. Each edge appears only once, i.e. for an edge between points *i* and *j*, there will only be either *(i, j)* or *(j, i)*.

`get_cpp_triangulation`(*self*)
Return the underlying C++ Triangulation object, creating it if necessary.

`static get_from_args_and_kwargs`(**args*, ***kwargs*)
Return a Triangulation object from the args and kwargs, and the remaining args and kwargs with the consumed values removed.

There are two alternatives: either the first argument is a Triangulation object, in which case it is returned, or the args and kwargs are sufficient to create a new Triangulation to return. In the latter case, see Triangulation.__init__ for the possible args and kwargs.

`get_masked_triangles`(*self*)
Return an array of triangles that are not masked.

`get_trifinder`(*self*)
Return the default *matplotlib.tri.TriFinder* of this triangulation, creating it if necessary. This allows the same TriFinder object to be easily shared.

`property neighbors`
Return integer array of shape (ntri, 3) containing neighbor triangles.

For each triangle, the indices of the three triangles that share the same edges, or -1 if there is no such neighboring triangle. `neighbors[i, j]` is the triangle that is the neighbor to the edge from point index `triangles[i, j]` to point index `triangles[i, (j+1)%3]`.

`set_mask`(*self*, *mask*)
Set or clear the mask array.

Parameters

mask

[None or bool array of length ntri]

`class matplotlib.tri.TriContourSet`(*ax*, **args*, ***kwargs*)
Bases: *matplotlib.contour.ContourSet*

Create and store a set of contour lines or filled regions for a triangular grid.

User-callable method: clabel

Attributes

ax

The axes object in which the contours are drawn.

collections

A silent_list of LineCollections or PolyCollections.

levels

Contour levels.

layers

Same as levels for line contours; half-way between levels for filled contours. See `_process_colors()`.

Draw triangular grid contour lines or filled regions, depending on whether keyword arg 'filled' is False (default) or True.

The first argument of the initializer must be an axes object. The remaining arguments and keyword arguments are described in the docstring of *tricontour*.

class matplotlib.tri.**TriFinder**(*triangulation*)

Abstract base class for classes used to find the triangles of a Triangulation in which (x, y) points lie.

Rather than instantiate an object of a class derived from TriFinder, it is usually better to use the function *Triangulation.get_trifinder*.

Derived classes implement __call__(x, y) where x and y are array-like point coordinates of the same shape.

class matplotlib.tri.**TrapezoidMapTriFinder**(*triangulation*)

Bases: `matplotlib.tri.trifinder.TriFinder`

TriFinder class implemented using the trapezoid map algorithm from the book "Computational Geometry, Algorithms and Applications", second edition, by M. de Berg, M. van Kreveld, M. Overmars and O. Schwarzkopf.

The triangulation must be valid, i.e. it must not have duplicate points, triangles formed from colinear points, or overlapping triangles. The algorithm has some tolerance to triangles formed from colinear points, but this should not be relied upon.

class matplotlib.tri.**TriInterpolator**(*triangulation*, *z*, *trifinder=None*)

Abstract base class for classes used to interpolate on a triangular grid.

Derived classes implement the following methods:

- `__call__(x, y)`, where x, y are array-like point coordinates of the same shape, and that returns a masked array of the same shape containing the interpolated z-values.
- `gradient(x, y)`, where x, y are array-like point coordinates of the same shape, and that returns a list of 2 masked arrays of the same shape containing the 2 derivatives of the interpolator (derivatives of interpolated z values with respect to x and y).

class matplotlib.tri.**LinearTriInterpolator**(*triangulation*, *z*, *trifinder=None*)

Bases: `matplotlib.tri.triinterpolate.TriInterpolator`

Linear interpolator on a triangular grid.

Each triangle is represented by a plane so that an interpolated value at point (x, y) lies on the plane of the triangle containing (x, y). Interpolated values

are therefore continuous across the triangulation, but their first derivatives are discontinuous at edges between triangles.

Parameters

triangulation

[*Triangulation*] The triangulation to interpolate over.

z

[array-like of shape (npoints,)] Array of values, defined at grid points, to interpolate between.

trifinder

[*TriFinder*, optional] If this is not specified, the Triangulation's default TriFinder will be used by calling *Triangulation.get_trifinder*.

Methods

`__call__` (x, y)	(Returns interpolated values at (x, y) points.)
`gradient` (x, y)	(Returns interpolated derivatives at (x, y) points.)

`gradient`(*self*, *x*, *y*)

Returns a list of 2 masked arrays containing interpolated derivatives at the specified (x, y) points.

Parameters

x, y

[array-like] x and y coordinates of the same shape and any number of dimensions.

Returns

dzdx, dzdy

[np.ma.array] 2 masked arrays of the same shape as *x* and *y*; values corresponding to (x, y) points outside of the triangulation are masked out. The first returned array contains the values of $\frac{\partial z}{\partial x}$ and the second those of $\frac{\partial z}{\partial y}$.

`class matplotlib.tri.CubicTriInterpolator`(*triangulation*, *z*, *kind='min_E'*, *trifinder=None*, *dz=None*)

Bases: `matplotlib.tri.triinterpolate.TriInterpolator`

Cubic interpolator on a triangular grid.

In one-dimension - on a segment - a cubic interpolating function is defined by the values of the function and its derivative at both ends. This is almost the same in 2-d inside a triangle, except that the values of the function and its 2 derivatives have to be defined at each triangle node.

The CubicTriInterpolator takes the value of the function at each node - provided by the user - and internally computes the value of the derivatives, resulting in a smooth interpolation. (As a special feature, the user can also impose the value of the derivatives at each node, but this is not supposed to be the common usage.)

Parameters

triangulation

[*Triangulation*] The triangulation to interpolate over.

z

[array-like of shape (npoints,)] Array of values, defined at grid points, to interpolate between.

kind

[{'min_E', 'geom', 'user'}, optional] Choice of the smoothing algorithm, in order to compute the interpolant derivatives (defaults to 'min_E'):

- if 'min_E': (default) The derivatives at each node is computed to minimize a bending energy.
- if 'geom': The derivatives at each node is computed as a weighted average of relevant triangle normals. To be used for speed optimization (large grids).
- if 'user': The user provides the argument *dz*, no computation is hence needed.

trifinder

[*TriFinder*, optional] If not specified, the Triangulation's default TriFinder will be used by calling *Triangulation.get_trifinder*.

dz

[tuple of array-likes (dzdx, dzdy), optional] Used only if *kind* ='user'. In this case *dz* must be provided as (dzdx, dzdy) where dzdx, dzdy are arrays of the same shape as *z* and are the interpolant first derivatives at the *triangulation* points.

Notes

This note is a bit technical and details how the cubic interpolation is computed.

The interpolation is based on a Clough-Tocher subdivision scheme of the *triangulation* mesh (to make it clearer, each triangle of the grid will be divided in 3 child-triangles, and on each child triangle the interpolated function is a cubic polynomial of the 2 coordinates). This technique originates from FEM (Finite Element Method) analysis; the element used is a reduced Hsieh-Clough-Tocher (HCT) element. Its shape functions are described in [1]. The assembled function is guaranteed to be C1-smooth, i.e. it is continuous and its first derivatives are also continuous (this is easy to show inside the triangles but is also true when crossing the edges).

In the default case (*kind* ='min_E'), the interpolant minimizes a curvature energy on the functional space generated by the HCT element shape functions - with imposed values but arbitrary derivatives at each node. The minimized functional is the integral of the so-called total curvature (implementation based on an algorithm from [2] - PCG sparse solver):

$$E(z) = \frac{1}{2} \int_{\Omega} \left(\left(\frac{\partial^2 z}{\partial x^2} \right)^2 + \left(\frac{\partial^2 z}{\partial y^2} \right)^2 + 2 \left(\frac{\partial^2 z}{\partial y \partial x} \right)^2 \right) dx\, dy$$

If the case *kind* ='geom' is chosen by the user, a simple geometric approximation is used (weighted average of the triangle normal vectors), which could improve speed on very large grids.

References

[1], [2]

Methods

`__call__` (x, y)	(Returns interpolated values at (x, y) points.)
`gradient` (x, y)	(Returns interpolated derivatives at (x, y) points.)

`gradient`(*self*, *x*, *y*)

Returns a list of 2 masked arrays containing interpolated derivatives at the specified (x, y) points.

Parameters

x, y

[array-like] x and y coordinates of the same shape and any number of dimensions.

Returns

dzdx, dzdy

[np.ma.array] 2 masked arrays of the same shape as *x* and *y*; values corresponding to (x, y) points outside of the triangulation are masked out. The first returned array contains the values of $\frac{\partial z}{\partial x}$ and the second those of $\frac{\partial z}{\partial y}$.

`class matplotlib.tri.TriRefiner(`*triangulation*`)`

Abstract base class for classes implementing mesh refinement.

A TriRefiner encapsulates a Triangulation object and provides tools for mesh refinement and interpolation.

Derived classes must implement:

- `refine_triangulation(return_tri_index=False, **kwargs)` , where the optional keyword arguments *kwargs* are defined in each TriRefiner concrete implementation, and which returns:
  - a refined triangulation,
  - optionally (depending on *return_tri_index*), for each point of the refined triangulation: the index of the initial triangulation triangle to which it belongs.
- `refine_field(z, triinterpolator=None, **kwargs)`, where:
  - *z* array of field values (to refine) defined at the base triangulation nodes,
  - *triinterpolator* is an optional *TriInterpolator*,
  - the other optional keyword arguments *kwargs* are defined in each TriRefiner concrete implementation;

  and which returns (as a tuple) a refined triangular mesh and the interpolated values of the field at the refined triangulation nodes.

`class matplotlib.tri.UniformTriRefiner(`*triangulation*`)`

Bases: `matplotlib.tri.trirefine.TriRefiner`

Uniform mesh refinement by recursive subdivisions.

Parameters

triangulation

[*Triangulation*] The encapsulated triangulation (to be refined)

`refine_field(`*self, z, triinterpolator=None, subdiv=3*`)`

Refine a field defined on the encapsulated triangulation.

Parameters

z

[1d-array-like of length `n_points`] Values of the field to refine, defined at the nodes of the encapsulated triangulation. (`n_points` is the number of points in the initial triangulation)

triinterpolator

[*TriInterpolator*, optional] Interpolator used for field interpolation. If not specified, a *CubicTriInterpolator* will be used.

subdiv

[int, default: 3] Recursion level for the subdivision. Each triangle is divided into `4**subdiv` child triangles.

Returns

refi_tri

[*Triangulation*] The returned refined triangulation.

refi_z

[1d array of length: *refi_tri* node count.] The returned interpolated field (at *refi_tri* nodes).

`refine_triangulation`(*self*, *return_tri_index=False*, *subdiv=3*)

Compute an uniformly refined triangulation *refi_triangulation* of the encapsulated `triangulation`.

This function refines the encapsulated triangulation by splitting each father triangle into 4 child sub-triangles built on the edges midside nodes, recursing *subdiv* times. In the end, each triangle is hence divided into `4**subdiv` child triangles.

Parameters

return_tri_index

[bool, default: False] Whether an index table indicating the father triangle index of each point is returned.

subdiv

[int, default: 3] Recursion level for the subdivision. Each triangle is divided into `4**subdiv` child triangles; hence, the default results in 64 refined subtriangles for each triangle of the initial triangulation.

Returns

refi_triangulation

[*Triangulation*] The refined triangulation.

found_index

[int array] Index of the initial triangulation containing triangle, for each point of *refi_triangulation*. Returned only if *return_tri_index* is set to True.

class matplotlib.tri.**TriAnalyzer**(*triangulation*)

Define basic tools for triangular mesh analysis and improvement.

A TriAnalyzer encapsulates a *Triangulation* object and provides basic tools for mesh analysis and mesh improvement.

Parameters

triangulation

[*Triangulation*] The encapsulated triangulation to analyze.

Attributes

scale_factors

Factors to rescale the triangulation into a unit square.

circle_ratios(*self*, *rescale=True*)

Return a measure of the triangulation triangles flatness.

The ratio of the incircle radius over the circumcircle radius is a widely used indicator of a triangle flatness. It is always `<= 0.5` and `== 0.5` only for equilateral triangles. Circle ratios below 0.01 denote very flat triangles.

To avoid unduly low values due to a difference of scale between the 2 axis, the triangular mesh can first be rescaled to fit inside a unit square with *scale_factors* (Only if *rescale* is True, which is its default value).

Parameters

rescale

[bool, default: True] If True, internally rescale (based on *scale_factors*), so that the (unmasked) triangles fit exactly inside a unit square mesh.

Returns

masked array

Ratio of the incircle radius over the circumcircle radius, for each 'rescaled' triangle of the encapsulated triangulation. Values corresponding to masked triangles are masked out.

`get_flat_tri_mask`(*self*, *min_circle_ratio=0.01*, *rescale=True*)

Eliminate excessively flat border triangles from the triangulation.

Returns a mask *new_mask* which allows to clean the encapsulated triangulation from its border-located flat triangles (according to their *circle_ratios()*). This mask is meant to be subsequently applied to the triangulation using *Triangulation.set_mask*. *new_mask* is an extension of the initial triangulation mask in the sense that an initially masked triangle will remain masked.

The *new_mask* array is computed recursively; at each step flat triangles are removed only if they share a side with the current mesh border. Thus no new holes in the triangulated domain will be created.

Parameters

min_circle_ratio

[float, default: 0.01] Border triangles with incircle/circumcircle radii ratio r/R will be removed if r/R < *min_circle_ratio*.

rescale

[bool, default: True] If True, first, internally rescale (based on *scale_factors*) so that the (unmasked) triangles fit exactly inside a unit square mesh. This rescaling accounts for the difference of scale which might exist between the 2 axis.

Returns

bool array-like

Mask to apply to encapsulated triangulation. All the initially masked triangles remain masked in the *new_mask*.

Notes

The rationale behind this function is that a Delaunay triangulation - of an unstructured set of points - sometimes contains almost flat triangles at its border, leading to artifacts in plots (especially for high-resolution contouring). Masked with computed *new_mask*, the encapsulated triangulation would contain no more unmasked border triangles with a circle ratio below *min_circle_ratio*, thus improving the mesh quality for subsequent plots or interpolation.

`property scale_factors`

Factors to rescale the triangulation into a unit square.

Returns

(float, float)

Scaling factors (kx, ky) so that the triangulation `[triangulation.x * kx, triangulation.y * ky]` fits exactly inside a unit square.

18.57 matplotlib.type1font

A class representing a Type 1 font.

This version reads pfa and pfb files and splits them for embedding in pdf files. It also supports SlantFont and ExtendFont transformations, similarly to pdfTeX and friends. There is no support yet for subsetting.

Usage:

```
>>> font = Type1Font(filename)
>>> clear_part, encrypted_part, finale = font.parts
>>> slanted_font = font.transform({'slant': 0.167})
>>> extended_font = font.transform({'extend': 1.2})
```

Sources:

- Adobe Technical Note #5040, Supporting Downloadable PostScript Language Fonts.
- Adobe Type 1 Font Format, Adobe Systems Incorporated, third printing, v1.1, 1993. ISBN 0-201-57044-0.

class matplotlib.type1font.**Type1Font**(*input*)

Bases: `object`

A class representing a Type-1 font, for use by backends.

Attributes

parts

[tuple] A 3-tuple of the cleartext part, the encrypted part, and the finale of zeros.

prop

[Dict[str, Any]] A dictionary of font properties.

Initialize a Type-1 font.

Parameters

input

[str or 3-tuple] Either a pfb file name, or a 3-tuple of already-decoded Type-1 font *parts*.

parts

`prop`

`transform`(*self*, *effects*)
Return a new font that is slanted and/or extended.

Parameters

effects

[dict] A dict with optional entries:

- **'slant'**

  [float, default: 0] Tangent of the angle that the font is to be slanted to the right. Negative values slant to the left.

- **'extend'**

  [float, default: 1] Scaling factor for the font width. Values less than 1 condense the glyphs.

Returns

Type1Font

18.58 matplotlib.units

The classes here provide support for using custom classes with Matplotlib, e.g., those that do not expose the array interface but know how to convert themselves to arrays. It also supports classes with units and units conversion. Use cases include converters for custom objects, e.g., a list of datetime objects, as well as for objects that are unit aware. We don't assume any particular units implementation; rather a units implementation must provide the register with the Registry converter dictionary and a *ConversionInterface*. For example, here is a complete implementation which supports plotting with native datetime objects:

```
import matplotlib.units as units
import matplotlib.dates as dates
import matplotlib.ticker as ticker
import datetime

class DateConverter(units.ConversionInterface):

    @staticmethod
    def convert(value, unit, axis):
        'Convert a datetime value to a scalar or array'
        return dates.date2num(value)

    @staticmethod
    def axisinfo(unit, axis):
```

(continues on next page)

(continued from previous page)

```
        'Return major and minor tick locators and formatters'
        if unit!='date': return None
        majloc = dates.AutoDateLocator()
        majfmt = dates.AutoDateFormatter(majloc)
        return AxisInfo(majloc=majloc,
                        majfmt=majfmt,
                        label='date')

    @staticmethod
    def default_units(x, axis):
        'Return the default unit for x or None'
        return 'date'

# Finally we register our object type with the Matplotlib units registry.
units.registry[datetime.date] = DateConverter()
```

class matplotlib.units.**AxisInfo**(*majloc=None*, *minloc=None*, *majfmt=None*, *minfmt=None*, *label=None*, *default_limits=None*)

Bases: object

Information to support default axis labeling, tick labeling, and limits.

An instance of this class must be returned by *ConversionInterface.axisinfo*.

Parameters

majloc, minloc

[Locator, optional] Tick locators for the major and minor ticks.

majfmt, minfmt

[Formatter, optional] Tick formatters for the major and minor ticks.

label

[str, optional] The default axis label.

default_limits

[optional] The default min and max limits of the axis if no data has been plotted.

Notes

If any of the above are `None`, the axis will simply use the default value.

exception matplotlib.units.**ConversionError**
Bases: `TypeError`

class matplotlib.units.**ConversionInterface**
Bases: `object`

The minimal interface for a converter to take custom data types (or sequences) and convert them to values Matplotlib can use.

static axisinfo(*unit*, *axis*)
Return an *AxisInfo* for the axis with the specified units.

static convert(*obj*, *unit*, *axis*)
Convert *obj* using *unit* for the specified *axis*.

If *obj* is a sequence, return the converted sequence. The output must be a sequence of scalars that can be used by the numpy array layer.

static default_units(*x*, *axis*)
Return the default unit for *x* or `None` for the given axis.

static is_numlike(*x*)
The Matplotlib datalim, autoscaling, locators etc work with scalars which are the units converted to floats given the current unit. The converter may be passed these floats, or arrays of them, even when units are set.

class matplotlib.units.**DecimalConverter**
Bases: *matplotlib.units.ConversionInterface*

Converter for decimal.Decimal data to float.

static axisinfo(*unit*, *axis*)
Return an *AxisInfo* for the axis with the specified units.

static convert(*value*, *unit*, *axis*)
Convert Decimals to floats.

The *unit* and *axis* arguments are not used.

Parameters

value

[decimal.Decimal or iterable] Decimal or list of Decimal need to be converted

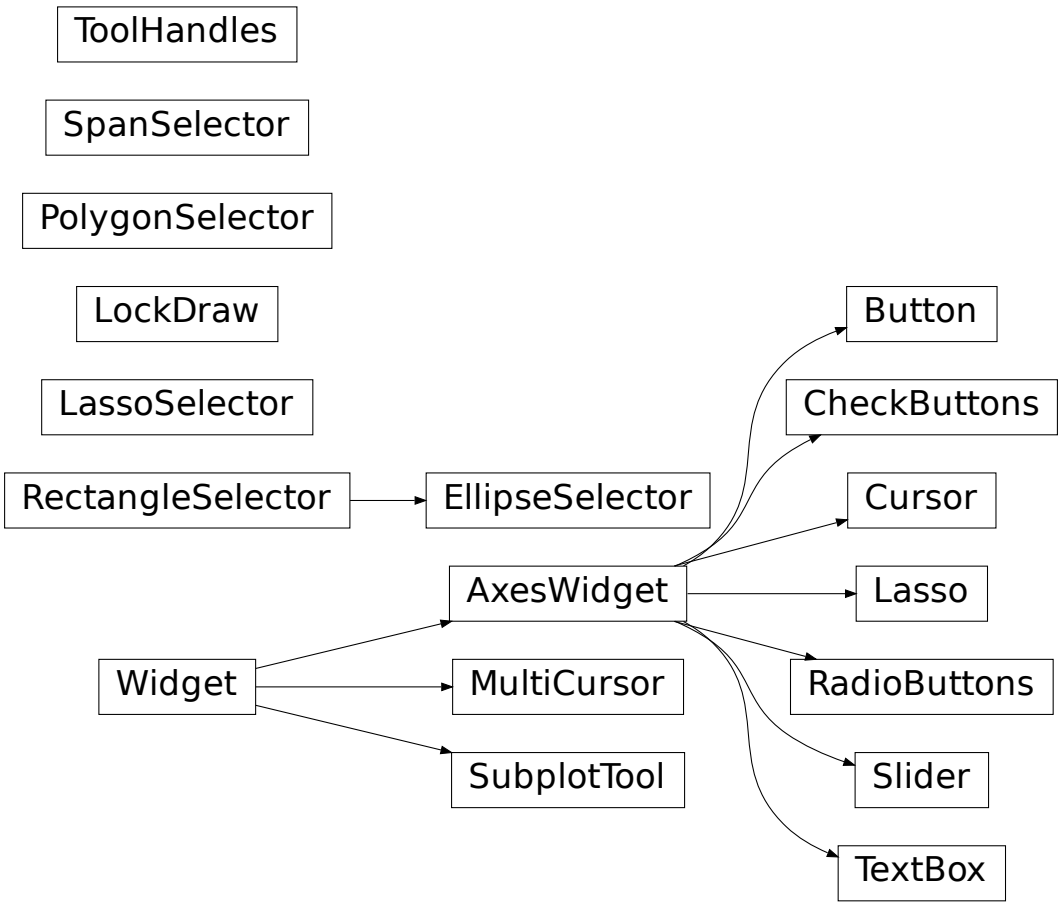
static default_units(*x*, *axis*)
Return the default unit for *x* or `None` for the given axis.

class matplotlib.units.**Registry**
Bases: `dict`

Register types with conversion interface.

`get_converter`(*self*, *x*)
Get the converter interface instance for *x*, or None.

18.59 matplotlib.widgets



18.59.1 GUI neutral widgets

Widgets that are designed to work for any of the GUI backends. All of these widgets require you to predefine a *matplotlib.axes.Axes* instance and pass that as the first parameter. Matplotlib doesn't try to be too smart with respect to layout -- you will have to figure out how wide and tall you want your Axes to be to accommodate your widget.

class matplotlib.widgets.**AxesWidget**(*ax*)

Bases: *matplotlib.widgets.Widget*

Widget connected to a single *Axes*.

To guarantee that the widget remains responsive and not garbage-collected, a reference to the object should be maintained by the user.

This is necessary because the callback registry maintains only weak-refs to the functions, which are member functions of the widget. If there are no references to the widget object it may be garbage collected which will disconnect the callbacks.

Attributes

ax

[*Axes*] The parent axes for the widget.

canvas

[*FigureCanvasBase*] The parent figure canvas for the widget.

`active`

[bool] Is the widget active?

`connect_event`(*self*, *event*, *callback*)

Connect callback with an event.

This should be used in lieu of `figure.canvas.mpl_connect` since this function stores callback ids for later clean up.

`disconnect_events`(*self*)

Disconnect all events created by this widget.

class matplotlib.widgets.**Button**(*ax*, *label*, *image=None*, *color='0.85'*, *hovercolor='0.95'*)

Bases: *matplotlib.widgets.AxesWidget*

A GUI neutral button.

For the button to remain responsive you must keep a reference to it. Call *on_clicked* to connect to the button.

Attributes

ax

The *matplotlib.axes.Axes* the button renders into.

label

A *matplotlib.text.Text* instance.

color

The color of the button when not hovering.

hovercolor

The color of the button when hovering.

Parameters

ax

[*Axes*] The *Axes* instance the button will be placed into.

label

[str] The button text.

image

[array-like or PIL Image] The image to place in the button, if not *None*. The parameter is directly forwarded to *imshow*.

color

[color] The color of the button when not activated.

hovercolor

[color] The color of the button when the mouse is over it.

`disconnect`(*self*, *cid*)
Remove the callback function with connection id *cid*.

`on_clicked`(*self*, *func*)
Connect the callback function *func* to button click events.

Returns a connection id, which can be used to disconnect the callback.

`class matplotlib.widgets.CheckButtons`(*ax*, *labels*, *actives=None*)
Bases: *matplotlib.widgets.AxesWidget*

A GUI neutral set of check buttons.

For the check buttons to remain responsive you must keep a reference to this object.

Connect to the CheckButtons with the *on_clicked* method.

Attributes

ax

[*Axes*] The parent axes for the widget.

labels

[list of *Text*]

rectangles

[list of *Rectangle*]

lines

[list of (*Line2D*, *Line2D*) pairs] List of lines for the x's in the check boxes. These lines exist for each box, but have `set_visible(False)` when its box is not checked.

Add check buttons to *matplotlib.axes.Axes* instance *ax*

Parameters

ax

[*Axes*] The parent axes for the widget.

labels

[list of str] The labels of the check buttons.

actives

[list of bool, optional] The initial check states of the buttons. The list must have the same length as *labels*. If not given, all buttons are unchecked.

`disconnect`(*self*, *cid*)
Remove the observer with connection id *cid*.

`get_status`(*self*)
Return a tuple of the status (True/False) of all of the check buttons.

`on_clicked`(*self*, *func*)
Connect the callback function *func* to button click events.

Returns a connection id, which can be used to disconnect the callback.

`set_active`(*self*, *index*)
Toggle (activate or deactivate) a check button by index.

Callbacks will be triggered if `eventson` is True.

Parameters

index

[int] Index of the check button to toggle.

Raises

ValueError

If *index* is invalid.

class matplotlib.widgets.Cursor(*ax*, *horizOn=True*, *vertOn=True*, *useblit=False*, ***lineprops*)

Bases: *matplotlib.widgets.AxesWidget*

A crosshair cursor that spans the axes and moves with mouse cursor.

For the cursor to remain responsive you must keep a reference to it.

Parameters

ax

[*matplotlib.axes.Axes*] The *Axes* to attach the cursor to.

horizOn

[bool, default: True] Whether to draw the horizontal line.

vertOn

[bool, default: True] Whether to draw the vertical line.

useblit

[bool, default: False] Use blitting for faster drawing if supported by the backend.

Other Parameters

****lineprops**

Line2D properties that control the appearance of the lines. See also *axhline*.

Examples

See /gallery/widgets/cursor.

clear(*self*, *event*)
Internal event handler to clear the cursor.

onmove(*self*, *event*)
Internal event handler to draw the cursor when the mouse moves.

class matplotlib.widgets.EllipseSelector(*ax*, *onselect*, *drawtype='box'*, *minspanx=0*, *minspany=0*, *useblit=False*, *lineprops=None*, *rectprops=None*, *spancoords='data'*, *button=None*, *maxdist=10*, *marker_props=None*, *interactive=False*, *state_modifier_keys=None*)

Bases: *matplotlib.widgets.RectangleSelector*

Select an elliptical region of an axes.

For the cursor to remain responsive you must keep a reference to it.

Example usage:

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.widgets import EllipseSelector

def onselect(eclick, erelease):
    "eclick and erelease are matplotlib events at press and release."
    print('startposition: (%f, %f)' % (eclick.xdata, eclick.ydata))
    print('endposition  : (%f, %f)' % (erelease.xdata, erelease.ydata))
    print('used button  : ', eclick.button)

def toggle_selector(event):
    print(' Key pressed.')
    if event.key in ['Q', 'q'] and toggle_selector.ES.active:
        print('EllipseSelector deactivated.')
        toggle_selector.RS.set_active(False)
    if event.key in ['A', 'a'] and not toggle_selector.ES.active:
        print('EllipseSelector activated.')
        toggle_selector.ES.set_active(True)

x = np.arange(100.) / 99
y = np.sin(x)
fig, ax = plt.subplots()
ax.plot(x, y)

toggle_selector.ES = EllipseSelector(ax, onselect, drawtype='line')
fig.canvas.mpl_connect('key_press_event', toggle_selector)
plt.show()
```

Parameters

ax

[*Axes*] The parent axes for the widget.

onselect

[function] A callback function that is called after a selection is completed. It must have the signature:

```
def onselect(eclick: MouseEvent, erelease: MouseEvent)
```

where *eclick* and *erelease* are the mouse click and release *MouseEvent*s that start and complete the selection.

drawtype

[{"box", "line", "none"}, default: "box"] Whether to draw the full rectangle box, the diagonal line of the rectangle, or nothing at all.

minspanx

[float, default: 0] Selections with an x-span less than *minspanx* are ignored.

minspany

[float, default: 0] Selections with an y-span less than *minspany* are ignored.

useblit

[bool, default: False] Whether to use blitting for faster drawing (if supported by the backend).

lineprops

[dict, optional] Properties with which the line is drawn, if `drawtype == "line"`. Default:

```
dict(color="black", linestyle="-", linewidth=2, alpha=0.5)
```

rectprops

[dict, optional] Properties with which the rectangle is drawn, if `drawtype == "box"`. Default:

```
dict(facecolor="red", edgecolor="black", alpha=0.2, fill=True)
```

spancoords

[{"data", "pixels"}, default: "data"] Whether to interpret *minspanx* and *minspany* in data or in pixel coordinates.

button

[*MouseButton*, list of *MouseButton*, default: all buttons] Button(s) that trigger rectangle selection.

maxdist

[float, default: 10] Distance in pixels within which the interactive tool handles can be activated.

marker_props

[dict] Properties with which the interactive handles are drawn. Currently not implemented and ignored.

interactive

[bool, default: False] Whether to draw a set of handles that allow interaction with the widget after it is drawn.

state_modifier_keys

[dict, optional] Keyboard modifiers which affect the widget's behavior. Values amend the defaults.

- "move": Move the existing shape, default: no modifier.
- "clear": Clear the current shape, default: "escape".
- "square": Makes the shape square, default: "shift".
- "center": Make the initial point the center of the shape, default: "ctrl".

"square" and "center" can be combined.

`draw_shape`(*self*, *extents*)

`class matplotlib.widgets.Lasso`(*ax*, *xy*, *callback=None*, *useblit=True*)

Bases: *matplotlib.widgets.AxesWidget*

Selection curve of an arbitrary shape.

The selected path can be used in conjunction with *contains_point* to select data points from an image.

Unlike *LassoSelector*, this must be initialized with a starting point *xy*, and the *Lasso* events are destroyed upon release.

Parameters

ax

[*Axes*] The parent axes for the widget.

xy

[(float, float)] Coordinates of the start of the lasso.

callback

[callable] Whenever the lasso is released, the *callback* function is called and passed the vertices of the selected path.

`onmove`(*self*, *event*)

`onrelease`(*self*, *event*)

class matplotlib.widgets.**LassoSelector**(*ax*, *onselect=None*, *useblit=True*, *lineprops=None*, *button=None*)

Bases: matplotlib.widgets._SelectorWidget

Selection curve of an arbitrary shape.

For the selector to remain responsive you must keep a reference to it.

The selected path can be used in conjunction with *contains_point* to select data points from an image.

In contrast to *Lasso*, *LassoSelector* is written with an interface similar to *RectangleSelector* and *SpanSelector*, and will continue to interact with the axes until disconnected.

Example usage:

```
ax = subplot(111)
ax.plot(x, y)

def onselect(verts):
    print(verts)
lasso = LassoSelector(ax, onselect)
```

Parameters

ax

[*Axes*] The parent axes for the widget.

onselect

[function] Whenever the lasso is released, the *onselect* function is called and passed the vertices of the selected path.

button

[*MouseButton* or list of *MouseButton*, optional] The mouse buttons used for rectangle selection. Default is None, which corresponds to all buttons.

onpress(*self*, *event*)

onrelease(*self*, *event*)

class matplotlib.widgets.**LockDraw**

Bases: object

Some widgets, like the cursor, draw onto the canvas, and this is not desirable under all circumstances, like when the toolbar is in zoom-to-rect mode and drawing a rectangle. To avoid this, a widget can acquire a canvas' lock with canvas.widgetlock(widget) before drawing on the canvas; this will prevent other widgets from doing so at the same time (if they also try to acquire the lock first).

`available`(*self*, *o*)
Return whether drawing is available to *o*.

`isowner`(*self*, *o*)
Return whether *o* owns this lock.

`locked`(*self*)
Return whether the lock is currently held by an owner.

`release`(*self*, *o*)
Release the lock from *o*.

`class matplotlib.widgets.MultiCursor`(*canvas*, *axes*, *useblit=True*, *horizOn=False*, *vertOn=True*, ***lineprops*)
Bases: *matplotlib.widgets.Widget*

Provide a vertical (default) and/or horizontal line cursor shared between multiple axes.

For the cursor to remain responsive you must keep a reference to it.

Example usage:

```
from matplotlib.widgets import MultiCursor
import matplotlib.pyplot as plt
import numpy as np

fig, (ax1, ax2) = plt.subplots(nrows=2, sharex=True)
t = np.arange(0.0, 2.0, 0.01)
ax1.plot(t, np.sin(2*np.pi*t))
ax2.plot(t, np.sin(4*np.pi*t))

multi = MultiCursor(fig.canvas, (ax1, ax2), color='r', lw=1,
                    horizOn=False, vertOn=True)
plt.show()
```

`clear`(*self*, *event*)
Clear the cursor.

`connect`(*self*)
Connect events.

`disconnect`(*self*)
Disconnect events.

`onmove`(*self*, *event*)

`class matplotlib.widgets.PolygonSelector`(*ax*, *onselect*, *useblit=False*, *lineprops=None*, *markerprops=None*, *vertex_select_radius=15*)
Bases: `matplotlib.widgets._SelectorWidget`

Select a polygon region of an axes.

Place vertices with each mouse click, and make the selection by completing the polygon (clicking on the first vertex). Hold the *ctrl* key and click and drag a

vertex to reposition it (the *ctrl* key is not necessary if the polygon has already been completed). Hold the *shift* key and click and drag anywhere in the axes to move all vertices. Press the *esc* key to start a new polygon.

For the selector to remain responsive you must keep a reference to it.

Parameters

ax

[*Axes*] The parent axes for the widget.

onselect

[function] When a polygon is completed or modified after completion, the *onselect* function is called and passed a list of the vertices as `(xdata, ydata)` tuples.

useblit

[bool, default: False]

lineprops

[dict, default: `dict(color='k', linestyle='-', linewidth=2, alpha=0.5)`.] Artist properties for the line representing the edges of the polygon.

markerprops

[dict, default: `dict(marker='o', markersize=7, mec='k', mfc='k', alpha=0.5)`.] Artist properties for the markers drawn at the vertices of the polygon.

vertex_select_radius

[float, default: 15px] A vertex is selected (to complete the polygon or to move a vertex) if the mouse click is within *vertex_select_radius* pixels of the vertex.

Examples

/gallery/widgets/polygon_selector_demo

`onmove`(*self*, *event*)
Cursor move event handler and validator

`property verts`
The polygon vertices, as a list of `(x, y)` pairs.

`class matplotlib.widgets.RadioButtons`(*ax*, *labels*, *active=0*, *activecolor='blue'*)
Bases: *matplotlib.widgets.AxesWidget*

A GUI neutral radio button.

For the buttons to remain responsive you must keep a reference to this object.

Connect to the RadioButtons with the *on_clicked* method.

Attributes

ax

[*Axes*] The parent axes for the widget.

activecolor

[color] The color of the selected button.

labels

[list of *Text*] The button labels.

circles

[list of *Circle*] The buttons.

value_selected

[str] The label text of the currently selected button.

Add radio buttons to an *Axes*.

Parameters

ax

[*Axes*] The axes to add the buttons to.

labels

[list of str] The button labels.

active

[int] The index of the initially selected button.

activecolor

[color] The color of the selected button.

`disconnect`(*self*, *cid*)
Remove the observer with connection id *cid*.

`on_clicked`(*self*, *func*)
Connect the callback function *func* to button click events.

Returns a connection id, which can be used to disconnect the callback.

`set_active`(*self*, *index*)
Select button with number *index*.

Callbacks will be triggered if `eventson` is True.

class matplotlib.widgets.**RectangleSelector**(*ax*, *onselect*, *drawtype='box'*, *minspanx=0*, *minspany=0*, *useblit=False*, *lineprops=None*, *rectprops=None*, *spancoords='data'*, *button=None*, *maxdist=10*, *marker_props=None*, *interactive=False*, *state_modifier_keys=None*)

Bases: matplotlib.widgets._SelectorWidget

Select a rectangular region of an axes.

For the cursor to remain responsive you must keep a reference to it.

Examples

/gallery/widgets/rectangle_selector

Parameters

ax

[*Axes*] The parent axes for the widget.

onselect

[function] A callback function that is called after a selection is completed. It must have the signature:

```
def onselect(eclick: MouseEvent, erelease: MouseEvent)
```

where *eclick* and *erelease* are the mouse click and release *MouseEvent*s that start and complete the selection.

drawtype

[{"box", "line", "none"}, default: "box"] Whether to draw the full rectangle box, the diagonal line of the rectangle, or nothing at all.

minspanx

[float, default: 0] Selections with an x-span less than *minspanx* are ignored.

minspany

[float, default: 0] Selections with an y-span less than *minspany* are ignored.

useblit

[bool, default: False] Whether to use blitting for faster drawing (if supported by the backend).

lineprops

[dict, optional] Properties with which the line is drawn, if `drawtype == "line"`. Default:

```
dict(color="black", linestyle="-", linewidth=2, alpha=0.5)
```

rectprops

[dict, optional] Properties with which the rectangle is drawn, if `drawtype == "box"`. Default:

```
dict(facecolor="red", edgecolor="black", alpha=0.2, fill=True)
```

spancoords

[{"data", "pixels"}, default: "data"] Whether to interpret *minspanx* and *minspany* in data or in pixel coordinates.

button

[*MouseButton*, list of *MouseButton*, default: all buttons] Button(s) that trigger rectangle selection.

maxdist

[float, default: 10] Distance in pixels within which the interactive tool handles can be activated.

marker_props

[dict] Properties with which the interactive handles are drawn. Currently not implemented and ignored.

interactive

[bool, default: False] Whether to draw a set of handles that allow interaction with the widget after it is drawn.

state_modifier_keys

[dict, optional] Keyboard modifiers which affect the widget's behavior. Values amend the defaults.

- "move": Move the existing shape, default: no modifier.
- "clear": Clear the current shape, default: "escape".
- "square": Makes the shape square, default: "shift".
- "center": Make the initial point the center of the shape, default: "ctrl".

"square" and "center" can be combined.

`property center`
Center of rectangle

property corners
Corners of rectangle from lower left, moving clockwise.

draw_shape(*self*, *extents*)

property edge_centers
Midpoint of rectangle edges from left, moving clockwise.

property extents
Return (xmin, xmax, ymin, ymax).

property geometry
Return an array of shape (2, 5) containing the x (`RectangleSelector.geometry[1, :]`) and y (`RectangleSelector.geometry[0, :]`) coordinates of the four corners of the rectangle starting and ending in the top left corner.

class matplotlib.widgets.Slider(*ax*, *label*, *valmin*, *valmax*, *valinit=0.5*, *valfmt=None*, *closedmin=True*, *closedmax=True*, *slidermin=None*, *slidermax=None*, *dragging=True*, *valstep=None*, *orientation='horizontal'*, ***kwargs*)

Bases: *matplotlib.widgets.AxesWidget*

A slider representing a floating point range.

Create a slider from *valmin* to *valmax* in axes *ax*. For the slider to remain responsive you must maintain a reference to it. Call *on_changed()* to connect to the slider event.

Attributes

val

[float] Slider value.

Parameters

ax

[Axes] The Axes to put the slider in.

label

[str] Slider label.

valmin

[float] The minimum value of the slider.

valmax

[float] The maximum value of the slider.

valinit

[float, default: 0.5] The slider initial position.

valfmt

[str, default: None] %-format string used to format the slider value. If None, a *ScalarFormatter* is used instead.

closedmin

[bool, default: True] Whether the slider interval is closed on the bottom.

closedmax

[bool, default: True] Whether the slider interval is closed on the top.

slidermin

[Slider, default: None] Do not allow the current slider to have a value less than the value of the Slider *slidermin*.

slidermax

[Slider, default: None] Do not allow the current slider to have a value greater than the value of the Slider *slidermax*.

dragging

[bool, default: True] If True the slider can be dragged by the mouse.

valstep

[float, default: None] If given, the slider will snap to multiples of *valstep*.

orientation

[{'horizontal', 'vertical'}, default: 'horizontal'] The orientation of the slider.

Notes

Additional kwargs are passed on to `self.poly` which is the *Rectangle* that draws the slider knob. See the *Rectangle* documentation for valid property names (`facecolor`, `edgecolor`, `alpha`, etc.).

`disconnect`(*self*, *cid*)

Remove the observer with connection id *cid*

Parameters

cid

[int] Connection id of the observer to be removed

on_changed(*self*, *func*)
When the slider value is changed call *func* with the new slider value

Parameters

func

[callable] Function to call when slider is changed. The function must accept a single float as its arguments.

Returns

int

Connection id (which can be used to disconnect *func*)

reset(*self*)
Reset the slider to the initial value

set_val(*self*, *val*)
Set slider value to *val*

Parameters

val

[float]

class matplotlib.widgets.**SpanSelector**(*ax*, *onselect*, *direction*, *minspan=None*, *useblit=False*, *rectprops=None*, *onmove_callback=None*, *span_stays=False*, *button=None*)
Bases: matplotlib.widgets._SelectorWidget

Visually select a min/max range on a single axis and call a function with those values.

To guarantee that the selector remains responsive, keep a reference to it.

In order to turn off the SpanSelector, set span_selector.active to False. To turn it back on, set it to True.

Parameters

ax

[*matplotlib.axes.Axes*]

onselect

[func(min, max), min/max are floats]

direction

[{"horizontal", "vertical"}] The direction along which to draw the span selector.

minspan

[float, default: None] If selection is less than *minspan*, do not call *onselect*.

useblit

[bool, default: False] If True, use the backend-dependent blitting features for faster canvas updates.

rectprops

[dict, default: None] Dictionary of *matplotlib.patches.Patch* properties.

onmove_callback

[func(min, max), min/max are floats, default: None] Called on mouse move while the span is being selected.

span_stays

[bool, default: False] If True, the span stays visible after the mouse is released.

button

[*MouseButton* or list of *MouseButton*] The mouse buttons which activate the span selector.

Examples

```
>>> import matplotlib.pyplot as plt
>>> import matplotlib.widgets as mwidgets
>>> fig, ax = plt.subplots()
>>> ax.plot([1, 2, 3], [10, 50, 100])
>>> def onselect(vmin, vmax):
...     print(vmin, vmax)
>>> rectprops = dict(facecolor='blue', alpha=0.5)
>>> span = mwidgets.SpanSelector(ax, onselect, 'horizontal',
...                              rectprops=rectprops)
>>> fig.show()
```

See also: /gallery/widgets/span_selector

`ignore`(*self*, *event*)
Return whether *event* should be ignored.

This method should be called at the beginning of any event callback.

`new_axes`(*self*, *ax*)
Set SpanSelector to operate on a new Axes.

class matplotlib.widgets.SubplotTool(*targetfig*, *toolfig*)

Bases: *matplotlib.widgets.Widget*

A tool to adjust the subplot params of a *matplotlib.figure.Figure*.

Parameters

targetfig

[*Figure*] The figure instance to adjust.

toolfig

[*Figure*] The figure instance to embed the subplot tool into.

property axbottom

property axhspace

property axleft

property axright

property axtop

property axwspace

funcbottom(*self*, *val*)

[*Deprecated*]

Notes

Deprecated since version 3.3:

funchspace(*self*, *val*)

[*Deprecated*]

Notes

Deprecated since version 3.3:

funcleft(*self*, *val*)

[*Deprecated*]

Notes

Deprecated since version 3.3:

`funcright`(*self*, *val*)
[*Deprecated*]

Notes

Deprecated since version 3.3:

`functop`(*self*, *val*)
[*Deprecated*]

Notes

Deprecated since version 3.3:

`funcwspace`(*self*, *val*)
[*Deprecated*]

Notes

Deprecated since version 3.3:

`class matplotlib.widgets.TextBox`(*ax*, *label*, *initial=''*, *color='.95'*, *hovercolor='1'*, *label_pad=0.01*)

Bases: *matplotlib.widgets.AxesWidget*

A GUI neutral text input box.

For the text box to remain responsive you must keep a reference to it.

Call *on_text_change* to be updated whenever the text changes.

Call *on_submit* to be updated whenever the user hits enter or leaves the text entry field.

Attributes

ax

[*Axes*] The parent axes for the widget.

label

[*Text*]

color

[color] The color of the text box when not hovering.

hovercolor

[color] The color of the text box when hovering.

Parameters

ax

[*Axes*] The *Axes* instance the button will be placed into.

label

[str] Label for this text box.

initial

[str] Initial value in the text box.

color

[color] The color of the box.

hovercolor

[color] The color of the box when the mouse is over it.

label_pad

[float] The distance between the label and the right side of the textbox.

`begin_typing`(*self*, *x*)

`disconnect`(*self*, *cid*)
Remove the observer with connection id *cid*.

`on_submit`(*self*, *func*)
When the user hits enter or leaves the submission box, call this *func* with event.

A connection id is returned which can be used to disconnect.

`on_text_change`(*self*, *func*)
When the text changes, call this *func* with event.

A connection id is returned which can be used to disconnect.

`property params_to_disable`

`position_cursor`(*self*, *x*)

`set_val`(*self*, *val*)

`stop_typing`(*self*)

`property text`

class matplotlib.widgets.ToolHandles(*ax*, *x*, *y*, *marker='o'*, *marker_props=None*, *useblit=True*)

Bases: object

Control handles for canvas tools.

Parameters

ax

[*matplotlib.axes.Axes*] Matplotlib axes where tool handles are displayed.

x, y

[1D arrays] Coordinates of control handles.

marker

[str] Shape of marker used to display handle. See *matplotlib.pyplot.plot*.

marker_props

[dict] Additional marker properties. See *matplotlib.lines.Line2D*.

closest(*self*, *x*, *y*)
Return index and pixel distance to closest index.

set_animated(*self*, *val*)

set_data(*self*, *pts*, *y=None*)
Set x and y positions of handles

set_visible(*self*, *val*)

property x

property y

class matplotlib.widgets.Widget

Bases: object

Abstract base class for GUI neutral widgets

property active
Is the widget active?

drawon = True

eventson = True

get_active(*self*)
Get whether the widget is active.

ignore(*self*, *event*)
Return whether *event* should be ignored.

This method should be called at the beginning of any event callback.

`set_active`(*self, active*)
Set whether the widget is active.

CHAPTER

NINETEEN

TOOLKITS

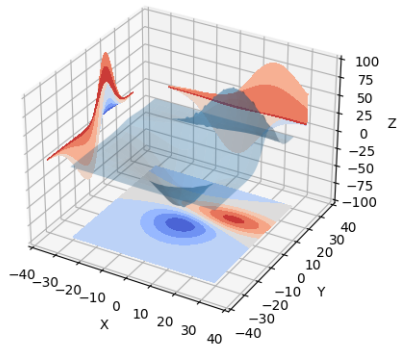
Toolkits are collections of application-specific functions that extend Matplotlib. The following toolkits are included:

19.1 Toolkits

Toolkits are collections of application-specific functions that extend Matplotlib.

19.1.1 mplot3d

`mpl_toolkits.mplot3d` provides some basic 3D plotting (scatter, surf, line, mesh) tools. Not the fastest or most feature complete 3D library out there, but it ships with Matplotlib and thus may be a lighter weight solution for some use cases. Check out the *mplot3d tutorial* for more information.

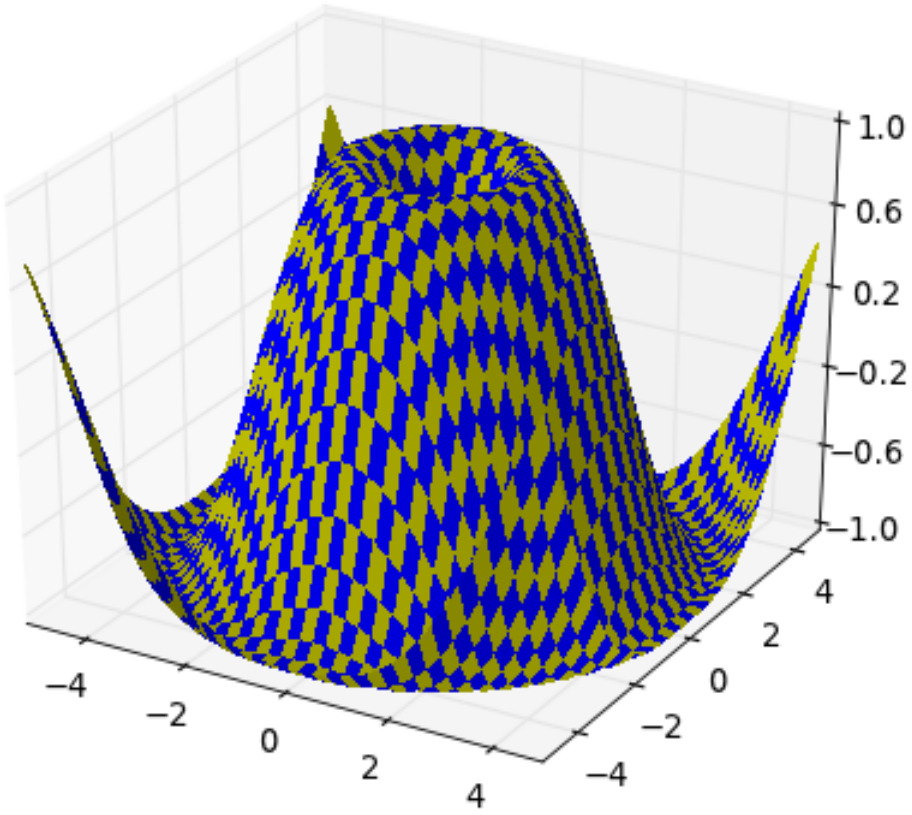


mplot3d

Matplotlib mplot3d toolkit

The mplot3d toolkit adds simple 3D plotting capabilities to matplotlib by supplying an axes object that can create a 2D projection of a 3D scene. The resulting graph will have the same look and feel as regular 2D plots.

See the *mplot3d tutorial* for more information on how to use this toolkit.



The interactive backends also provide the ability to rotate and zoom the 3D scene. One can rotate the 3D scene by simply clicking-and-dragging the scene. Zooming is done by right-clicking the scene and dragging the mouse up and down. Note that one does not use the zoom button like one would use for regular 2D plots.

mplot3d FAQ

How is mplot3d different from Mayavi?

Mayavi is a very powerful and featureful 3D graphing library. For advanced 3D scenes and excellent rendering capabilities, it is highly recommended to use Mayavi.

mplot3d was intended to allow users to create simple 3D graphs with the same "look-and-feel" as matplotlib's 2D plots. Furthermore, users can use the same toolkit that they are already familiar with to generate both their 2D and 3D plots.

My 3D plot doesn't look right at certain viewing angles

This is probably the most commonly reported issue with mplot3d. The problem is that -- from some viewing angles -- a 3D object would appear in front of another object, even though it is physically behind it. This can result in plots that do not look "physically correct."

Unfortunately, while some work is being done to reduce the occurrence of this artifact, it is currently an intractable problem, and can not be fully solved until matplotlib supports 3D graphics rendering at its core.

The problem occurs due to the reduction of 3D data down to 2D + z-order scalar. A single value represents the 3rd dimension for all parts of 3D objects in a collection. Therefore, when the bounding boxes of two collections intersect, it becomes possible for this artifact to occur. Furthermore, the intersection of two 3D objects (such as polygons or patches) can not be rendered properly in matplotlib's 2D rendering engine.

This problem will likely not be solved until OpenGL support is added to all of the backends (patches are greatly welcomed). Until then, if you need complex 3D scenes, we recommend using MayaVi.

I don't like how the 3D plot is laid out, how do I change that?

Historically, mplot3d has suffered from a hard-coding of parameters used to control visuals such as label spacing, tick length, and grid line width. Work is being done to eliminate this issue. For matplotlib v1.1.0, there is a semi-official manner to modify these parameters. See the note in the *mplot3d.axis3d* section of the mplot3d API documentation for more information.

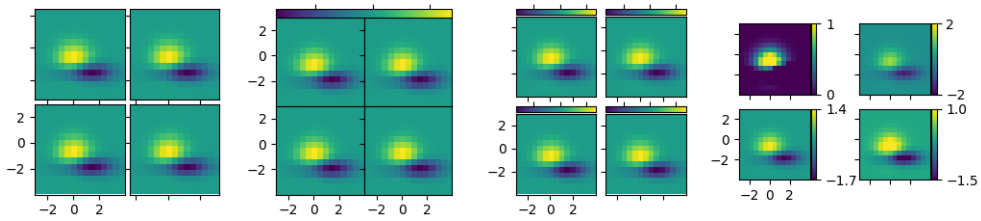
Links

- mpl3d API: *mplot3d API*

19.1.2 Matplotlib axes_grid1 Toolkit

The matplotlib *mpl_toolkits.axes_grid1* toolkit is a collection of helper classes to ease displaying multiple images in matplotlib. While the aspect parameter in matplotlib adjust the position of the single axes, axes_grid1 toolkit provides a framework to adjust the position of multiple axes according to their aspects.

See *What is axes_grid1 toolkit?* for a guide on the usage of axes_grid1.



The submodules of the axes_grid1 API are:

axes_grid1.anchored_artists	
axes_grid1.axes_divider	Helper classes to adjust the positions of multiple axes at drawing time.
axes_grid1.axes_grid	
axes_grid1.axes_rgb	
axes_grid1.axes_size	Provides classes of simple units that will be used with AxesDivider class (or others) to determine the size of each axes.
axes_grid1.colorbar	Colorbar toolkit with two classes and a function:
axes_grid1.inset_locator	A collection of functions and objects for creating or placing inset axes.
axes_grid1.mpl_axes	
axes_grid1.parasite_axes	

mpl_toolkits.axes_grid1.anchored_artists

Classes

AnchoredAuxTransformBox(transform, loc[, ...])	An anchored container with transformed coordinates.
AnchoredDirectionArrows(transform, label_x, ...)	Draw two perpendicular arrows to indicate directions.
AnchoredDrawingArea(width, height, xdescent, ...)	An anchored container with a fixed size and fillable DrawingArea.
AnchoredEllipse(transform, width, height, ...)	Draw an anchored ellipse of a given size.
AnchoredSizeBar(transform, size, label, loc)	Draw a horizontal scale bar with a center-aligned label underneath.

mpl_toolkits.axes_grid1.anchored_artists.AnchoredAuxTransformBox

class mpl_toolkits.axes_grid1.anchored_artists.**AnchoredAuxTransformBox**(*transform*, *loc*, *pad=0.4*, *borderpad=0.5*, *prop=None*, *frameon=True*, ***kwargs*)

Bases: *matplotlib.offsetbox.AnchoredOffsetbox*

An anchored container with transformed coordinates.

Artists added to the *drawing_area* are scaled according to the coordinates of the transformation used. The dimensions of this artist will scale to contain the artists added.

Parameters

transform

[*matplotlib.transforms.Transform*] The transformation object for the coordinate system in use, i.e., `matplotlib.axes.Axes.transData`.

loc

[int] Location of this artist. Valid location codes are:

```
'upper right'  : 1,
'upper left'   : 2,
```

(continues on next page)

(continued from previous page)

```
'lower left'   : 3,
'lower right'  : 4,
'right'        : 5,
'center left'  : 6,
'center right' : 7,
'lower center' : 8,
'upper center' : 9,
'center'       : 10
```

pad

[float, default: 0.4] Padding around the child objects, in fraction of the font size.

borderpad

[float, default: 0.5] Border padding, in fraction of the font size.

prop

[*matplotlib.font_manager.FontProperties*, optional] Font property used as a reference for paddings.

frameon

[bool, default: True] If True, draw a box around this artists.

****kwargs**

Keyworded arguments to pass to *matplotlib.offsetbox.AnchoredOffsetbox*.

Examples

To display an ellipse in the upper left, with a width of 0.1 and height of 0.4 in data coordinates:

```
>>> box = AnchoredAuxTransformBox(ax.transData, loc='upper left')
>>> el = Ellipse((0, 0), width=0.1, height=0.4, angle=30)
>>> box.drawing_area.add_artist(el)
>>> ax.add_artist(box)
```

Attributes

drawing_area

[*matplotlib.offsetbox.AuxTransformBox*] A container for artists to display.

`__init__`(*self*, *transform*, *loc*, *pad=0.4*, *borderpad=0.5*, *prop=None*, *frameon=True*, ***kwargs*)

An anchored container with transformed coordinates.

Artists added to the *drawing_area* are scaled according to the coordinates of the transformation used. The dimensions of this artist will scale to contain the artists added.

Parameters

transform

[*matplotlib.transforms.Transform*] The transformation object for the coordinate system in use, i.e., `matplotlib.axes.Axes.transData`.

loc

[int] Location of this artist. Valid location codes are:

```
'upper right'  : 1,
'upper left'   : 2,
'lower left'   : 3,
'lower right'  : 4,
'right'        : 5,
'center left'  : 6,
'center right' : 7,
'lower center' : 8,
'upper center' : 9,
'center'       : 10
```

pad

[float, default: 0.4] Padding around the child objects, in fraction of the font size.

borderpad

[float, default: 0.5] Border padding, in fraction of the font size.

prop

[*matplotlib.font_manager.FontProperties*, optional] Font property used as a reference for paddings.

frameon

[bool, default: True] If True, draw a box around this artists.

****kwargs**

Keyworded arguments to pass to *matplotlib.offsetbox.AnchoredOffsetbox*.

Examples

To display an ellipse in the upper left, with a width of 0.1 and height of 0.4 in data coordinates:

```
>>> box = AnchoredAuxTransformBox(ax.transData, loc='upper left')
>>> el = Ellipse((0, 0), width=0.1, height=0.4, angle=30)
>>> box.drawing_area.add_artist(el)
>>> ax.add_artist(box)
```

Attributes

drawing_area

[*matplotlib.offsetbox.AuxTransformBox*] A container for artists to display.

```
__module__ = 'mpl_toolkits.axes_grid1.anchored_artists'
```

Examples using `mpl_toolkits.axes_grid1.anchored_artists.AnchoredAuxTransformBox`

- sphx_glr_gallery_userdemo_anchored_box03.py

mpl_toolkits.axes_grid1.anchored_artists.AnchoredDirectionArrows

class mpl_toolkits.axes_grid1.anchored_artists.**AnchoredDirectionArrows**(*transform*, *label_x*, *label_y*, *length=0.15*, *fontsize=0.08*, *loc=2*, *angle=0*, *aspect_ratio=1*, *pad=0.4*, *borderpad=0.4*, *frameon=False*, *color='w'*, *alpha=1*, *sep_x=0.01*, *sep_y=0*, *fontproperties=None*, *back_length=0.15*, *head_width=10*, *head_length=15*, *tail_width=2*, *text_props=None*, *arrow_props=None*, ***kwargs*)

Bases: *matplotlib.offsetbox.AnchoredOffsetbox*

Draw two perpendicular arrows to indicate directions.

Parameters

transform

[*matplotlib.transforms.Transform*] The transformation object for the coordinate system in use, i.e., `matplotlib.axes.Axes.transAxes`.

label_x, label_y

[str] Label text for the x and y arrows

length

[float, default: 0.15] Length of the arrow, given in coordinates of *transform*.

fontsize

[float, default: 0.08] Size of label strings, given in coordinates of *transform*.

loc

[int, default: 2] Location of the direction arrows. Valid location codes are:

```
'upper right'  : 1,
'upper left'   : 2,
'lower left'   : 3,
'lower right'  : 4,
'right'        : 5,
'center left'  : 6,
'center right' : 7,
'lower center' : 8,
'upper center' : 9,
'center'       : 10
```

angle

[float, default: 0] The angle of the arrows in degrees.

aspect_ratio

[float, default: 1] The ratio of the length of arrow_x and arrow_y. Negative numbers can be used to change the direction.

pad

[float, default: 0.4] Padding around the labels and arrows, in fraction of the font size.

borderpad

[float, default: 0.4] Border padding, in fraction of the font size.

frameon

[bool, default: False] If True, draw a box around the arrows and labels.

color

[str, default: 'white'] Color for the arrows and labels.

alpha

[float, default: 1] Alpha values of the arrows and labels

sep_x, sep_y

[float, default: 0.01 and 0 respectively] Separation between the arrows and labels in coordinates of *transform*.

fontproperties

[*matplotlib.font_manager.FontProperties*, optional] Font properties for the label text.

back_length

[float, default: 0.15] Fraction of the arrow behind the arrow crossing.

head_width

[float, default: 10] Width of arrow head, sent to ArrowStyle.

head_length

[float, default: 15] Length of arrow head, sent to ArrowStyle.

tail_width

[float, default: 2] Width of arrow tail, sent to ArrowStyle.

text_props, arrow_props

[dict] Properties of the text and arrows, passed to *textpath.TextPath* and *patches.FancyArrowPatch*.

****kwargs**

Keyword arguments to pass to *matplotlib.offsetbox.AnchoredOffsetbox*.

Notes

If *prop* is passed as a keyword argument, but *fontproperties* is not, then *prop* is be assumed to be the intended *fontproperties*. Using both *prop* and *fontproperties* is not supported.

Examples

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> from mpl_toolkits.axes_grid1.anchored_artists import (
...     AnchoredDirectionArrows)
>>> fig, ax = plt.subplots()
>>> ax.imshow(np.random.random((10, 10)))
```

(continues on next page)

(continued from previous page)

```
>>> arrows = AnchoredDirectionArrows(ax.transAxes, '111', '110')
>>> ax.add_artist(arrows)
>>> fig.show()
```

Using several of the optional parameters, creating downward pointing arrow and high contrast text labels.

```
>>> import matplotlib.font_manager as fm
>>> fontprops = fm.FontProperties(family='monospace')
>>> arrows = AnchoredDirectionArrows(ax.transAxes, 'East', 'South',
...                                  loc='lower left', color='k',
...                                  aspect_ratio=-1, sep_x=0.02,
...                                  sep_y=-0.01,
...                                  text_props={'ec':'w', 'fc':'k'},
...                                  fontproperties=fontprops)
```

Attributes

arrow_x, arrow_y

[*matplotlib.patches.FancyArrowPatch*] Arrow x and y

text_path_x, text_path_y

[*matplotlib.textpath.TextPath*] Path for arrow labels

p_x, p_y

[*matplotlib.patches.PathPatch*] Patch for arrow labels

box

[*matplotlib.offsetbox.AuxTransformBox*] Container for the arrows and labels.

`__init__`(*self*, *transform*, *label_x*, *label_y*, *length=0.15*, *fontsize=0.08*, *loc=2*, *angle=0*, *aspect_ratio=1*, *pad=0.4*, *borderpad=0.4*, *frameon=False*, *color='w'*, *alpha=1*, *sep_x=0.01*, *sep_y=0*, *fontproperties=None*, *back_length=0.15*, *head_width=10*, *head_length=15*, *tail_width=2*, *text_props=None*, *arrow_props=None*, ***kwargs*)

Draw two perpendicular arrows to indicate directions.

Parameters

transform

[*matplotlib.transforms.Transform*] The transformation object for the coordinate system in use, i.e., `matplotlib.axes.Axes.transAxes`.

label_x, label_y

[str] Label text for the x and y arrows

length

[float, default: 0.15] Length of the arrow, given in coordinates of *transform*.

fontsize

[float, default: 0.08] Size of label strings, given in coordinates of *transform*.

loc

[int, default: 2] Location of the direction arrows. Valid location codes are:

```
'upper right'  : 1,
'upper left'   : 2,
'lower left'   : 3,
'lower right'  : 4,
'right'        : 5,
'center left'  : 6,
'center right' : 7,
'lower center' : 8,
'upper center' : 9,
'center'       : 10
```

angle

[float, default: 0] The angle of the arrows in degrees.

aspect_ratio

[float, default: 1] The ratio of the length of arrow_x and arrow_y. Negative numbers can be used to change the direction.

pad

[float, default: 0.4] Padding around the labels and arrows, in fraction of the font size.

borderpad

[float, default: 0.4] Border padding, in fraction of the font size.

frameon

[bool, default: False] If True, draw a box around the arrows and labels.

color

[str, default: 'white'] Color for the arrows and labels.

alpha

[float, default: 1] Alpha values of the arrows and labels

sep_x, sep_y

[float, default: 0.01 and 0 respectively] Separation between the arrows and labels in coordinates of *transform*.

fontproperties

[*matplotlib.font_manager.FontProperties*, optional] Font properties for the label text.

back_length

[float, default: 0.15] Fraction of the arrow behind the arrow crossing.

head_width

[float, default: 10] Width of arrow head, sent to ArrowStyle.

head_length

[float, default: 15] Length of arrow head, sent to ArrowStyle.

tail_width

[float, default: 2] Width of arrow tail, sent to ArrowStyle.

text_props, arrow_props

[dict] Properties of the text and arrows, passed to *textpath.TextPath* and *patches.FancyArrowPatch*.

****kwargs**

Keyworded arguments to pass to *matplotlib.offsetbox.AnchoredOffsetbox*.

Notes

If *prop* is passed as a keyword argument, but *fontproperties* is not, then *prop* is be assumed to be the intended *fontproperties*. Using both *prop* and *fontproperties* is not supported.

Examples

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> from mpl_toolkits.axes_grid1.anchored_artists import (
...     AnchoredDirectionArrows)
>>> fig, ax = plt.subplots()
>>> ax.imshow(np.random.random((10, 10)))
>>> arrows = AnchoredDirectionArrows(ax.transAxes, '111', '110')
>>> ax.add_artist(arrows)
>>> fig.show()
```

Using several of the optional parameters, creating downward pointing arrow and high contrast text labels.

```
>>> import matplotlib.font_manager as fm
>>> fontprops = fm.FontProperties(family='monospace')
>>> arrows = AnchoredDirectionArrows(ax.transAxes, 'East', 'South',
...                                  loc='lower left', color='k',
...                                  aspect_ratio=-1, sep_x=0.02,
...                                  sep_y=-0.01,
...                                  text_props={'ec':'w', 'fc':'k'},
...                                  fontproperties=fontprops)
```

Attributes

arrow_x, arrow_y

[*matplotlib.patches.FancyArrowPatch*] Arrow x and y

text_path_x, text_path_y

[*matplotlib.textpath.TextPath*] Path for arrow labels

p_x, p_y

[*matplotlib.patches.PathPatch*] Patch for arrow labels

box

[*matplotlib.offsetbox.AuxTransformBox*] Container for the arrows and labels.

```
__module__ = 'mpl_toolkits.axes_grid1.anchored_artists'
```

Examples using `mpl_toolkits.axes_grid1.anchored_artists.AnchoredDirectionArrows`

- sphx_glr_gallery_axes_grid1_demo_anchored_direction_arrows.py

mpl_toolkits.axes_grid1.anchored_artists.AnchoredDrawingArea

class mpl_toolkits.axes_grid1.anchored_artists.**AnchoredDrawingArea**(*width*, *height*, *xdescent*, *ydescent*, *loc*, *pad=0.4*, *borderpad=0.5*, *prop=None*, *frameon=True*, ***kwargs*)

Bases: *matplotlib.offsetbox.AnchoredOffsetbox*

An anchored container with a fixed size and fillable DrawingArea.

Artists added to the *drawing_area* will have their coordinates interpreted as pixels. Any transformations set on the artists will be overridden.

Parameters

width, height

[float] width and height of the container, in pixels.

xdescent, ydescent

[float] descent of the container in the x- and y- direction, in pixels.

loc

[int] Location of this artist. Valid location codes are:

```
'upper right'  : 1,
'upper left'   : 2,
'lower left'   : 3,
'lower right'  : 4,
'right'        : 5,
'center left'  : 6,
'center right' : 7,
'lower center' : 8,
'upper center' : 9,
'center'       : 10
```

pad

[float, default: 0.4] Padding around the child objects, in fraction of the font size.

borderpad

[float, default: 0.5] Border padding, in fraction of the font size.

prop

[*matplotlib.font_manager.FontProperties*, optional] Font property used as a reference for paddings.

frameon

[bool, default: True] If True, draw a box around this artists.

****kwargs**

Keyword arguments to pass to *matplotlib.offsetbox.AnchoredOffsetbox*.

Examples

To display blue and red circles of different sizes in the upper right of an axes *ax*:

```
>>> ada = AnchoredDrawingArea(20, 20, 0, 0,
...                           loc='upper right', frameon=False)
>>> ada.drawing_area.add_artist(Circle((10, 10), 10, fc="b"))
>>> ada.drawing_area.add_artist(Circle((30, 10), 5, fc="r"))
>>> ax.add_artist(ada)
```

Attributes

drawing_area

[*matplotlib.offsetbox.DrawingArea*] A container for artists to display.

`__init__`(*self*, *width*, *height*, *xdescent*, *ydescent*, *loc*, *pad=0.4*, *borderpad=0.5*, *prop=None*, *frameon=True*, ***kwargs*)

An anchored container with a fixed size and fillable DrawingArea.

Artists added to the *drawing_area* will have their coordinates interpreted as pixels. Any transformations set on the artists will be overridden.

Parameters

width, height

[float] width and height of the container, in pixels.

xdescent, ydescent

[float] descent of the container in the x- and y- direction, in pixels.

loc

[int] Location of this artist. Valid location codes are:

```
'upper right'  : 1,
'upper left'   : 2,
'lower left'   : 3,
'lower right'  : 4,
'right'        : 5,
'center left'  : 6,
'center right' : 7,
'lower center' : 8,
'upper center' : 9,
'center'       : 10
```

pad

[float, default: 0.4] Padding around the child objects, in fraction of the font size.

borderpad

[float, default: 0.5] Border padding, in fraction of the font size.

prop

[*matplotlib.font_manager.FontProperties*, optional] Font property used as a reference for paddings.

frameon

[bool, default: True] If True, draw a box around this artists.

****kwargs**

Keywords arguments to pass to *matplotlib.offsetbox.AnchoredOffsetbox*.

Examples

To display blue and red circles of different sizes in the upper right of an axes *ax*:

```
>>> ada = AnchoredDrawingArea(20, 20, 0, 0,
...                           loc='upper right', frameon=False)
>>> ada.drawing_area.add_artist(Circle((10, 10), 10, fc="b"))
>>> ada.drawing_area.add_artist(Circle((30, 10), 5, fc="r"))
>>> ax.add_artist(ada)
```

Attributes

drawing_area

[*matplotlib.offsetbox.DrawingArea*] A container for artists to display.

```
__module__ = 'mpl_toolkits.axes_grid1.anchored_artists'
```

Examples using `mpl_toolkits.axes_grid1.anchored_artists.AnchoredDrawingArea`

- sphx_glr_gallery_axes_grid1_simple_anchored_artists.py
- sphx_glr_gallery_userdemo_anchored_box02.py

mpl_toolkits.axes_grid1.anchored_artists.AnchoredEllipse

class mpl_toolkits.axes_grid1.anchored_artists.**AnchoredEllipse**(*transform*, *width*, *height*, *angle*, *loc*, *pad=0.1*, *borderpad=0.1*, *prop=None*, *frameon=True*, ***kwargs*)

Bases: *matplotlib.offsetbox.AnchoredOffsetbox*

Draw an anchored ellipse of a given size.

Parameters

transform

[*matplotlib.transforms.Transform*] The transformation object for the coordinate system in use, i.e., `matplotlib.axes.Axes.transData`.

width, height

[float] Width and height of the ellipse, given in coordinates of *transform*.

angle

[float] Rotation of the ellipse, in degrees, anti-clockwise.

loc

[int] Location of this size bar. Valid location codes are:

```
'upper right'  : 1,
'upper left'   : 2,
'lower left'   : 3,
'lower right'  : 4,
'right'        : 5,
'center left'  : 6,
'center right' : 7,
'lower center' : 8,
'upper center' : 9,
'center'       : 10
```

pad

[float, optional] Padding around the ellipse, in fraction of the font size. Defaults to 0.1.

borderpad

[float, default: 0.1] Border padding, in fraction of the font size.

frameon

[bool, default: True] If True, draw a box around the ellipse.

prop

[*matplotlib.font_manager.FontProperties*, optional] Font property used as a reference for paddings.

****kwargs**

Keyworded arguments to pass to *matplotlib.offsetbox.AnchoredOffsetbox*.

Attributes

ellipse

[*matplotlib.patches.Ellipse*] Ellipse patch drawn.

`__init__`(*self*, *transform*, *width*, *height*, *angle*, *loc*, *pad=0.1*, *borderpad=0.1*, *prop=None*, *frameon=True*, ***kwargs*)

Draw an anchored ellipse of a given size.

Parameters

transform

[*matplotlib.transforms.Transform*] The transformation object for the coordinate system in use, i.e., `matplotlib.axes.Axes.transData`.

width, height

[float] Width and height of the ellipse, given in coordinates of *transform*.

angle

[float] Rotation of the ellipse, in degrees, anti-clockwise.

loc

[int] Location of this size bar. Valid location codes are:

```
'upper right'  : 1,
'upper left'   : 2,
'lower left'   : 3,
'lower right'  : 4,
'right'        : 5,
'center left'  : 6,
'center right' : 7,
'lower center' : 8,
'upper center' : 9,
'center'       : 10
```

pad

[float, optional] Padding around the ellipse, in fraction of the font size. Defaults to 0.1.

borderpad

[float, default: 0.1] Border padding, in fraction of the font size.

frameon

[bool, default: True] If True, draw a box around the ellipse.

prop

[*matplotlib.font_manager.FontProperties*, optional] Font property used as a reference for paddings.

****kwargs**

Keyworded arguments to pass to *matplotlib.offsetbox.AnchoredOffsetbox*.

Attributes

ellipse

[*matplotlib.patches.Ellipse*] Ellipse patch drawn.

```
__module__ = 'mpl_toolkits.axes_grid1.anchored_artists'
```

Examples using `mpl_toolkits.axes_grid1.anchored_artists.AnchoredEllipse`

- sphx_glr_gallery_axes_grid1_simple_anchored_artists.py

mpl_toolkits.axes_grid1.anchored_artists.AnchoredSizeBar

class mpl_toolkits.axes_grid1.anchored_artists.**AnchoredSizeBar**(*transform*, *size*, *label*, *loc*, *pad=0.1*, *borderpad=0.1*, *sep=2*, *frameon=True*, *size_vertical=0*, *color='black'*, *label_top=False*, *fontproperties=None*, *fill_bar=None*, ***kwargs*)

Bases: *matplotlib.offsetbox.AnchoredOffsetbox*

Draw a horizontal scale bar with a center-aligned label underneath.

Parameters

transform

[*matplotlib.transforms.Transform*] The transformation object for the coordinate system in use, i.e., `matplotlib.axes.Axes.transData`.

size

[float] Horizontal length of the size bar, given in coordinates of *transform*.

label

[str] Label to display.

loc

[int] Location of this size bar. Valid location codes are:

```
'upper right'  : 1,
'upper left'   : 2,
'lower left'   : 3,
'lower right'  : 4,
'right'        : 5,
'center left'  : 6,
'center right' : 7,
'lower center' : 8,
'upper center' : 9,
'center'       : 10
```

pad

[float, default: 0.1] Padding around the label and size bar, in fraction of the font size.

borderpad

[float, default: 0.1] Border padding, in fraction of the font size.

sep

[float, default: 2] Separation between the label and the size bar, in points.

frameon

[bool, default: True] If True, draw a box around the horizontal bar and label.

size_vertical

[float, default: 0] Vertical length of the size bar, given in coordinates of *transform*.

color

[str, default: 'black'] Color for the size bar and label.

label_top

[bool, default: False] If True, the label will be over the size bar.

fontproperties

[*matplotlib.font_manager.FontProperties*, optional] Font properties for the label text.

fill_bar

[bool, optional] If True and if size_vertical is nonzero, the size bar will be filled in with the color specified by the size bar. Defaults to True if `size_vertical` is greater than zero and False otherwise.

****kwargs**

Keyworded arguments to pass to *matplotlib.offsetbox.AnchoredOffsetbox*.

Notes

If *prop* is passed as a keyworded argument, but *fontproperties* is not, then *prop* is be assumed to be the intended *fontproperties*. Using both *prop* and *fontproperties* is not supported.

Examples

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> from mpl_toolkits.axes_grid1.anchored_artists import (
...     AnchoredSizeBar)
>>> fig, ax = plt.subplots()
>>> ax.imshow(np.random.random((10, 10)))
>>> bar = AnchoredSizeBar(ax.transData, 3, '3 data units', 4)
>>> ax.add_artist(bar)
>>> fig.show()
```

Using all the optional parameters

```
>>> import matplotlib.font_manager as fm
>>> fontprops = fm.FontProperties(size=14, family='monospace')
>>> bar = AnchoredSizeBar(ax.transData, 3, '3 units', 4, pad=0.5,
...                       sep=5, borderpad=0.5, frameon=False,
...                       size_vertical=0.5, color='white',
...                       fontproperties=fontprops)
```

Attributes

size_bar

[*matplotlib.offsetbox.AuxTransformBox*] Container for the size bar.

txt_label

[*matplotlib.offsetbox.TextArea*] Container for the label of the size bar.

`__init__`(*self*, *transform*, *size*, *label*, *loc*, *pad=0.1*, *borderpad=0.1*, *sep=2*, *frameon=True*, *size_vertical=0*, *color='black'*, *label_top=False*, *fontproperties=None*, *fill_bar=None*, ***kwargs*)

Draw a horizontal scale bar with a center-aligned label underneath.

Parameters

transform

[*matplotlib.transforms.Transform*] The transformation object for the coordinate system in use, i.e., `matplotlib.axes.Axes.transData`.

size

[float] Horizontal length of the size bar, given in coordinates of *transform*.

label

[str] Label to display.

loc

[int] Location of this size bar. Valid location codes are:

```
'upper right'  : 1,
'upper left'   : 2,
'lower left'   : 3,
'lower right'  : 4,
'right'        : 5,
'center left'  : 6,
'center right' : 7,
'lower center' : 8,
'upper center' : 9,
'center'       : 10
```

pad

[float, default: 0.1] Padding around the label and size bar, in fraction of the font size.

borderpad

[float, default: 0.1] Border padding, in fraction of the font size.

sep

[float, default: 2] Separation between the label and the size bar, in points.

frameon

[bool, default: True] If True, draw a box around the horizontal bar and label.

size_vertical

[float, default: 0] Vertical length of the size bar, given in coordinates of *transform*.

color

[str, default: 'black'] Color for the size bar and label.

label_top

[bool, default: False] If True, the label will be over the size bar.

fontproperties

[*matplotlib.font_manager.FontProperties*, optional] Font properties for the label text.

fill_bar

[bool, optional] If True and if size_vertical is nonzero, the size bar will be filled in with the color specified by the size bar. De-

faults to True if `size_vertical` is greater than zero and False otherwise.

****kwargs**

Keyworded arguments to pass to *matplotlib.offsetbox.AnchoredOffsetbox*.

Notes

If *prop* is passed as a keyworded argument, but *fontproperties* is not, then *prop* is be assumed to be the intended *fontproperties*. Using both *prop* and *fontproperties* is not supported.

Examples

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> from mpl_toolkits.axes_grid1.anchored_artists import (
...     AnchoredSizeBar)
>>> fig, ax = plt.subplots()
>>> ax.imshow(np.random.random((10, 10)))
>>> bar = AnchoredSizeBar(ax.transData, 3, '3 data units', 4)
>>> ax.add_artist(bar)
>>> fig.show()
```

Using all the optional parameters

```
>>> import matplotlib.font_manager as fm
>>> fontprops = fm.FontProperties(size=14, family='monospace')
>>> bar = AnchoredSizeBar(ax.transData, 3, '3 units', 4, pad=0.5,
...                       sep=5, borderpad=0.5, frameon=False,
...                       size_vertical=0.5, color='white',
...                       fontproperties=fontprops)
```

Attributes

size_bar

[*matplotlib.offsetbox.AuxTransformBox*] Container for the size bar.

txt_label

[*matplotlib.offsetbox.TextArea*] Container for the label of the size bar.

```
__module__ = 'mpl_toolkits.axes_grid1.anchored_artists'
```

Examples using `mpl_toolkits.axes_grid1.anchored_artists.AnchoredSizeBar`

- sphx_glr_gallery_axes_grid1_inset_locator_demo2.py
- sphx_glr_gallery_axes_grid1_simple_anchored_artists.py

mpl_toolkits.axes_grid1.axes_divider

Helper classes to adjust the positions of multiple axes at drawing time.

Classes

AxesDivider(axes[, xref, yref])	Divider based on the pre-existing axes.
AxesLocator(axes_divider, nx, ny[, nx1, ny1])	A simple callable object, initialized with AxesDivider class, returns the position and size of the given cell.
Divider(fig, pos, horizontal, vertical[, ...])	An Axes positioning class.
HBoxDivider(fig, *args[, horizontal, ...])	**Parameters**
SubplotDivider(fig, *args[, horizontal, ...])	The Divider class whose rectangle area is specified as a subplot geometry.
VBoxDivider(fig, *args[, horizontal, ...])	The Divider class whose rectangle area is specified as a subplot geometry.

mpl_toolkits.axes_grid1.axes_divider.AxesDivider

class mpl_toolkits.axes_grid1.axes_divider.**AxesDivider**(*axes*, *xref=None*, *yref=None*)

Bases: *mpl_toolkits.axes_grid1.axes_divider.Divider*

Divider based on the pre-existing axes.

Parameters

axes

[*Axes*]

xref

yref

__init__(*self*, *axes*, *xref=None*, *yref=None*)

Parameters

axes

[*Axes*]

xref

yref

`__module__ = 'mpl_toolkits.axes_grid1.axes_divider'`

`append_axes`(*self*, *position*, *size*, *pad=None*, *add_to_figure=True*, ***kwargs*)
Create an axes at the given *position* with the same height (or width) of the main axes.

position

["left"|"right"|"bottom"|"top"]

size and *pad* should be axes_grid.axes_size compatible.

`get_anchor`(*self*)
Return the anchor.

`get_aspect`(*self*)
Return aspect.

`get_position`(*self*)
Return the position of the rectangle.

`get_subplotspec`(*self*)

`new_horizontal`(*self*, *size*, *pad=None*, *pack_start=False*, ***kwargs*)
Add a new axes on the right (or left) side of the main axes.

Parameters

size

[*axes_size* or float or str] A width of the axes. If float or string is given, *from_any* function is used to create the size, with *ref_size* set to AxesX instance of the current axes.

pad

[*axes_size* or float or str] Pad between the axes. It takes same argument as *size*.

pack_start

[bool] If False, the new axes is appended at the end of the list, i.e., it became the right-most axes. If True, it is inserted at the start of the list, and becomes the left-most axes.

****kwargs**

All extra keywords arguments are passed to the created axes. If *axes_class* is given, the new axes will be created as an instance of the given class. Otherwise, the same class of the main axes will be used.

`new_vertical`(*self*, *size*, *pad=None*, *pack_start=False*, ***kwargs*)

Add a new axes on the top (or bottom) side of the main axes.

Parameters

size

[*axes_size* or float or str] A height of the axes. If float or string is given, *from_any* function is used to create the size, with *ref_size* set to AxesX instance of the current axes.

pad

[*axes_size* or float or str] Pad between the axes. It takes same argument as *size*.

pack_start

[bool] If False, the new axes is appended at the end of the list, i.e., it became the right-most axes. If True, it is inserted at the start of the list, and becomes the left-most axes.

****kwargs**

All extra keywords arguments are passed to the created axes. If *axes_class* is given, the new axes will be created as an instance of the given class. Otherwise, the same class of the main axes will be used.

Examples using `mpl_toolkits.axes_grid1.axes_divider.AxesDivider`

- sphx_glr_gallery_axes_grid1_demo_colorbar_with_axes_divider.py
- sphx_glr_gallery_axes_grid1_make_room_for_ylabel_using_axesgrid.py
- sphx_glr_gallery_axes_grid1_scatter_hist_locatable_axes.py
- sphx_glr_gallery_axes_grid1_simple_colorbar.py
- *Tight Layout guide*

mpl_toolkits.axes_grid1.axes_divider.AxesLocator

class mpl_toolkits.axes_grid1.axes_divider.**AxesLocator**(*axes_divider*, *nx*, *ny*, *nx1=None*, *ny1=None*)

Bases: object

A simple callable object, initialized with AxesDivider class, returns the position and size of the given cell.

Parameters

axes_divider

[AxesDivider]

nx, nx1

[int] Integers specifying the column-position of the cell. When *nx1* is None, a single *nx*-th column is specified. Otherwise location of columns spanning between *nx* to *nx1* (but excluding *nx1*-th column) is specified.

ny, ny1

[int] Same as *nx* and *nx1*, but for row positions.

__call__(*self*, *axes*, *renderer*)

Call self as a function.

__dict__ = mappingproxy({'__module__': 'mpl_toolkits.axes_grid1.axes_divider', '__doc__':

__init__(*self*, *axes_divider*, *nx*, *ny*, *nx1=None*, *ny1=None*)

Parameters

axes_divider

[AxesDivider]

nx, nx1

[int] Integers specifying the column-position of the cell. When *nx1* is None, a single *nx*-th column is specified. Otherwise location of columns spanning between *nx* to *nx1* (but excluding *nx1*-th column) is specified.

ny, ny1

[int] Same as *nx* and *nx1*, but for row positions.

__module__ = 'mpl_toolkits.axes_grid1.axes_divider'

__weakref__

list of weak references to the object (if defined)

get_subplotspec(*self*)

Examples using `mpl_toolkits.axes_grid1.axes_divider.AxesLocator`

mpl_toolkits.axes_grid1.axes_divider.Divider

class mpl_toolkits.axes_grid1.axes_divider.Divider(*fig*, *pos*, *horizontal*, *vertical*, *aspect=None*, *anchor='C'*)

Bases: `object`

An Axes positioning class.

The divider is initialized with lists of horizontal and vertical sizes (*mpl_toolkits.axes_grid1.axes_size*) based on which a given rectangular area will be divided.

The *new_locator* method then creates a callable object that can be used as the *axes_locator* of the axes.

Parameters

fig

[Figure]

pos

[tuple of 4 floats] Position of the rectangle that will be divided.

horizontal

[list of *axes_size*] Sizes for horizontal division.

vertical

[list of *axes_size*] Sizes for vertical division.

aspect

[bool] Whether overall rectangular area is reduced so that the relative part of the horizontal and vertical scales have the same scale.

anchor

[{'C', 'SW', 'S', 'SE', 'E', 'NE', 'N', 'NW', 'W'}] Placement of the reduced rectangle, when *aspect* is True.

__dict__ = mappingproxy({'__module__': 'mpl_toolkits.axes_grid1.axes_divider', '__doc__':

__init__(*self*, *fig*, *pos*, *horizontal*, *vertical*, *aspect=None*, *anchor='C'*)

Parameters

fig

[Figure]

pos

[tuple of 4 floats] Position of the rectangle that will be divided.

horizontal

[list of *axes_size*] Sizes for horizontal division.

vertical

[list of *axes_size*] Sizes for vertical division.

aspect

[bool] Whether overall rectangular area is reduced so that the relative part of the horizontal and vertical scales have the same scale.

anchor

[{'C', 'SW', 'S', 'SE', 'E', 'NE', 'N', 'NW', 'W'}] Placement of the reduced rectangle, when *aspect* is True.

`__module__ = 'mpl_toolkits.axes_grid1.axes_divider'`

`__weakref__`
list of weak references to the object (if defined)

`add_auto_adjustable_area`(*self*, *use_axes*, *pad=0.1*, *adjust_dirs=None*)

`append_size`(*self*, *position*, *size*)

`get_anchor`(*self*)
Return the anchor.

`get_aspect`(*self*)
Return aspect.

`get_horizontal`(*self*)
Return horizontal sizes.

`get_horizontal_sizes`(*self*, *renderer*)

`get_locator`(*self*)

`get_position`(*self*)
Return the position of the rectangle.

`get_position_runtime`(*self*, *ax*, *renderer*)

`get_vertical`(*self*)
Return vertical sizes.

`get_vertical_sizes`(*self*, *renderer*)

`get_vsize_hsize`(*self*)

`locate`(*self*, *nx*, *ny*, *nx1=None*, *ny1=None*, *axes=None*, *renderer=None*)

Parameters

nx, nx1

[int] Integers specifying the column-position of the cell. When *nx1* is None, a single *nx*-th column is specified. Otherwise location of columns spanning between *nx* to *nx1* (but excluding *nx1*-th column) is specified.

ny, ny1

[int] Same as *nx* and *nx1*, but for row positions.

axes

renderer

`new_locator`(*self*, *nx*, *ny*, *nx1=None*, *ny1=None*)

Return a new *AxesLocator* for the specified cell.

Parameters

nx, nx1

[int] Integers specifying the column-position of the cell. When *nx1* is None, a single *nx*-th column is specified. Otherwise location of columns spanning between *nx* to *nx1* (but excluding *nx1*-th column) is specified.

ny, ny1

[int] Same as *nx* and *nx1*, but for row positions.

`set_anchor`(*self*, *anchor*)

Parameters

anchor

[{'C', 'SW', 'S', 'SE', 'E', 'NE', 'N', 'NW', 'W'}]

anchor position

value	description
'C'	Center
'SW'	bottom left
'S'	bottom
'SE'	bottom right
'E'	right
'NE'	top right
'N'	top
'NW'	top left
'W'	left

`set_aspect`(*self*, *aspect=False*)

Parameters

aspect

[bool]

`set_horizontal`(*self*, *h*)

Parameters

h

[list of *axes_size*] sizes for horizontal division

`set_locator`(*self*, *_locator*)

`set_position`(*self*, *pos*)

Set the position of the rectangle.

Parameters

pos

[tuple of 4 floats] position of the rectangle that will be divided

`set_vertical`(*self*, *v*)

Parameters

v

[list of *axes_size*] sizes for vertical division

Examples using `mpl_toolkits.axes_grid1.axes_divider.Divider`

mpl_toolkits.axes_grid1.axes_divider.HBoxDivider

`class mpl_toolkits.axes_grid1.axes_divider.HBoxDivider(`*fig*, **args*, *horizontal=None*, *vertical=None*, *aspect=None*, *anchor='C'*`)`

Bases: *mpl_toolkits.axes_grid1.axes_divider.SubplotDivider*

Parameters

fig

[*matplotlib.figure.Figure*]

***args**

[tuple (*nrows*, *ncols*, *index*) or int] The array of subplots in the figure has dimensions `(nrows, ncols)`, and *index* is the index of the subplot being created. *index* starts at 1 in the upper left corner and increases to the right.

If *nrows*, *ncols*, and *index* are all single digit numbers, then *args* can be passed as a single 3-digit number (e.g. 234 for (2, 3, 4)).

`__module__ = 'mpl_toolkits.axes_grid1.axes_divider'`

`locate(`*self*, *nx*, *ny*, *nx1=None*, *ny1=None*, *axes=None*, *renderer=None*`)`

Parameters

axes_divider

[AxesDivider]

nx, nx1

[int] Integers specifying the column-position of the cell. When *nx1* is None, a single *nx*-th column is specified. Otherwise location of columns spanning between *nx* to *nx1* (but excluding *nx1*-th column) is specified.

ny, ny1

[int] Same as *nx* and *nx1*, but for row positions.

axes

renderer

`new_locator(`*self*, *nx*, *nx1=None*`)`

Create a new `AxesLocator` for the specified cell.

Parameters

nx, nx1

[int] Integers specifying the column-position of the cell. When *nx1* is None, a single *nx*-th column is specified. Otherwise location of columns spanning between *nx* to *nx1* (but excluding *nx1*-th column) is specified.

ny, ny1

[int] Same as *nx* and *nx1*, but for row positions.

Examples using `mpl_toolkits.axes_grid1.axes_divider.HBoxDivider`

- sphx_glr_gallery_axes_grid1_demo_axes_hbox_divider.py

mpl_toolkits.axes_grid1.axes_divider.SubplotDivider

`class mpl_toolkits.axes_grid1.axes_divider.SubplotDivider(`*fig*, **args*, *horizontal=None*, *vertical=None*, *aspect=None*, *anchor='C'*`)`

Bases: *mpl_toolkits.axes_grid1.axes_divider.Divider*

The Divider class whose rectangle area is specified as a subplot geometry.

Parameters

fig

[*matplotlib.figure.Figure*]

***args**

[tuple (*nrows*, *ncols*, *index*) or int] The array of subplots in the figure has dimensions `(nrows, ncols)`, and *index* is the index of the subplot being created. *index* starts at 1 in the upper left corner and increases to the right.

If *nrows*, *ncols*, and *index* are all single digit numbers, then *args* can be passed as a single 3-digit number (e.g. 234 for (2, 3, 4)).

`__init__(`*self*, *fig*, **args*, *horizontal=None*, *vertical=None*, *aspect=None*, *anchor='C'*`)`

Parameters

fig

[*matplotlib.figure.Figure*]

***args**

[tuple (*nrows*, *ncols*, *index*) or int] The array of subplots in the figure has dimensions `(nrows, ncols)`, and *index* is the index of the subplot being created. *index* starts at 1 in the upper left corner and increases to the right.

If *nrows*, *ncols*, and *index* are all single digit numbers, then *args* can be passed as a single 3-digit number (e.g. 234 for (2, 3, 4)).

`__module__ = 'mpl_toolkits.axes_grid1.axes_divider'`

`change_geometry`(*self*, *numrows*, *numcols*, *num*)
Change subplot geometry, e.g., from (1, 1, 1) to (2, 2, 3).

`get_geometry`(*self*)
Get the subplot geometry, e.g., (2, 2, 3).

`get_position`(*self*)
Return the bounds of the subplot box.

`get_subplotspec`(*self*)
Get the SubplotSpec instance.

`set_subplotspec`(*self*, *subplotspec*)
Set the SubplotSpec instance.

`update_params`(*self*)
Update the subplot position from fig.subplotpars.

Examples using `mpl_toolkits.axes_grid1.axes_divider.SubplotDivider`

mpl_toolkits.axes_grid1.axes_divider.VBoxDivider

`class mpl_toolkits.axes_grid1.axes_divider.VBoxDivider`(*fig*, **args*, *horizontal=None*, *vertical=None*, *aspect=None*, *anchor='C'*)

Bases: *mpl_toolkits.axes_grid1.axes_divider.HBoxDivider*

The Divider class whose rectangle area is specified as a subplot geometry.

Parameters

fig

[*matplotlib.figure.Figure*]

***args**

[tuple (*nrows*, *ncols*, *index*) or int] The array of subplots in the figure has dimensions `(nrows, ncols)`, and *index* is the index of the subplot being created. *index* starts at 1 in the upper left corner and increases to the right.

If *nrows*, *ncols*, and *index* are all single digit numbers, then *args* can be passed as a single 3-digit number (e.g. 234 for (2, 3, 4)).

`__module__ = 'mpl_toolkits.axes_grid1.axes_divider'`

`locate`(*self*, *nx*, *ny*, *nx1=None*, *ny1=None*, *axes=None*, *renderer=None*)

Parameters

axes_divider

[AxesDivider]

nx, nx1

[int] Integers specifying the column-position of the cell. When *nx1* is None, a single *nx*-th column is specified. Otherwise location of columns spanning between *nx* to *nx1* (but excluding *nx1*-th column) is specified.

ny, ny1

[int] Same as *nx* and *nx1*, but for row positions.

axes

renderer

`new_locator`(*self*, *ny*, *ny1=None*)

Create a new `AxesLocator` for the specified cell.

Parameters

ny, ny1

[int] Integers specifying the row-position of the cell. When *ny1* is None, a single *ny*-th row is specified. Otherwise location of rows spanning between *ny* to *ny1* (but excluding *ny1*-th row) is specified.

Examples using `mpl_toolkits.axes_grid1.axes_divider.VBoxDivider`

Functions

make_axes_area_auto_adjustable(ax[, ...])	
make_axes_locatable(axes)	

mpl_toolkits.axes_grid1.axes_divider.make_axes_area_auto_adjustable

mpl_toolkits.axes_grid1.axes_divider.**make_axes_area_auto_adjustable**(*ax*, *use_axes=None*, *pad=0.1*, *adjust_dirs=None*)

Examples using `mpl_toolkits.axes_grid1.axes_divider.make_axes_area_auto_adjustable`

- sphx_glr_gallery_axes_grid1_make_room_for_ylabel_using_axesgrid.py

mpl_toolkits.axes_grid1.axes_divider.make_axes_locatable

mpl_toolkits.axes_grid1.axes_divider.**make_axes_locatable**(*axes*)

Examples using `mpl_toolkits.axes_grid1.axes_divider.make_axes_locatable`

mpl_toolkits.axes_grid1.axes_grid

Classes

AxesGrid	alias of *mpl_toolkits.axes_grid1.axes_grid.ImageGrid*
CbarAxes(*args, orientation, **kwargs)	
CbarAxesBase(*args, orientation, **kwargs)	
Grid(fig, rect, nrows_ncols[, ngrids, ...])	A grid of Axes.
ImageGrid(fig, rect, nrows_ncols[, ngrids, ...])	**Parameters**

mpl_toolkits.axes_grid1.axes_grid.AxesGrid

mpl_toolkits.axes_grid1.axes_grid.**AxesGrid**
alias of *mpl_toolkits.axes_grid1.axes_grid.ImageGrid*

mpl_toolkits.axes_grid1.axes_grid.CbarAxes

class mpl_toolkits.axes_grid1.axes_grid.**CbarAxes**(**args*, *orientation*, ***kwargs*)
Bases: *mpl_toolkits.axes_grid1.axes_grid.CbarAxesBase*, *mpl_toolkits.axes_grid1.mpl_axes.Axes*

__module__ = 'mpl_toolkits.axes_grid1.axes_grid'

Examples using mpl_toolkits.axes_grid1.axes_grid.CbarAxes

mpl_toolkits.axes_grid1.axes_grid.CbarAxesBase

class mpl_toolkits.axes_grid1.axes_grid.**CbarAxesBase**(**args*, *orientation*, ***kwargs*)
Bases: object

__dict__ = mappingproxy({'__module__': 'mpl_toolkits.axes_grid1.axes_grid', '__init__': <

__init__(*self*, **args*, *orientation*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'mpl_toolkits.axes_grid1.axes_grid'

__weakref__
list of weak references to the object (if defined)

property **cbid**

cla(*self*)

colorbar(*self*, *mappable*, *, *ticks=None*, ***kwargs*)

property **locator**

toggle_label(*self*, *b*)

Examples using `mpl_toolkits.axes_grid1.axes_grid.CbarAxesBase`

mpl_toolkits.axes_grid1.axes_grid.Grid

`class mpl_toolkits.axes_grid1.axes_grid.Grid`(*fig*, *rect*, *nrows_ncols*, *ngrids=None*, *direction='row'*, *axes_pad=0.02*, *add_all=<deprecated parameter>*, *share_all=False*, *share_x=True*, *share_y=True*, *label_mode='L'*, *axes_class=None*, *, *aspect=False*)

Bases: `object`

A grid of Axes.

In Matplotlib, the axes location (and size) is specified in normalized figure coordinates. This may not be ideal for images that needs to be displayed with a given aspect ratio; for example, it is difficult to display multiple images of a same size with some fixed padding between them. AxesGrid can be used in such case.

Parameters

fig

[*Figure*] The parent figure.

rect

[(float, float, float, float) or int] The axes position, as a `(left, bottom, width, height)` tuple or as a three-digit subplot position code (e.g., "121").

nrows_ncols

[(int, int)] Number of rows and columns in the grid.

ngrids

[int or None, default: None] If not None, only the first *ngrids* axes in the grid are created.

direction

[{"row", "column"}, default: "row"] Whether axes are created in row-major ("row by row") or column-major order ("column by column").

axes_pad

[float or (float, float), default: 0.02] Padding or (horizontal padding, vertical padding) between axes, in inches.

add_all

[bool, default: True] Whether to add the axes to the figure using *Figure.add_axes*. This parameter is deprecated.

share_all

[bool, default: False] Whether all axes share their x- and y-axis. Overrides *share_x* and *share_y*.

share_x

[bool, default: True] Whether all axes of a column share their x-axis.

share_y

[bool, default: True] Whether all axes of a row share their y-axis.

label_mode

[{"L", "1", "all"}, default: "L"] Determines which axes will get tick labels:

- "L": All axes on the left column get vertical tick labels; all axes on the bottom row get horizontal tick labels.
- "1": Only the bottom left axes is labelled.
- "all": all axes are labelled.

axes_class

[subclass of *matplotlib.axes.Axes*, default: None]

aspect

[bool, default: False] Whether the axes aspect ratio follows the aspect ratio of the data limits.

`__dict__ = mappingproxy({'__module__': 'mpl_toolkits.axes_grid1.axes_grid', '__doc__': '\`

`__getitem__`(*self*, *i*)

`__init__`(*self*, *fig*, *rect*, *nrows_ncols*, *ngrids=None*, *direction='row'*, *axes_pad=0.02*, *add_all=<deprecated parameter>*, *share_all=False*, *share_x=True*, *share_y=True*, *label_mode='L'*, *axes_class=None*, ***, *aspect=False*)

Parameters

fig

[*Figure*] The parent figure.

rect

[(float, float, float, float) or int] The axes position, as a `(left, bottom, width, height)` tuple or as a three-digit subplot position code (e.g., "121").

nrows_ncols

[(int, int)] Number of rows and columns in the grid.

ngrids

[int or None, default: None] If not None, only the first *ngrids* axes in the grid are created.

direction

[{"row", "column"}, default: "row"] Whether axes are created in row-major ("row by row") or column-major order ("column by column").

axes_pad

[float or (float, float), default: 0.02] Padding or (horizontal padding, vertical padding) between axes, in inches.

add_all

[bool, default: True] Whether to add the axes to the figure using *Figure.add_axes*. This parameter is deprecated.

share_all

[bool, default: False] Whether all axes share their x- and y-axis. Overrides *share_x* and *share_y*.

share_x

[bool, default: True] Whether all axes of a column share their x-axis.

share_y

[bool, default: True] Whether all axes of a row share their y-axis.

label_mode

[{"L", "1", "all"}, default: "L"] Determines which axes will get tick labels:

- "L": All axes on the left column get vertical tick labels; all axes on the bottom row get horizontal tick labels.
- "1": Only the bottom left axes is labelled.
- "all": all axes are labelled.

axes_class

[subclass of *matplotlib.axes.Axes*, default: None]

aspect

[bool, default: False] Whether the axes aspect ratio follows the aspect ratio of the data limits.

`__len__`(*self*)

`__module__ = 'mpl_toolkits.axes_grid1.axes_grid'`

`__weakref__`
list of weak references to the object (if defined)

`get_aspect`(*self*)
Return the aspect of the SubplotDivider.

`get_axes_locator`(*self*)

`get_axes_pad`(*self*)
Return the axes padding.

Returns

hpad, vpad

Padding (horizontal pad, vertical pad) in inches.

`get_divider`(*self*)

`get_geometry`(*self*)
Return the number of rows and columns of the grid as (nrows, ncols).

`get_vsize_hsize`(*self*)

`set_aspect`(*self*, *aspect*)
Set the aspect of the SubplotDivider.

`set_axes_locator`(*self*, *locator*)

`set_axes_pad`(*self*, *axes_pad*)
Set the padding between the axes.

Parameters

axes_pad

[(float, float)] The padding (horizontal pad, vertical pad) in inches.

`set_label_mode`(*self*, *mode*)
Define which axes have tick labels.

Parameters

mode

[{"L", "1", "all"}] The label mode:

- "L": All axes on the left column get vertical tick labels; all axes on the bottom row get horizontal tick labels.
- "1": Only the bottom left axes is labelled.

- "all": all axes are labelled.

Examples using `mpl_toolkits.axes_grid1.axes_grid.Grid`

mpl_toolkits.axes_grid1.axes_grid.ImageGrid

class mpl_toolkits.axes_grid1.axes_grid.**ImageGrid**(*fig*, *rect*, *nrows_ncols*, *ngrids=None*, *direction='row'*, *axes_pad=0.02*, *add_all=<deprecated parameter>*, *share_all=False*, *aspect=True*, *label_mode='L'*, *cbar_mode=None*, *cbar_location='right'*, *cbar_pad=None*, *cbar_size='5%'*, *cbar_set_cax=True*, *axes_class=None*)

Bases: *mpl_toolkits.axes_grid1.axes_grid.Grid*

Parameters

fig

[*Figure*] The parent figure.

rect

[(float, float, float, float) or int] The axes position, as a `(left, bottom, width, height)` tuple or as a three-digit subplot position code (e.g., "121").

nrows_ncols

[(int, int)] Number of rows and columns in the grid.

ngrids

[int or None, default: None] If not None, only the first *ngrids* axes in the grid are created.

direction

[{"row", "column"}, default: "row"] Whether axes are created in row-major ("row by row") or column-major order ("column by column"). This also affects the order in which axes are accessed using indexing (`grid[index]`).

axes_pad

[float or (float, float), default: 0.02in] Padding or (horizontal padding, vertical padding) between axes, in inches.

add_all

[bool, default: True] Whether to add the axes to the figure using *Figure.add_axes*. This parameter is deprecated.

share_all

[bool, default: False] Whether all axes share their x- and y-axis.

aspect

[bool, default: True] Whether the axes aspect ratio follows the aspect ratio of the data limits.

label_mode

[{"L", "1", "all"}, default: "L"] Determines which axes will get tick labels:

- "L": All axes on the left column get vertical tick labels; all axes on the bottom row get horizontal tick labels.
- "1": Only the bottom left axes is labelled.
- "all": all axes are labelled.

cbar_mode

[{"each", "single", "edge", None}, default: None] Whether to create a colorbar for "each" axes, a "single" colorbar for the entire grid, colorbars only for axes on the "edge" determined by *cbar_location*, or no colorbars. The colorbars are stored in the `cbar_axes` attribute.

cbar_location

[{"left", "right", "bottom", "top"}, default: "right"]

cbar_pad

[float, default: None] Padding between the image axes and the colorbar axes.

cbar_size

[size specification (see `Size.from_any`), default: "5%"] Colorbar size.

cbar_set_cax

[bool, default: True] If True, each axes in the grid has a *cax* attribute that is bound to associated *cbar_axes*.

axes_class

[subclass of *matplotlib.axes.Axes*, default: None]

`__init__`(*self*, *fig*, *rect*, *nrows_ncols*, *ngrids=None*, *direction='row'*, *axes_pad=0.02*, *add_all=<deprecated parameter>*, *share_all=False*, *aspect=True*, *label_mode='L'*, *cbar_mode=None*, *cbar_location='right'*, *cbar_pad=None*, *cbar_size='5%'*, *cbar_set_cax=True*, *axes_class=None*)

Parameters

fig

[*Figure*] The parent figure.

rect

[(float, float, float, float) or int] The axes position, as a `(left, bottom, width, height)` tuple or as a three-digit subplot position code (e.g., "121").

nrows_ncols

[(int, int)] Number of rows and columns in the grid.

ngrids

[int or None, default: None] If not None, only the first *ngrids* axes in the grid are created.

direction

[{"row", "column"}, default: "row"] Whether axes are created in row-major ("row by row") or column-major order ("column by column"). This also affects the order in which axes are accessed using indexing (`grid[index]`).

axes_pad

[float or (float, float), default: 0.02in] Padding or (horizontal padding, vertical padding) between axes, in inches.

add_all

[bool, default: True] Whether to add the axes to the figure using *Figure.add_axes*. This parameter is deprecated.

share_all

[bool, default: False] Whether all axes share their x- and y-axis.

aspect

[bool, default: True] Whether the axes aspect ratio follows the aspect ratio of the data limits.

label_mode

[{"L", "1", "all"}, default: "L"] Determines which axes will get tick labels:

- "L": All axes on the left column get vertical tick labels; all axes on the bottom row get horizontal tick labels.
- "1": Only the bottom left axes is labelled.
- "all": all axes are labelled.

cbar_mode

[{"each", "single", "edge", None}, default: None] Whether to create a colorbar for "each" axes, a "single" colorbar for the entire grid, colorbars only for axes on the "edge" determined by *cbar_location*, or no colorbars. The colorbars are stored in the `cbar_axes` attribute.

cbar_location

[{"left", "right", "bottom", "top"}, default: "right"]

cbar_pad

[float, default: None] Padding between the image axes and the colorbar axes.

cbar_size

[size specification (see `Size.from_any`), default: "5%"] Colorbar size.

cbar_set_cax

[bool, default: True] If True, each axes in the grid has a *cax* attribute that is bound to associated *cbar_axes*.

axes_class

[subclass of *matplotlib.axes.Axes*, default: None]

```
__module__ = 'mpl_toolkits.axes_grid1.axes_grid'
```

Examples using `mpl_toolkits.axes_grid1.axes_grid.ImageGrid`

mpl_toolkits.axes_grid1.axes_rgb

Classes

RGBAxes(*args[, pad, add_all])	4-panel imshow (RGB, R, G, B).
RGBAxesBase(*args[, pad, add_all])	[*Deprecated*]

mpl_toolkits.axes_grid1.axes_rgb.RGBAxes

class mpl_toolkits.axes_grid1.axes_rgb.RGBAxes(*args, pad=0, add_all=<deprecated parameter>, **kwargs)

Bases: object

4-panel imshow (RGB, R, G, B).

Layout: +---------------+-----+ | | R | + +-----+ | RGB | G | + +-----+ | | B | +---------------+-----+

Subclasses can override the _defaultAxesClass attribute.

Attributes

RGB

[_defaultAxesClass] The axes object for the three-channel imshow.

R

[_defaultAxesClass] The axes object for the red channel imshow.

G

[_defaultAxesClass] The axes object for the green channel imshow.

B

[_defaultAxesClass] The axes object for the blue channel imshow.

Parameters

pad

[float, default: 0] fraction of the axes height to put as padding.

add_all

[bool, default: True] Whether to add the {rgb, r, g, b} axes to the figure. This parameter is deprecated.

axes_class

[matplotlib.axes.Axes]

***args**

Unpacked into axes_class() init for RGB

****kwargs**

Unpacked into axes_class() init for RGB, R, G, B axes

`__dict__ = mappingproxy({'__module__': 'mpl_toolkits.axes_grid1.axes_rgb', '__doc__': '\n`

`__init__`(*self*, **args*, *pad=0*, *add_all=<deprecated parameter>*, ***kwargs*)

Parameters

pad

[float, default: 0] fraction of the axes height to put as padding.

add_all

[bool, default: True] Whether to add the {rgb, r, g, b} axes to the figure. This parameter is deprecated.

axes_class

[matplotlib.axes.Axes]

***args**

Unpacked into axes_class() init for RGB

****kwargs**

Unpacked into axes_class() init for RGB, R, G, B axes

`__module__ = 'mpl_toolkits.axes_grid1.axes_rgb'`

`__weakref__`
list of weak references to the object (if defined)

`add_RGB_to_figure`(*self*)
[*Deprecated*] Add red, green and blue axes to the RGB composite's axes figure.

Notes

Deprecated since version 3.3.

`imshow_rgb`(*self*, *r*, *g*, *b*, ***kwargs*)
Create the four images {rgb, r, g, b}.

Parameters

r

[array-like] The red array

g

[array-like] The green array

b

[array-like] The blue array

kwargs

[imshow kwargs] kwargs get unpacked into the imshow calls for the four images

Returns

rgb

[matplotlib.image.AxesImage]

r

[matplotlib.image.AxesImage]

g

[matplotlib.image.AxesImage]

b

[matplotlib.image.AxesImage]

Examples using `mpl_toolkits.axes_grid1.axes_rgb.RGBAxes`

- sphx_glr_gallery_pyplots_whats_new_99_axes_grid.py
- sphx_glr_gallery_axes_grid1_demo_axes_rgb.py

mpl_toolkits.axes_grid1.axes_rgb.RGBAxesBase

`class mpl_toolkits.axes_grid1.axes_rgb.RGBAxesBase(*args, pad=0, add_all=<deprecated parameter>, **kwargs)`

Bases: *mpl_toolkits.axes_grid1.axes_rgb.RGBAxes*

[*Deprecated*]

Notes

Deprecated since version 3.3:

Parameters

pad

[float, default: 0] fraction of the axes height to put as padding.

add_all

[bool, default: True] Whether to add the {rgb, r, g, b} axes to the figure. This parameter is deprecated.

axes_class

[matplotlib.axes.Axes]

***args**

Unpacked into axes_class() init for RGB

****kwargs**

Unpacked into axes_class() init for RGB, R, G, B axes

`__init__`(*self*, **args*, *pad=0*, *add_all=<deprecated parameter>*, ***kwargs*)

Parameters

pad

[float, default: 0] fraction of the axes height to put as padding.

add_all

[bool, default: True] Whether to add the {rgb, r, g, b} axes to the figure. This parameter is deprecated.

axes_class

[matplotlib.axes.Axes]

***args**

Unpacked into axes_class() init for RGB

****kwargs**

Unpacked into axes_class() init for RGB, R, G, B axes

`__module__ = 'mpl_toolkits.axes_grid1.axes_rgb'`

Examples using `mpl_toolkits.axes_grid1.axes_rgb.RGBAxesBase`

- sphx_glr_gallery_pyplots_whats_new_99_axes_grid.py
- sphx_glr_gallery_axes_grid1_demo_axes_rgb.py
- sphx_glr_gallery_axes_grid1_simple_rgb.py

Functions

imshow_rgb(ax, r, g, b, **kwargs)	[*Deprecated*]
make_rgb_axes(ax[, pad, axes_class, add_all])	**Parameters**

mpl_toolkits.axes_grid1.axes_rgb.imshow_rgb

mpl_toolkits.axes_grid1.axes_rgb.**imshow_rgb**(*ax*, *r*, *g*, *b*, ***kwargs*)
[*Deprecated*]

Notes

Deprecated since version 3.3:

Examples using mpl_toolkits.axes_grid1.axes_rgb.imshow_rgb

mpl_toolkits.axes_grid1.axes_rgb.make_rgb_axes

mpl_toolkits.axes_grid1.axes_rgb.**make_rgb_axes**(*ax*, *pad=0.01*, *axes_class=None*, *add_all=<deprecated parameter>*, ***kwargs*)

Parameters

pad

[float] Fraction of the axes height.

Examples using mpl_toolkits.axes_grid1.axes_rgb.make_rgb_axes

- sphx_glr_gallery_axes_grid1_demo_axes_rgb.py

mpl_toolkits.axes_grid1.axes_size

Provides classes of simple units that will be used with AxesDivider class (or others) to determine the size of each axes. The unit classes define `get_size` method that returns a tuple of two floats, meaning relative and absolute sizes, respectively.

Note that this class is nothing more than a simple tuple of two floats. Take a look at the Divider class to see how these two values are used.

Classes

Add(a, b)	
AddList(add_list)	
AxesX(axes[, aspect, ref_ax])	Scaled size whose relative part corresponds to the data width of the *axes* multiplied by the *aspect*.
AxesY(axes[, aspect, ref_ax])	Scaled size whose relative part corresponds to the data height of the *axes* multiplied by the *aspect*.
Fixed(fixed_size)	Simple fixed size with absolute part = *fixed_size* and relative part = 0.
Fraction(fraction, ref_size)	An instance whose size is a *fraction* of the *ref_size*.
GetExtentHelper(ax, direction)	
MaxExtent(artist_list, w_or_h)	Size whose absolute part is either the largest width or the largest height of the given *artist_list*.
MaxHeight(artist_list)	Size whose absolute part is the largest height of the given *artist_list*.
MaxWidth(artist_list)	Size whose absolute part is the largest width of the given *artist_list*.
Padded(size, pad)	Return a instance where the absolute part of *size* is increase by the amount of *pad*.
Scalable	alias of *mpl_toolkits.axes_grid1.axes_size.Scaled*
Scaled(scalable_size)	Simple scaled(?) size with absolute part = 0 and relative part = *scalable_size*.
SizeFromFunc(func)	

mpl_toolkits.axes_grid1.axes_size.Add

class mpl_toolkits.axes_grid1.axes_size.Add(*a*, *b*)

Bases: mpl_toolkits.axes_grid1.axes_size._Base

__init__(*self*, *a*, *b*)

Initialize self. See help(type(self)) for accurate signature.

__module__ = 'mpl_toolkits.axes_grid1.axes_size'

get_size(*self*, *renderer*)

Examples using mpl_toolkits.axes_grid1.axes_size.Add

mpl_toolkits.axes_grid1.axes_size.AddList

class mpl_toolkits.axes_grid1.axes_size.AddList(*add_list*)

Bases: mpl_toolkits.axes_grid1.axes_size._Base

__init__(*self*, *add_list*)

Initialize self. See help(type(self)) for accurate signature.

__module__ = 'mpl_toolkits.axes_grid1.axes_size'

get_size(*self*, *renderer*)

Examples using mpl_toolkits.axes_grid1.axes_size.AddList

mpl_toolkits.axes_grid1.axes_size.AxesX

class mpl_toolkits.axes_grid1.axes_size.AxesX(*axes*, *aspect=1.0*, *ref_ax=None*)

Bases: mpl_toolkits.axes_grid1.axes_size._Base

Scaled size whose relative part corresponds to the data width of the *axes* multiplied by the *aspect*.

__init__(*self*, *axes*, *aspect=1.0*, *ref_ax=None*)

Initialize self. See help(type(self)) for accurate signature.

__module__ = 'mpl_toolkits.axes_grid1.axes_size'

get_size(*self*, *renderer*)

Examples using `mpl_toolkits.axes_grid1.axes_size.AxesX`

- sphx_glr_gallery_axes_grid1_demo_axes_hbox_divider.py
- sphx_glr_gallery_axes_grid1_simple_axes_divider3.py

mpl_toolkits.axes_grid1.axes_size.AxesY

class mpl_toolkits.axes_grid1.axes_size.**AxesY**(*axes*, *aspect=1.0*, *ref_ax=None*)

Bases: mpl_toolkits.axes_grid1.axes_size._Base

Scaled size whose relative part corresponds to the data height of the *axes* multiplied by the *aspect*.

__init__(*self*, *axes*, *aspect=1.0*, *ref_ax=None*)

Initialize self. See help(type(self)) for accurate signature.

__module__ = 'mpl_toolkits.axes_grid1.axes_size'

get_size(*self*, *renderer*)

Examples using `mpl_toolkits.axes_grid1.axes_size.AxesY`

- sphx_glr_gallery_axes_grid1_demo_axes_hbox_divider.py
- sphx_glr_gallery_axes_grid1_simple_axes_divider3.py

mpl_toolkits.axes_grid1.axes_size.Fixed

class mpl_toolkits.axes_grid1.axes_size.**Fixed**(*fixed_size*)

Bases: mpl_toolkits.axes_grid1.axes_size._Base

Simple fixed size with absolute part = *fixed_size* and relative part = 0.

__init__(*self*, *fixed_size*)

Initialize self. See help(type(self)) for accurate signature.

__module__ = 'mpl_toolkits.axes_grid1.axes_size'

get_size(*self*, *renderer*)

Examples using `mpl_toolkits.axes_grid1.axes_size.Fixed`

- sphx_glr_gallery_axes_grid1_demo_axes_hbox_divider.py
- sphx_glr_gallery_axes_grid1_demo_fixed_size_axes.py
- sphx_glr_gallery_axes_grid1_simple_axes_divider1.py
- sphx_glr_gallery_axes_grid1_simple_axes_divider3.py

mpl_toolkits.axes_grid1.axes_size.Fraction

class mpl_toolkits.axes_grid1.axes_size.**Fraction**(*fraction*, *ref_size*)

Bases: mpl_toolkits.axes_grid1.axes_size._Base

An instance whose size is a *fraction* of the *ref_size*.

```
>>> s = Fraction(0.3, AxesX(ax))
```

__init__(*self*, *fraction*, *ref_size*)

Initialize self. See help(type(self)) for accurate signature.

__module__ = 'mpl_toolkits.axes_grid1.axes_size'

get_size(*self*, *renderer*)

Examples using `mpl_toolkits.axes_grid1.axes_size.Fraction`

mpl_toolkits.axes_grid1.axes_size.GetExtentHelper

class mpl_toolkits.axes_grid1.axes_size.**GetExtentHelper**(*ax*, *direction*)

Bases: object

__call__(*self*, *renderer*)

Call self as a function.

__dict__ = mappingproxy({'__module__': 'mpl_toolkits.axes_grid1.axes_size', '_get_func_ma

__init__(*self*, *ax*, *direction*)

Initialize self. See help(type(self)) for accurate signature.

__module__ = 'mpl_toolkits.axes_grid1.axes_size'

__weakref__

list of weak references to the object (if defined)

Examples using `mpl_toolkits.axes_grid1.axes_size.GetExtentHelper`

mpl_toolkits.axes_grid1.axes_size.MaxExtent

class mpl_toolkits.axes_grid1.axes_size.**MaxExtent**(*artist_list*, *w_or_h*)

Bases: mpl_toolkits.axes_grid1.axes_size._Base

Size whose absolute part is either the largest width or the largest height of the given *artist_list*.

__init__(*self*, *artist_list*, *w_or_h*)

Initialize self. See help(type(self)) for accurate signature.

__module__ = 'mpl_toolkits.axes_grid1.axes_size'

add_artist(*self*, *a*)

get_size(*self*, *renderer*)

Examples using `mpl_toolkits.axes_grid1.axes_size.MaxExtent`

mpl_toolkits.axes_grid1.axes_size.MaxHeight

class mpl_toolkits.axes_grid1.axes_size.**MaxHeight**(*artist_list*)

Bases: *mpl_toolkits.axes_grid1.axes_size.MaxExtent*

Size whose absolute part is the largest height of the given *artist_list*.

__init__(*self*, *artist_list*)

Initialize self. See help(type(self)) for accurate signature.

__module__ = 'mpl_toolkits.axes_grid1.axes_size'

Examples using `mpl_toolkits.axes_grid1.axes_size.MaxHeight`

mpl_toolkits.axes_grid1.axes_size.MaxWidth

class mpl_toolkits.axes_grid1.axes_size.**MaxWidth**(*artist_list*)

Bases: *mpl_toolkits.axes_grid1.axes_size.MaxExtent*

Size whose absolute part is the largest width of the given *artist_list*.

__init__(*self*, *artist_list*)

Initialize self. See help(type(self)) for accurate signature.

__module__ = 'mpl_toolkits.axes_grid1.axes_size'

Examples using `mpl_toolkits.axes_grid1.axes_size.MaxWidth`

mpl_toolkits.axes_grid1.axes_size.Padded

class mpl_toolkits.axes_grid1.axes_size.**Padded**(*size*, *pad*)

Bases: `mpl_toolkits.axes_grid1.axes_size._Base`

Return a instance where the absolute part of *size* is increase by the amount of *pad*.

__init__(*self*, *size*, *pad*)

Initialize self. See help(type(self)) for accurate signature.

__module__ = 'mpl_toolkits.axes_grid1.axes_size'

get_size(*self*, *renderer*)

Examples using `mpl_toolkits.axes_grid1.axes_size.Padded`

mpl_toolkits.axes_grid1.axes_size.Scalable

mpl_toolkits.axes_grid1.axes_size.**Scalable**

alias of *mpl_toolkits.axes_grid1.axes_size.Scaled*

mpl_toolkits.axes_grid1.axes_size.Scaled

class mpl_toolkits.axes_grid1.axes_size.**Scaled**(*scalable_size*)

Bases: `mpl_toolkits.axes_grid1.axes_size._Base`

Simple scaled(?) size with absolute part = 0 and relative part = *scalable_size*.

__init__(*self*, *scalable_size*)

Initialize self. See help(type(self)) for accurate signature.

__module__ = 'mpl_toolkits.axes_grid1.axes_size'

get_size(*self*, *renderer*)

Examples using `mpl_toolkits.axes_grid1.axes_size.Scaled`

- sphx_glr_gallery_axes_grid1_demo_axes_hbox_divider.py
- sphx_glr_gallery_axes_grid1_demo_fixed_size_axes.py
- sphx_glr_gallery_axes_grid1_simple_axes_divider1.py

mpl_toolkits.axes_grid1.axes_size.SizeFromFunc

class mpl_toolkits.axes_grid1.axes_size.**SizeFromFunc**(*func*)

Bases: mpl_toolkits.axes_grid1.axes_size._Base

__init__(*self*, *func*)

Initialize self. See help(type(self)) for accurate signature.

__module__ = 'mpl_toolkits.axes_grid1.axes_size'

get_size(*self*, *renderer*)

Examples using mpl_toolkits.axes_grid1.axes_size.SizeFromFunc

Functions

from_any(size[, fraction_ref])	Create a Fixed unit when the first argument is a float, or a Fraction unit if that is a string that ends with %.

mpl_toolkits.axes_grid1.axes_size.from_any

mpl_toolkits.axes_grid1.axes_size.**from_any**(*size*, *fraction_ref=None*)

Create a Fixed unit when the first argument is a float, or a Fraction unit if that is a string that ends with %. The second argument is only meaningful when Fraction unit is created.

```
>>> a = Size.from_any(1.2) # => Size.Fixed(1.2)
>>> Size.from_any("50%", a) # => Size.Fraction(0.5, a)
```

Examples using mpl_toolkits.axes_grid1.axes_size.from_any

mpl_toolkits.axes_grid1.colorbar

Colorbar toolkit with two classes and a function:

ColorbarBase

the base class with full colorbar drawing functionality. It can be used as-is to make a colorbar for a given colormap; a mappable object (e.g., image) is not needed.

Colorbar

the derived class for use with images or contour plots.

make_axes()

a function for resizing an axes and adding a second axes suitable for a colorbar

The *colorbar()* method uses *make_axes()* and *Colorbar*; the *colorbar()* function is a thin wrapper over *colorbar()*.

class mpl_toolkits.axes_grid1.colorbar.CbarAxesLocator(*locator=None*, *extend='neither'*, *orientation='vertical'*)

CbarAxesLocator is a axes_locator for colorbar axes. It adjust the position of the axes to make a room for extended ends, i.e., the extended ends are located outside the axes area.

locator

[the bbox returned from the locator is used as a] initial axes location. If None, axes.bbox is used.

extend : same as in ColorbarBase *orientation* : same as in ColorbarBase

get_end_vertices(*self*)

Return a tuple of two vertices for the colorbar extended ends.

The first vertices is for the minimum end, and the second is for the maximum end.

get_original_position(*self*, *axes*, *renderer*)

Return the original position of the axes.

get_path_ends(*self*)

Return the paths for extended ends.

get_path_patch(*self*)

Return the path for axes patch.

class mpl_toolkits.axes_grid1.colorbar.Colorbar(*ax*, *mappable*, ***kw*)

Parameters

norm

[*matplotlib.colors.Normalize* (or subclass thereof)] The normalizing object which scales data, typically into the interval `[0, 1]`. If *None*, *norm* defaults to a *colors.Normalize* object which initializes its scaling based on the first data processed.

cmap

[str or *Colormap*] The colormap used to map normalized data values to RGBA colors.

add_lines(*self*, *CS*)

Add the lines from a non-filled *ContourSet* to the colorbar.

update_bruteforce(*self*, *mappable*)
Update the colorbar artists to reflect the change of the associated mappable.

update_normal(*self*, *mappable*)
Update solid patches, lines, etc.

This is meant to be called when the norm of the image or contour plot to which this colorbar belongs changes.

If the norm on the mappable is different than before, this resets the locator and formatter for the axis, so if these have been customized, they will need to be customized again. However, if the norm only changes values of *vmin*, *vmax* or *cmap* then the old formatter and locator will be preserved.

class mpl_toolkits.axes_grid1.colorbar.ColorbarBase(*ax*, *cmap=None*, *norm=None*, *alpha=1.0*, *values=None*, *boundaries=None*, *orientation='vertical'*, *extend='neither'*, *spacing='uniform'*, *ticks=None*, *format=None*, *drawedges=False*, *filled=True*)

Draw a colorbar in an existing axes.

This is a base class for the *Colorbar* class, which is the basis for the *colorbar()* method and pyplot function.

It is also useful by itself for showing a colormap. If the *cmap* kwarg is given but *boundaries* and *values* are left as None, then the colormap will be displayed on a 0-1 scale. To show the under- and over-value colors, specify the *norm* as:

```
colors.Normalize(clip=False)
```

To show the colors versus index instead of on the 0-1 scale, use:

```
norm=colors.NoNorm.
```

Useful attributes:

ax
the Axes instance in which the colorbar is drawn

lines
a LineCollection if lines were drawn, otherwise None

dividers
a LineCollection if *drawedges* is True, otherwise None

Useful public methods are `set_label()` and *add_lines()*.

Parameters

norm

[*matplotlib.colors.Normalize* (or subclass thereof)] The normalizing object which scales data, typically into the interval `[0, 1]`. If *None*, *norm* defaults to a *colors.Normalize* object which initializes its scaling based on the first data processed.

cmap

[str or *Colormap*] The colormap used to map normalized data values to RGBA colors.

`add_lines`(*self*, *levels*, *colors*, *linewidths*)
Draw lines on the colorbar. It deletes preexisting lines.

`set_alpha`(*self*, *alpha*)
Set the alpha value for transparency.

`set_label_text`(*self*, *label*, ***kw*)
Label the long axis of the colorbar.

`update_artists`(*self*)
Update the colorbar associated artists, *filled* and *ends*. Note that *lines* are not updated. This needs to be called whenever clim of associated image changes.

mpl_toolkits.axes_grid1.colorbar.`colorbar`(*mappable*, *cax=None*, *ax=None*, ***kw*)

Create a colorbar for a ScalarMappable instance.

Documentation for the pyplot thin wrapper:

Add a colorbar to a plot.

Function signatures for the *pyplot* interface; all but the first are also method signatures for the *colorbar()* method:

```
colorbar(**kwargs)
colorbar(mappable, **kwargs)
colorbar(mappable, cax=cax, **kwargs)
colorbar(mappable, ax=ax, **kwargs)
```

arguments:

mappable

the `Image`, *ContourSet*, etc. to which the colorbar applies; this argument is mandatory for the *colorbar()* method but optional for the *colorbar()* function, which sets the default to the current image.

keyword arguments:

cax

None | axes object into which the colorbar will be drawn

ax

None | parent axes object from which space for a new colorbar axes will be stolen

Additional keyword arguments are of two kinds:

axes properties:

Property	Description
orientation	vertical or horizontal
fraction	0.15; fraction of original axes to use for colorbar
pad	0.05 if vertical, 0.15 if horizontal; fraction of original axes between colorbar and new image axes
shrink	1.0; fraction by which to shrink the colorbar
aspect	20; ratio of long to short dimensions

colorbar properties:

Property	Description
extend	['neither' | 'both' | 'min' | 'max'] If not 'neither', make pointed end(s) for out-of- range values. These are set for a given colormap using the colormap set_under and set_over methods.
spacing	['uniform' | 'proportional'] Uniform spacing gives each discrete color the same space; proportional makes the space proportional to the data interval.
ticks	[None | list of ticks | Locator object] If None, ticks are determined automatically from the input.
format	[None | format string | Formatter object] If None, the *ScalarFormatter* is used. If a format string is given, e.g., '%.3f', that is used. An alternative *Formatter* object may be given instead.
drawedges	bool Whether to draw lines at color boundaries.

The following will probably be useful only in the context of indexed colors (that is, when the mappable has norm=NoNorm()), or other unusual circumstances.

Property	Description
boundaries	None or a sequence
values	None or a sequence which must be of length 1 less than the sequence of *boundaries*. For each region delimited by adjacent entries in *boundaries*, the color mapped to the corresponding value in values will be used.

If *mappable* is a `ContourSet`, its *extend* kwarg is included automatically.

Note that the *shrink* kwarg provides a simple way to keep a vertical colorbar, for example, from being taller than the axes of the mappable to which the colorbar is attached; but it is a manual method requiring some trial and error. If the colorbar is too tall (or a horizontal colorbar is too wide) use a smaller value of *shrink*.

For more precise control, you can manually specify the positions of the axes objects in which the mappable and the colorbar are drawn. In this case, do not use any of the axes properties kwargs.

It is known that some vector graphics viewer (svg and pdf) renders white gaps between segments of the colorbar. This is due to bugs in the viewers not matplotlib. As a workaround the colorbar can be rendered with overlapping segments:

```
cbar = colorbar()
cbar.solids.set_edgecolor("face")
draw()
```

However this has negative consequences in other circumstances. Particularly with semi transparent images (alpha < 1) and colorbar extensions and is not enabled by default see (issue #1188).

returns:

Colorbar instance; see also its base class, *ColorbarBase*. Call the *set_label()* method to label the colorbar.

The transData of the *cax* is adjusted so that the limits in the longest axis actually corresponds to the limits in colorbar range. On the other hand, the shortest axis has a data limits of [1,2], whose unconventional value is to prevent underflow when log scale is used.

`mpl_toolkits.axes_grid1.colorbar.`**`make_axes`**(*parent*, *, *fraction=0.15*, *shrink=1.0*, *aspect=20*, ***kw*)

Resize and reposition a parent axes, and return a child axes suitable for a colorbar

```
cax, kw = make_axes(parent, **kw)
```

Keyword arguments may include the following (with defaults):

orientation

'vertical' or 'horizontal'

Property	Description
orientation	vertical or horizontal
fraction	0.15; fraction of original axes to use for colorbar
pad	0.05 if vertical, 0.15 if horizontal; fraction of original axes between colorbar and new image axes
shrink	1.0; fraction by which to shrink the colorbar
aspect	20; ratio of long to short dimensions

All but the first of these are stripped from the input kw set.

Returns (cax, kw), the child axes and the reduced kw dictionary.

mpl_toolkits.axes_grid1.inset_locator

A collection of functions and objects for creating or placing inset axes.

Classes

AnchoredLocatorBase(bbox_to_anchor, ...[, ...])	**Parameters**
AnchoredSizeLocator(bbox_to_anchor, x_size, ...)	**Parameters**
AnchoredZoomLocator(parent_axes, zoom, loc)	**Parameters**
BboxConnector(bbox1, bbox2, loc1[, loc2])	Connect two bboxes with a straight line.
BboxConnectorPatch(bbox1, bbox2, loc1a, ...)	Connect two bboxes with a quadrilateral.
BboxPatch(bbox, **kwargs)	Patch showing the shape bounded by a Bbox.
InsetPosition(parent, lbwh)	An object for positioning an inset axes.

mpl_toolkits.axes_grid1.inset_locator.AnchoredLocatorBase

class mpl_toolkits.axes_grid1.inset_locator.**AnchoredLocatorBase**(*bbox_to_anchor*, *offsetbox*, *loc*, *borderpad=0.5*, *bbox_transform=None*)

Bases: *matplotlib.offsetbox.AnchoredOffsetbox*

Parameters

loc

[str] The box location. Supported values:

- 'upper right'
- 'upper left'
- 'lower left'
- 'lower right'
- 'center left'
- 'center right'
- 'lower center'
- 'upper center'
- 'center'

For backward compatibility, numeric values are accepted as well. See the parameter *loc* of *Legend* for details.

pad

[float, default: 0.4] Padding around the child as fraction of the fontsize.

borderpad

[float, default: 0.5] Padding between the offsetbox frame and the *bbox_to_anchor*.

child

[*OffsetBox*] The box that will be anchored.

prop

[*FontProperties*] This is only used as a reference for paddings. If not given, `rcParams["legend.fontsize"]` (default: `'medium'`) is used.

frameon

[bool] Whether to draw a frame around the box.

bbox_to_anchor

[*BboxBase*, 2-tuple, or 4-tuple of floats] Box that is used to position the legend in conjunction with *loc*.

bbox_transform

[None or *matplotlib.transforms.Transform*] The transform for the bounding box (*bbox_to_anchor*).

****kwargs**

All other parameters are passed on to *OffsetBox*.

Notes

See *Legend* for a detailed description of the anchoring mechanism.

`__call__`(*self*, *ax*, *renderer*)
Call self as a function.

`__init__`(*self*, *bbox_to_anchor*, *offsetbox*, *loc*, *borderpad=0.5*, *bbox_transform=None*)

Parameters

loc

[str] The box location. Supported values:

- 'upper right'
- 'upper left'
- 'lower left'
- 'lower right'
- 'center left'
- 'center right'
- 'lower center'
- 'upper center'
- 'center'

For backward compatibility, numeric values are accepted as well. See the parameter *loc* of *Legend* for details.

pad

[float, default: 0.4] Padding around the child as fraction of the fontsize.

borderpad

[float, default: 0.5] Padding between the offsetbox frame and the *bbox_to_anchor*.

child

[*OffsetBox*] The box that will be anchored.

prop

[*FontProperties*] This is only used as a reference for paddings. If not given, `rcParams["legend.fontsize"]` (default: `'medium'`) is used.

frameon

[bool] Whether to draw a frame around the box.

bbox_to_anchor

[*BboxBase*, 2-tuple, or 4-tuple of floats] Box that is used to position the legend in conjunction with *loc*.

bbox_transform

[None or *matplotlib.transforms.Transform*] The transform for the bounding box (*bbox_to_anchor*).

****kwargs**

All other parameters are passed on to *OffsetBox*.

Notes

See *Legend* for a detailed description of the anchoring mechanism.

`__module__ = 'mpl_toolkits.axes_grid1.inset_locator'`

`draw`(*self*, *renderer*)
Update the location of children if necessary and draw them to the given *renderer*.

Examples using `mpl_toolkits.axes_grid1.inset_locator.AnchoredLocatorBase`

mpl_toolkits.axes_grid1.inset_locator.AnchoredSizeLocator

class mpl_toolkits.axes_grid1.inset_locator.**AnchoredSizeLocator**(*bbox_to_anchor*, *x_size*, *y_size*, *loc*, *borderpad=0.5*, *bbox_transform=None*)

Bases: *mpl_toolkits.axes_grid1.inset_locator.AnchoredLocatorBase*

Parameters

loc

[str] The box location. Supported values:

- 'upper right'
- 'upper left'
- 'lower left'
- 'lower right'
- 'center left'
- 'center right'
- 'lower center'
- 'upper center'
- 'center'

For backward compatibility, numeric values are accepted as well. See the parameter *loc* of *Legend* for details.

pad

[float, default: 0.4] Padding around the child as fraction of the fontsize.

borderpad

[float, default: 0.5] Padding between the offsetbox frame and the *bbox_to_anchor*.

child

[*OffsetBox*] The box that will be anchored.

prop

[*FontProperties*] This is only used as a reference for paddings. If not given, `rcParams["legend.fontsize"]` (default: `'medium'`) is used.

frameon

[bool] Whether to draw a frame around the box.

bbox_to_anchor

[*BboxBase*, 2-tuple, or 4-tuple of floats] Box that is used to position the legend in conjunction with *loc*.

bbox_transform

[None or *matplotlib.transforms.Transform*] The transform for the bounding box (*bbox_to_anchor*).

****kwargs**

All other parameters are passed on to *OffsetBox*.

Notes

See *Legend* for a detailed description of the anchoring mechanism.

`__init__`(*self*, *bbox_to_anchor*, *x_size*, *y_size*, *loc*, *borderpad=0.5*, *bbox_transform=None*)

Parameters

loc

[str] The box location. Supported values:

- 'upper right'
- 'upper left'
- 'lower left'
- 'lower right'
- 'center left'
- 'center right'
- 'lower center'
- 'upper center'
- 'center'

For backward compatibility, numeric values are accepted as well. See the parameter *loc* of *Legend* for details.

pad

[float, default: 0.4] Padding around the child as fraction of the fontsize.

borderpad

[float, default: 0.5] Padding between the offsetbox frame and the *bbox_to_anchor*.

child

[*OffsetBox*] The box that will be anchored.

prop

[*FontProperties*] This is only used as a reference for paddings. If not given, `rcParams["legend.fontsize"]` (default: `'medium'`) is used.

frameon

[bool] Whether to draw a frame around the box.

bbox_to_anchor

[*BboxBase*, 2-tuple, or 4-tuple of floats] Box that is used to position the legend in conjunction with *loc*.

bbox_transform

[None or *matplotlib.transforms.Transform*] The transform for the bounding box (*bbox_to_anchor*).

****kwargs**

All other parameters are passed on to *OffsetBox*.

Notes

See *Legend* for a detailed description of the anchoring mechanism.

`__module__ = 'mpl_toolkits.axes_grid1.inset_locator'`

`get_extent`(*self*, *renderer*)

Return the extent of the box as (width, height, x, y).

This is the extent of the child plus the padding.

Examples using `mpl_toolkits.axes_grid1.inset_locator.AnchoredSizeLocator`

mpl_toolkits.axes_grid1.inset_locator.AnchoredZoomLocator

class mpl_toolkits.axes_grid1.inset_locator.**AnchoredZoomLocator**(*parent_axes*, *zoom*, *loc*, *borderpad=0.5*, *bbox_to_anchor=None*, *bbox_transform=None*)

Bases: *mpl_toolkits.axes_grid1.inset_locator.AnchoredLocatorBase*

Parameters

loc

[str] The box location. Supported values:

- 'upper right'
- 'upper left'
- 'lower left'
- 'lower right'
- 'center left'
- 'center right'
- 'lower center'
- 'upper center'
- 'center'

For backward compatibility, numeric values are accepted as well. See the parameter *loc* of *Legend* for details.

pad

[float, default: 0.4] Padding around the child as fraction of the fontsize.

borderpad

[float, default: 0.5] Padding between the offsetbox frame and the *bbox_to_anchor*.

child

[*OffsetBox*] The box that will be anchored.

prop

[*FontProperties*] This is only used as a reference for paddings. If not given, `rcParams["legend.fontsize"]` (default: `'medium'`) is used.

frameon

[bool] Whether to draw a frame around the box.

bbox_to_anchor

[*BboxBase*, 2-tuple, or 4-tuple of floats] Box that is used to position the legend in conjunction with *loc*.

bbox_transform

[None or *matplotlib.transforms.Transform*] The transform for the bounding box (*bbox_to_anchor*).

****kwargs**

All other parameters are passed on to *OffsetBox*.

Notes

See *Legend* for a detailed description of the anchoring mechanism.

`__init__`(*self*, *parent_axes*, *zoom*, *loc*, *borderpad=0.5*, *bbox_to_anchor=None*, *bbox_transform=None*)

Parameters

loc

[str] The box location. Supported values:

- 'upper right'
- 'upper left'
- 'lower left'
- 'lower right'
- 'center left'
- 'center right'
- 'lower center'
- 'upper center'
- 'center'

For backward compatibility, numeric values are accepted as well. See the parameter *loc* of *Legend* for details.

pad

[float, default: 0.4] Padding around the child as fraction of the fontsize.

borderpad

[float, default: 0.5] Padding between the offsetbox frame and the *bbox_to_anchor*.

child

[*OffsetBox*] The box that will be anchored.

prop

[*FontProperties*] This is only used as a reference for paddings. If not given, `rcParams["legend.fontsize"]` (default: `'medium'`) is used.

frameon

[bool] Whether to draw a frame around the box.

bbox_to_anchor

[*BboxBase*, 2-tuple, or 4-tuple of floats] Box that is used to position the legend in conjunction with *loc*.

bbox_transform

[None or *matplotlib.transforms.Transform*] The transform for the bounding box (*bbox_to_anchor*).

****kwargs**

All other parameters are passed on to *OffsetBox*.

Notes

See *Legend* for a detailed description of the anchoring mechanism.

`__module__ = 'mpl_toolkits.axes_grid1.inset_locator'`

`get_extent`(*self*, *renderer*)

Return the extent of the box as (width, height, x, y).

This is the extent of the child plus the padding.

Examples using `mpl_toolkits.axes_grid1.inset_locator.AnchoredZoomLocator`

mpl_toolkits.axes_grid1.inset_locator.BboxConnector

`class mpl_toolkits.axes_grid1.inset_locator.BboxConnector`(*bbox1*, *bbox2*, *loc1*, *loc2=None*, ***kwargs*)

Bases: *matplotlib.patches.Patch*

Connect two bboxes with a straight line.

Parameters

bbox1, bbox2

[*matplotlib.transforms.Bbox*] Bounding boxes to connect.

loc1

[{1, 2, 3, 4}] Corner of *bbox1* to draw the line. Valid values are:

```
'upper right'  : 1,
'upper left'   : 2,
'lower left'   : 3,
'lower right'  : 4
```

loc2

[{1, 2, 3, 4}, optional] Corner of *bbox2* to draw the line. If None, defaults to *loc1*. Valid values are:

```
'upper right'  : 1,
'upper left'   : 2,
'lower left'   : 3,
'lower right'  : 4
```

**kwargs

Patch properties for the line drawn. Valid arguments include:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and retur
alpha	float or None
animated	bool
antialiased or aa	unknown
capstyle	{'butt', 'round', 'projecting'}
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color	color
contains	unknown
edgecolor or ec	color or None or 'auto'
facecolor or fc	color or None
figure	*Figure*
fill	bool
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}

Table 11 – continued from previous page

Property	Description
linewidth or lw	float or None
path_effects	*AbstractPathEffect*
picker	None or bool or callable
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
visible	bool
zorder	float

__init__(*self*, *bbox1*, *bbox2*, *loc1*, *loc2=None*, ***kwargs*)

Connect two bboxes with a straight line.

Parameters

bbox1, bbox2

[*matplotlib.transforms.Bbox*] Bounding boxes to connect.

loc1

[{1, 2, 3, 4}] Corner of *bbox1* to draw the line. Valid values are:

```
'upper right'  : 1,
'upper left'   : 2,
'lower left'   : 3,
'lower right'  : 4
```

loc2

[{1, 2, 3, 4}, optional] Corner of *bbox2* to draw the line. If None, defaults to *loc1*. Valid values are:

```
'upper right'  : 1,
'upper left'   : 2,
'lower left'   : 3,
'lower right'  : 4
```

****kwargs**

Patch properties for the line drawn. Valid arguments include:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and retur
alpha	float or None
animated	bool
antialiased or aa	unknown

Table 12 – continued from previous page

Property	Description
capstyle	{'butt', 'round', 'projecting'}
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color	color
contains	unknown
edgecolor or ec	color or None or 'auto'
facecolor or fc	color or None
figure	*Figure*
fill	bool
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float or None
path_effects	*AbstractPathEffect*
picker	None or bool or callable
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
visible	bool
zorder	float

`__module__ = 'mpl_toolkits.axes_grid1.inset_locator'`

`static connect_bbox`(*bbox1*, *bbox2*, *loc1*, *loc2=None*)

Helper function to obtain a Path from one bbox to another.

Parameters

bbox1, bbox2

[*matplotlib.transforms.Bbox*] Bounding boxes to connect.

loc1

[{1, 2, 3, 4}] Corner of *bbox1* to use. Valid values are:

```
'upper right'  : 1,
'upper left'   : 2,
'lower left'   : 3,
'lower right'  : 4
```

loc2

[{1, 2, 3, 4}, optional] Corner of *bbox2* to use. If None, defaults to *loc1*. Valid values are:

```
'upper right'  : 1,
'upper left'   : 2,
'lower left'   : 3,
'lower right'  : 4
```

Returns

path

[*matplotlib.path.Path*] A line segment from the *loc1* corner of *bbox1* to the *loc2* corner of *bbox2*.

`static get_bbox_edge_pos(`*bbox*, *loc*`)`

Helper function to obtain the location of a corner of a bbox

Parameters

bbox

[*matplotlib.transforms.Bbox*]

loc

[{1, 2, 3, 4}] Corner of *bbox*. Valid values are:

```
'upper right'  : 1,
'upper left'   : 2,
'lower left'   : 3,
'lower right'  : 4
```

Returns

x, y

[float] Coordinates of the corner specified by *loc*.

`get_path(`*self*`)`

Return the path of this patch.

Examples using `mpl_toolkits.axes_grid1.inset_locator.BboxConnector`

- sphx_glr_gallery_subplots_axes_and_figures_axes_zoom_effect.py

mpl_toolkits.axes_grid1.inset_locator.BboxConnectorPatch

class mpl_toolkits.axes_grid1.inset_locator.**BboxConnectorPatch**(*bbox1*, *bbox2*, *loc1a*, *loc2a*, *loc1b*, *loc2b*, ***kwargs*)

Bases: *mpl_toolkits.axes_grid1.inset_locator.BboxConnector*

Connect two bboxes with a quadrilateral.

The quadrilateral is specified by two lines that start and end at corners of the bboxes. The four sides of the quadrilateral are defined by the two lines given, the line between the two corners specified in *bbox1* and the line between the two corners specified in *bbox2*.

Parameters

bbox1, bbox2

[*matplotlib.transforms.Bbox*] Bounding boxes to connect.

loc1a, loc2a

[{1, 2, 3, 4}] Corners of *bbox1* and *bbox2* to draw the first line. Valid values are:

```
'upper right'  : 1,
'upper left'   : 2,
'lower left'   : 3,
'lower right'  : 4
```

loc1b, loc2b

[{1, 2, 3, 4}] Corners of *bbox1* and *bbox2* to draw the second line. Valid values are:

```
'upper right'  : 1,
'upper left'   : 2,
'lower left'   : 3,
'lower right'  : 4
```

****kwargs**

Patch properties for the line drawn:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and retur
alpha	float or None
animated	bool
antialiased or aa	unknown
capstyle	{'butt', 'round', 'projecting'}
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color	color
contains	unknown
edgecolor or ec	color or None or 'auto'
facecolor or fc	color or None
figure	*Figure*
fill	bool
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float or None
path_effects	*AbstractPathEffect*
picker	None or bool or callable
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
visible	bool
zorder	float

`__init__`(*self*, *bbox1*, *bbox2*, *loc1a*, *loc2a*, *loc1b*, *loc2b*, ***kwargs*)

Connect two bboxes with a quadrilateral.

The quadrilateral is specified by two lines that start and end at corners of the bboxes. The four sides of the quadrilateral are defined by the two lines given, the line between the two corners specified in *bbox1* and the line between the two corners specified in *bbox2*.

Parameters

bbox1, bbox2

[*matplotlib.transforms.Bbox*] Bounding boxes to connect.

loc1a, loc2a

[{1, 2, 3, 4}] Corners of *bbox1* and *bbox2* to draw the first line.

Valid values are:

```
'upper right'  : 1,
'upper left'   : 2,
'lower left'   : 3,
'lower right'  : 4
```

loc1b, loc2b

[{1, 2, 3, 4}] Corners of *bbox1* and *bbox2* to draw the second line. Valid values are:

```
'upper right'  : 1,
'upper left'   : 2,
'lower left'   : 3,
'lower right'  : 4
```

**kwargs

Patch properties for the line drawn:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and retur
alpha	float or None
animated	bool
antialiased or aa	unknown
capstyle	{'butt', 'round', 'projecting'}
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color	color
contains	unknown
edgecolor or ec	color or None or 'auto'
facecolor or fc	color or None
figure	*Figure*
fill	bool
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float or None
path_effects	*AbstractPathEffect*
picker	None or bool or callable
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None

Table 14 – continued from previous page

Property	Description
transform	*Transform*
url	str
visible	bool
zorder	float

```
__module__ = 'mpl_toolkits.axes_grid1.inset_locator'
```

get_path(*self*)
Return the path of this patch.

Examples using mpl_toolkits.axes_grid1.inset_locator.BboxConnectorPatch

- sphx_glr_gallery_subplots_axes_and_figures_axes_zoom_effect.py

mpl_toolkits.axes_grid1.inset_locator.BboxPatch

class mpl_toolkits.axes_grid1.inset_locator.BboxPatch(*bbox*, ***kwargs*)
Bases: *matplotlib.patches.Patch*

Patch showing the shape bounded by a Bbox.

Parameters

bbox

[*matplotlib.transforms.Bbox*] Bbox to use for the extents of this patch.

****kwargs**

Patch properties. Valid arguments include:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and retur
alpha	float or None
animated	bool
antialiased or aa	unknown
capstyle	{'butt', 'round', 'projecting'}
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color	color
contains	unknown
edgecolor or ec	color or None or 'auto'

Table 15 – continued from previous page

Property	Description
facecolor or fc	color or None
figure	*Figure*
fill	bool
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float or None
path_effects	*AbstractPathEffect*
picker	None or bool or callable
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
visible	bool
zorder	float

__init__(*self*, *bbox*, ***kwargs*)

Patch showing the shape bounded by a Bbox.

Parameters

bbox

[*matplotlib.transforms.Bbox*] Bbox to use for the extents of this patch.

****kwargs**

Patch properties. Valid arguments include:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and retur
alpha	float or None
animated	bool
antialiased or aa	unknown
capstyle	{'butt', 'round', 'projecting'}
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color	color
contains	unknown

Table 16 – continued from previous page

Property	Description
edgecolor or ec	color or None or 'auto'
facecolor or fc	color or None
figure	*Figure*
fill	bool
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float or None
path_effects	*AbstractPathEffect*
picker	None or bool or callable
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
visible	bool
zorder	float

```
__module__ = 'mpl_toolkits.axes_grid1.inset_locator'
```

get_path(*self*)
Return the path of this patch.

Examples using mpl_toolkits.axes_grid1.inset_locator.BboxPatch

- sphx_glr_gallery_subplots_axes_and_figures_axes_zoom_effect.py

mpl_toolkits.axes_grid1.inset_locator.InsetPosition

class mpl_toolkits.axes_grid1.inset_locator.**InsetPosition**(*parent*, *lbwh*)
Bases: object

An object for positioning an inset axes.

This is created by specifying the normalized coordinates in the axes, instead of the figure.

Parameters

parent

[*matplotlib.axes.Axes*] Axes to use for normalizing coordinates.

lbwh

[iterable of four floats] The left edge, bottom edge, width, and height of the inset axes, in units of the normalized coordinate of the *parent* axes.

See also:

matplotlib.axes.Axes.set_axes_locator()

Examples

The following bounds the inset axes to a box with 20% of the parent axes's height and 40% of the width. The size of the axes specified ([0, 0, 1, 1]) ensures that the axes completely fills the bounding box:

```
>>> parent_axes = plt.gca()
>>> ax_ins = plt.axes([0, 0, 1, 1])
>>> ip = InsetPosition(ax, [0.5, 0.1, 0.4, 0.2])
>>> ax_ins.set_axes_locator(ip)
```

`__call__`(*self*, *ax*, *renderer*)
Call self as a function.

`__dict__ = mappingproxy({'__module__': 'mpl_toolkits.axes_grid1.inset_locator', '__init__`

`__init__`(*self*, *parent*, *lbwh*)
An object for positioning an inset axes.

This is created by specifying the normalized coordinates in the axes, instead of the figure.

Parameters

parent

[*matplotlib.axes.Axes*] Axes to use for normalizing coordinates.

lbwh

[iterable of four floats] The left edge, bottom edge, width, and height of the inset axes, in units of the normalized coordinate of the *parent* axes.

See also:

matplotlib.axes.Axes.set_axes_locator()

Examples

The following bounds the inset axes to a box with 20% of the parent axes's height and 40% of the width. The size of the axes specified ([0, 0, 1, 1]) ensures that the axes completely fills the bounding box:

```
>>> parent_axes = plt.gca()
>>> ax_ins = plt.axes([0, 0, 1, 1])
>>> ip = InsetPosition(ax, [0.5, 0.1, 0.4, 0.2])
>>> ax_ins.set_axes_locator(ip)
```

__module__ = 'mpl_toolkits.axes_grid1.inset_locator'

__weakref__
list of weak references to the object (if defined)

Examples using `mpl_toolkits.axes_grid1.inset_locator.InsetPosition`

Functions

inset_axes(parent_axes, width, height[, ...])	Create an inset axes with a given width and height.
mark_inset(parent_axes, inset_axes, loc1, ...)	Draw a box to mark the location of an area represented by an inset axes.
zoomed_inset_axes(parent_axes, zoom[, loc, ...])	Create an anchored inset axes by scaling a parent axes.

mpl_toolkits.axes_grid1.inset_locator.inset_axes

mpl_toolkits.axes_grid1.inset_locator.**inset_axes**(*parent_axes*, *width*, *height*, *loc='upper right'*, *bbox_to_anchor=None*, *bbox_transform=None*, *axes_class=None*, *axes_kwargs=None*, *borderpad=0.5*)

Create an inset axes with a given width and height.

Both sizes used can be specified either in inches or percentage. For example,:

```
inset_axes(parent_axes, width='40%', height='30%', loc=3)
```

creates in inset axes in the lower left corner of *parent_axes* which spans over 30% in height and 40% in width of the *parent_axes*. Since the usage of *inset_axes* may become slightly tricky when exceeding such standard cases, it is recommended to read the examples.

Parameters

parent_axes

[*matplotlib.axes.Axes*] Axes to place the inset axes.

width, height

[float or str] Size of the inset axes to create. If a float is provided, it is the size in inches, e.g. *width=1.3*. If a string is provided, it is the size in relative units, e.g. *width='40%'*. By default, i.e. if neither *bbox_to_anchor* nor *bbox_transform* are specified, those are relative to the parent_axes. Otherwise they are to be understood relative to the bounding box provided via *bbox_to_anchor*.

loc

[int or str, default: 1] Location to place the inset axes. The valid locations are:

```
'upper right'  : 1,
'upper left'   : 2,
'lower left'   : 3,
'lower right'  : 4,
'right'        : 5,
'center left'  : 6,
'center right' : 7,
'lower center' : 8,
'upper center' : 9,
'center'       : 10
```

bbox_to_anchor

[tuple or *matplotlib.transforms.BboxBase*, optional] Bbox that the inset axes will be anchored to. If None, a tuple of (0, 0, 1, 1) is used if *bbox_transform* is set to *parent_axes.transAxes* or *parent_axes.figure.transFigure*. Otherwise, *parent_axes.bbox* is used. If a tuple, can be either [left, bottom, width, height], or [left, bottom]. If the kwargs *width* and/or *height* are specified in relative units, the 2-tuple [left, bottom] cannot be used. Note that, unless *bbox_transform* is set, the units of the bounding box are interpreted in the pixel coordinate. When using *bbox_to_anchor* with tuple, it almost always makes sense to also specify a *bbox_transform*. This might often be the axes transform *parent_axes.transAxes*.

bbox_transform

[*matplotlib.transforms.Transform*, optional] Transformation for the bbox that contains the inset axes. If None, a *transforms.IdentityTransform* is used. The value of *bbox_to_anchor* (or the return value of its get_points method) is transformed by the *bbox_transform* and then interpreted as points in the

pixel coordinate (which is dpi dependent). You may provide *bbox_to_anchor* in some normalized coordinate, and give an appropriate transform (e.g., *parent_axes.transAxes*).

axes_class

[*matplotlib.axes.Axes* type, optional] If specified, the inset axes created will be created with this class's constructor.

axes_kwargs

[dict, optional] Keyworded arguments to pass to the constructor of the inset axes. Valid arguments include:

Property	Description
adjustable	{'box', 'datalim'}
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and r
alpha	float or None
anchor	2-tuple of floats or {'C', 'SW', 'S', 'SE', ...}
animated	bool
aspect	{'auto'} or num
autoscale_on	bool
autoscalex_on	bool
autoscaley_on	bool
axes_locator	Callable[[Axes, Renderer], Bbox]
axisbelow	bool or 'line'
box_aspect	None, or a number
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
contains	unknown
facecolor or fc	color
figure	*Figure*
frame_on	bool
gid	str
in_layout	bool
label	object
navigate	bool
navigate_mode	unknown
path_effects	*AbstractPathEffect*
picker	None or bool or callable
position	[left, bottom, width, height] or *Bbox*
prop_cycle	unknown
rasterization_zorder	float or None
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None

Table 18 – continued from previous page

Property	Description
title	str
transform	*Transform*
url	str
visible	bool
xbound	unknown
xlabel	str
xlim	(bottom: float, top: float)
xmargin	float greater than -0.5
xscale	{"linear", "log", "symlog", "logit", ...}
xticklabels	unknown
xticks	unknown
ybound	unknown
ylabel	str
ylim	(bottom: float, top: float)
ymargin	float greater than -0.5
yscale	{"linear", "log", "symlog", "logit", ...}
yticklabels	unknown
yticks	unknown
zorder	float

borderpad

[float, default: 0.5] Padding between inset axes and the bbox_to_anchor. The units are axes font size, i.e. for a default font size of 10 points *borderpad = 0.5* is equivalent to a padding of 5 points.

Returns

inset_axes

[*axes_class*] Inset axes object created.

Notes

The meaning of *bbox_to_anchor* and *bbox_to_transform* is interpreted differently from that of legend. The value of bbox_to_anchor (or the return value of its get_points method; the default is *parent_axes.bbox*) is transformed by the bbox_transform (the default is Identity transform) and then interpreted as points in the pixel coordinate (which is dpi dependent).

Thus, following three calls are identical and creates an inset axes with respect to the *parent_axes*:

```
axins = inset_axes(parent_axes, "30%", "40%")
axins = inset_axes(parent_axes, "30%", "40%",
                   bbox_to_anchor=parent_axes.bbox)
axins = inset_axes(parent_axes, "30%", "40%",
                   bbox_to_anchor=(0, 0, 1, 1),
                   bbox_transform=parent_axes.transAxes)
```

Examples using `mpl_toolkits.axes_grid1.inset_locator.inset_axes`

- sphx_glr_gallery_axes_grid1_demo_colorbar_of_inset_axes.py
- sphx_glr_gallery_axes_grid1_demo_colorbar_with_inset_locator.py
- sphx_glr_gallery_axes_grid1_inset_locator_demo.py
- *Overview of axes_grid1 toolkit*

mpl_toolkits.axes_grid1.inset_locator.mark_inset

mpl_toolkits.axes_grid1.inset_locator.**mark_inset**(*parent_axes*, *inset_axes*, *loc1*, *loc2*, ***kwargs*)

Draw a box to mark the location of an area represented by an inset axes.

This function draws a box in *parent_axes* at the bounding box of *inset_axes*, and shows a connection with the inset axes by drawing lines at the corners, giving a "zoomed in" effect.

Parameters

parent_axes

[*matplotlib.axes.Axes*] Axes which contains the area of the inset axes.

inset_axes

[*matplotlib.axes.Axes*] The inset axes.

loc1, loc2

[{1, 2, 3, 4}] Corners to use for connecting the inset axes and the area in the parent axes.

****kwargs**

Patch properties for the lines and box drawn:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and retur
alpha	float or None

Table 19 – continued from previous page

Property	Description
animated	bool
antialiased or aa	unknown
capstyle	{'butt', 'round', 'projecting'}
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color	color
contains	unknown
edgecolor or ec	color or None or 'auto'
facecolor or fc	color or None
figure	*Figure*
fill	bool
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float or None
path_effects	*AbstractPathEffect*
picker	None or bool or callable
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
visible	bool
zorder	float

Returns

pp

[*matplotlib.patches.Patch*] The patch drawn to represent the area of the inset axes.

p1, p2

[*matplotlib.patches.Patch*] The patches connecting two corners of the inset axes and its area.

Examples using `mpl_toolkits.axes_grid1.inset_locator.mark_inset`

- sphx_glr_gallery_axes_grid1_inset_locator_demo2.py

mpl_toolkits.axes_grid1.inset_locator.zoomed_inset_axes

mpl_toolkits.axes_grid1.inset_locator.**zoomed_inset_axes**(*parent_axes*, *zoom*, *loc='upper right'*, *bbox_to_anchor=None*, *bbox_transform=None*, *axes_class=None*, *axes_kwargs=None*, *borderpad=0.5*)

Create an anchored inset axes by scaling a parent axes. For usage, also see the examples.

Parameters

parent_axes

[*matplotlib.axes.Axes*] Axes to place the inset axes.

zoom

[float] Scaling factor of the data axes. *zoom* > 1 will enlargen the coordinates (i.e., "zoomed in"), while *zoom* < 1 will shrink the coordinates (i.e., "zoomed out").

loc

[int or str, default: 'upper right'] Location to place the inset axes. The valid locations are:

```
'upper right'  : 1,
'upper left'   : 2,
'lower left'   : 3,
'lower right'  : 4,
'right'        : 5,
'center left'  : 6,
'center right' : 7,
'lower center' : 8,
'upper center' : 9,
'center'       : 10
```

bbox_to_anchor

[tuple or *matplotlib.transforms.BboxBase*, optional] Bbox that the inset axes will be anchored to. If None, *parent_axes.bbox* is used. If a tuple, can be either [left, bottom, width, height], or [left, bottom]. If the kwargs *width* and/or *height* are specified in relative units, the 2-tuple [left, bottom] cannot be used. Note that

the units of the bounding box are determined through the transform in use. When using *bbox_to_anchor* it almost always makes sense to also specify a *bbox_transform*. This might often be the axes transform *parent_axes.transAxes*.

bbox_transform

[*matplotlib.transforms.Transform*, optional] Transformation for the bbox that contains the inset axes. If None, a *transforms.IdentityTransform* is used (i.e. pixel coordinates). This is useful when not providing any argument to *bbox_to_anchor*. When using *bbox_to_anchor* it almost always makes sense to also specify a *bbox_transform*. This might often be the axes transform *parent_axes.transAxes*. Inversely, when specifying the axes- or figure-transform here, be aware that not specifying *bbox_to_anchor* will use *parent_axes.bbox*, the units of which are in display (pixel) coordinates.

axes_class

[*matplotlib.axes.Axes* type, optional] If specified, the inset axes created will be created with this class's constructor.

axes_kwargs

[dict, optional] Keyworded arguments to pass to the constructor of the inset axes. Valid arguments include:

Property	Description
adjustable	{'box', 'datalim'}
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and r
alpha	float or None
anchor	2-tuple of floats or {'C', 'SW', 'S', 'SE', ...}
animated	bool
aspect	{'auto'} or num
autoscale_on	bool
autoscalex_on	bool
autoscaley_on	bool
axes_locator	Callable[[Axes, Renderer], Bbox]
axisbelow	bool or 'line'
box_aspect	None, or a number
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
contains	unknown
facecolor or fc	color
figure	*Figure*
frame_on	bool
gid	str

Table 20 – continued from previous page

Property	Description
in_layout	bool
label	object
navigate	bool
navigate_mode	unknown
path_effects	*AbstractPathEffect*
picker	None or bool or callable
position	[left, bottom, width, height] or *Bbox*
prop_cycle	unknown
rasterization_zorder	float or None
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
title	str
transform	*Transform*
url	str
visible	bool
xbound	unknown
xlabel	str
xlim	(bottom: float, top: float)
xmargin	float greater than -0.5
xscale	{"linear", "log", "symlog", "logit", ...}
xticklabels	unknown
xticks	unknown
ybound	unknown
ylabel	str
ylim	(bottom: float, top: float)
ymargin	float greater than -0.5
yscale	{"linear", "log", "symlog", "logit", ...}
yticklabels	unknown
yticks	unknown
zorder	float

borderpad

[float, default: 0.5] Padding between inset axes and the bbox_to_anchor. The units are axes font size, i.e. for a default font size of 10 points *borderpad = 0.5* is equivalent to a padding of 5 points.

Returns

inset_axes

[*axes_class*] Inset axes object created.

Examples using `mpl_toolkits.axes_grid1.inset_locator.zoomed_inset_axes`

- sphx_glr_gallery_axes_grid1_demo_colorbar_of_inset_axes.py
- sphx_glr_gallery_axes_grid1_inset_locator_demo2.py

mpl_toolkits.axes_grid1.mpl_axes

Classes

Axes(fig, rect[, facecolor, frameon, ...])	Build an axes in a figure.
SimpleAxisArtist(axis, axisnum, spine)	
SimpleChainedObjects(objects)	

mpl_toolkits.axes_grid1.mpl_axes.Axes

class mpl_toolkits.axes_grid1.mpl_axes.**Axes**(*fig*, *rect*, *facecolor=None*, *frameon=True*, *sharex=None*, *sharey=None*, *label=''*, *xscale=None*, *yscale=None*, *box_aspect=None*, **kwargs*)

Bases: `matplotlib.axes._axes.Axes`

Build an axes in a figure.

Parameters

fig

[*Figure*] The axes is build in the *Figure* *fig*.

rect

[[left, bottom, width, height]] The axes is build in the rectangle *rect*. *rect* is in *Figure* coordinates.

sharex, sharey

[*Axes*, optional] The x or y *axis* is shared with the x or y axis in the input *Axes*.

frameon

[bool, default: True] Whether the axes frame is visible.

box_aspect

[None, or a number, optional] Sets the aspect of the axes box. See *set_box_aspect* for details.

****kwargs**

Other optional keyword arguments:

Property	Description
adjustable	{'box', 'datalim'}
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and r
alpha	float or None
anchor	2-tuple of floats or {'C', 'SW', 'S', 'SE', ...}
animated	bool
aspect	{'auto'} or num
autoscale_on	bool
autoscalex_on	bool
autoscaley_on	bool
axes_locator	Callable[[Axes, Renderer], Bbox]
axisbelow	bool or 'line'
box_aspect	None, or a number
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
contains	unknown
facecolor or fc	color
figure	*Figure*
frame_on	bool
gid	str
in_layout	bool
label	object
navigate	bool
navigate_mode	unknown
path_effects	*AbstractPathEffect*
picker	None or bool or callable
position	[left, bottom, width, height] or *Bbox*
prop_cycle	unknown
rasterization_zorder	float or None
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
title	str
transform	*Transform*
url	str
visible	bool
xbound	unknown
xlabel	str
xlim	(bottom: float, top: float)
xmargin	float greater than -0.5
xscale	{"linear", "log", "symlog", "logit", ...}

Table 22 – continued from previous page

Property	Description
xticklabels	unknown
xticks	unknown
ybound	unknown
ylabel	str
ylim	(bottom: float, top: float)
ymargin	float greater than -0.5
yscale	{"linear", "log", "symlog", "logit", ...}
yticklabels	unknown
yticks	unknown
zorder	float

Returns

Axes

The new *Axes* object.

`class AxisDict`*() -> new empty dictionary dict(mapping) -> new dictionary initialized from a mapping object's (key, value) pairs dict(iterable) -> new dictionary initialized as if via: d = {} for k, v in iterable: d[k] = v dict(**kwargs) -> new dictionary initialized with the name=value pairs in the keyword argument list. For example: dict(one=1, two=2)*

Bases: `dict`

`__call__`(*self*, **v*, ***kwargs*)
Call self as a function.

`__dict__ = mappingproxy({'__module__': 'mpl_toolkits.axes_grid1.mpl_axes', '__init__'`

`__getitem__`(*self*, *k*)
x.__getitem__(y) <==> x[y]

`__init__`(*self*, *axes*)
Initialize self. See help(type(self)) for accurate signature.

`__module__ = 'mpl_toolkits.axes_grid1.mpl_axes'`

`__weakref__`
list of weak references to the object (if defined)

`__module__ = 'mpl_toolkits.axes_grid1.mpl_axes'`

`property axis`
Convenience method to get or set some axis properties.

Call signatures:

```
xmin, xmax, ymin, ymax = axis()
xmin, xmax, ymin, ymax = axis([xmin, xmax, ymin, ymax])
```

(continues on next page)

(continued from previous page)

```
xmin, xmax, ymin, ymax = axis(option)
xmin, xmax, ymin, ymax = axis(**kwargs)
```

Parameters

xmin, xmax, ymin, ymax

[float, optional] The axis limits to be set. This can also be achieved using

```
ax.set(xlim=(xmin, xmax), ylim=(ymin, ymax))
```

option

[bool or str] If a bool, turns axis lines and labels on or off. If a string, possible values are:

Value	Description
'on'	Turn on axis lines and labels. Same as `True`.
'off'	Turn off axis lines and labels. Same as `False`.
'equal'	Set equal scaling (i.e., make circles circular) by changing axis limits. This is the same as `ax.set_aspect('equal', adjustable='datalim')`. Explicit data limits may not be respected in this case.
'scaled'	Set equal scaling (i.e., make circles circular) by changing dimensions of the plot box. This is the same as `ax.set_aspect('equal', adjustable='box', anchor='C')`. Additionally, further autoscaling will be disabled.
'tight'	Set limits just large enough to show all data, then disable further autoscaling.
'auto'	Automatic scaling (fill plot box with data).
'image'	'scaled' with axis limits equal to data limits.
'square'	Square plot; similar to 'scaled', but initially forcing `xmax-xmin == ymax-ymin`.

emit

[bool, default: True] Whether observers are notified of the axis limit change. This option is passed on to *set_xlim* and *set_ylim*.

Returns

xmin, xmax, ymin, ymax

[float] The axis limits.

See also:

matplotlib.axes.Axes.set_xlim

matplotlib.axes.Axes.set_ylim

cla(*self*)
Clear the current axes.

Examples using mpl_toolkits.axes_grid1.mpl_axes.Axes

- sphx_glr_gallery_axes_grid1_demo_axes_divider.py
- sphx_glr_gallery_axes_grid1_demo_axes_grid2.py
- sphx_glr_gallery_axes_grid1_parasite_simple2.py
- sphx_glr_gallery_axes_grid1_simple_axesgrid.py
- sphx_glr_gallery_axes_grid1_simple_axesgrid2.py
- *Tight Layout guide*

mpl_toolkits.axes_grid1.mpl_axes.SimpleAxisArtist

class mpl_toolkits.axes_grid1.mpl_axes.**SimpleAxisArtist**(*axis*, *axisnum*, *spine*)
Bases: *matplotlib.artist.Artist*

__init__(*self*, *axis*, *axisnum*, *spine*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'mpl_toolkits.axes_grid1.mpl_axes'

property label

property major_ticklabels

property major_ticks

set_label(*self*, *txt*)
Set a label that will be displayed in the legend.

Parameters

s

[object] *s* will be converted to a string by calling str.

set_visible(*self*, *b*)
Set the artist's visibility.

Parameters

b

[bool]

`toggle`(*self*, *all=None*, *ticks=None*, *ticklabels=None*, *label=None*)

Examples using `mpl_toolkits.axes_grid1.mpl_axes.SimpleAxisArtist`

mpl_toolkits.axes_grid1.mpl_axes.SimpleChainedObjects

class mpl_toolkits.axes_grid1.mpl_axes.**SimpleChainedObjects**(*objects*)

Bases: `object`

`__call__`(*self*, **args*, ***kwargs*)

Call self as a function.

`__dict__ = mappingproxy({'__module__': 'mpl_toolkits.axes_grid1.mpl_axes', '__init__': <f`

`__getattr__`(*self*, *k*)

`__init__`(*self*, *objects*)

Initialize self. See help(type(self)) for accurate signature.

`__module__ = 'mpl_toolkits.axes_grid1.mpl_axes'`

`__weakref__`

list of weak references to the object (if defined)

Examples using `mpl_toolkits.axes_grid1.mpl_axes.SimpleChainedObjects`

mpl_toolkits.axes_grid1.parasite_axes

Classes

HostAxes	alias of `mpl_toolkits.axes_grid1.parasite_axes.AxesHostAxes`
HostAxesBase(*args, **kwargs)	
ParasiteAxes	alias of `mpl_toolkits.axes_grid1.parasite_axes.AxesParasite`
ParasiteAxesAuxTrans	alias of `mpl_toolkits.axes_grid1.parasite_axes.AxesParasiteParasiteAuxTrans`
ParasiteAxesAuxTransBase(parent_axes, ...[, ...])	
ParasiteAxesBase(parent_axes, **kwargs)	

mpl_toolkits.axes_grid1.parasite_axes.HostAxes

mpl_toolkits.axes_grid1.parasite_axes.**HostAxes**
alias of mpl_toolkits.axes_grid1.parasite_axes.AxesHostAxes

mpl_toolkits.axes_grid1.parasite_axes.HostAxesBase

class mpl_toolkits.axes_grid1.parasite_axes.**HostAxesBase**(**args*, ***kwargs*)
Bases: object

__dict__ = mappingproxy({'__module__': 'mpl_toolkits.axes_grid1.parasite_axes', '__init__

__init__(*self*, **args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'mpl_toolkits.axes_grid1.parasite_axes'

__weakref__
list of weak references to the object (if defined)

cla(*self*)

draw(*self*, *renderer*)

get_aux_axes(*self*, *tr*, *viewlim_mode='equal'*, *axes_class=<class 'mpl_toolkits.axes_grid1.parasite_axes.AxesParasite'>*)

get_tightbbox(*self*, *renderer*, *call_axes_locator=True*, *bbox_extra_artists=None*)

pick(*self*, *mouseevent*)

twin(*self*, *aux_trans=None*, *axes_class=None*)
Create a twin of Axes with no shared axis.

While self will have ticks on the left and bottom axis, the returned axes will have ticks on the top and right axis.

twinx(*self*, *axes_class=None*)
Create a twin of Axes with a shared x-axis but independent y-axis.

The y-axis of self will have ticks on the left and the returned axes will have ticks on the right.

twiny(*self*, *axes_class=None*)
Create a twin of Axes with a shared y-axis but independent x-axis.

The x-axis of self will have ticks on the bottom and the returned axes will have ticks on the top.

Examples using `mpl_toolkits.axes_grid1.parasite_axes.HostAxesBase`

- sphx_glr_gallery_axes_grid1_parasite_simple2.py
- sphx_glr_gallery_axisartist_demo_curvelinear_grid.py
- sphx_glr_gallery_axisartist_demo_floating_axes.py
- sphx_glr_gallery_axisartist_demo_floating_axis.py
- sphx_glr_gallery_axisartist_demo_parasite_axes.py

mpl_toolkits.axes_grid1.parasite_axes.ParasiteAxes

mpl_toolkits.axes_grid1.parasite_axes.**ParasiteAxes**
alias of mpl_toolkits.axes_grid1.parasite_axes.AxesParasite

mpl_toolkits.axes_grid1.parasite_axes.ParasiteAxesAuxTrans

mpl_toolkits.axes_grid1.parasite_axes.**ParasiteAxesAuxTrans**
alias of mpl_toolkits.axes_grid1.parasite_axes.AxesParasiteParasiteAuxTrans

mpl_toolkits.axes_grid1.parasite_axes.ParasiteAxesAuxTransBase

class mpl_toolkits.axes_grid1.parasite_axes.**ParasiteAxesAuxTransBase**(*parent_axes*, *aux_transform*, *viewlim_mode=None*, ***kwargs*)

Bases: object

__dict__ = mappingproxy({'__module__': 'mpl_toolkits.axes_grid1.parasite_axes', '__init__

__init__(*self*, *parent_axes*, *aux_transform*, *viewlim_mode=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'mpl_toolkits.axes_grid1.parasite_axes'

__weakref__
list of weak references to the object (if defined)

apply_aspect(*self*, *position=None*)

contour(*self*, **XYCL*, ***kwargs*)

contourf(*self*, **XYCL*, ***kwargs*)

get_viewlim_mode(*self*)

pcolor(*self*, **XYC*, ***kwargs*)

pcolormesh(*self*, **XYC*, ***kwargs*)

set_viewlim_mode(*self*, *mode*)

update_viewlim(*self*)

Examples using mpl_toolkits.axes_grid1.parasite_axes.ParasiteAxesAuxTransBase

- sphx_glr_gallery_axisartist_demo_curvelinear_grid.py

mpl_toolkits.axes_grid1.parasite_axes.ParasiteAxesBase

class mpl_toolkits.axes_grid1.parasite_axes.ParasiteAxesBase(*parent_axes*, ***kwargs*)

Bases: object

__dict__ = mappingproxy({'__module__': 'mpl_toolkits.axes_grid1.parasite_axes', 'get_imag

__init__(*self*, *parent_axes*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

__module__ = 'mpl_toolkits.axes_grid1.parasite_axes'

__weakref__

list of weak references to the object (if defined)

cla(*self*)

get_images_artists(*self*)

pick(*self*, *mouseevent*)

Examples using mpl_toolkits.axes_grid1.parasite_axes.ParasiteAxesBase

- sphx_glr_gallery_axisartist_demo_curvelinear_grid.py
- sphx_glr_gallery_axisartist_demo_parasite_axes.py

Functions

host_axes(*args[, axes_class, figure])	Create axes that can act as a hosts to parasitic axes.
host_axes_class_factory([axes_class])	
host_subplot(*args[, axes_class, figure])	Create a subplot that can act as a host to parasitic axes.
host_subplot_class_factory(axes_class)	
parasite_axes_auxtrans_class_factory([...])	
parasite_axes_class_factory([axes_class])	

mpl_toolkits.axes_grid1.parasite_axes.host_axes

mpl_toolkits.axes_grid1.parasite_axes.**host_axes**(**args*, *axes_class=<class 'mpl_toolkits.axes_grid1.mpl_axes.Axes'>*, *figure=None*, ***kwargs*)

Create axes that can act as a hosts to parasitic axes.

Parameters

figure

[*matplotlib.figure.Figure*] Figure to which the axes will be added. Defaults to the current figure *pyplot.gcf()*.

***args, **kwargs**

Will be passed on to the underlying `Axes` object creation.

Examples using mpl_toolkits.axes_grid1.parasite_axes.host_axes

mpl_toolkits.axes_grid1.parasite_axes.host_axes_class_factory

mpl_toolkits.axes_grid1.parasite_axes.**host_axes_class_factory**(*axes_class=None*)

Examples using mpl_toolkits.axes_grid1.parasite_axes.host_axes_class_factory

mpl_toolkits.axes_grid1.parasite_axes.host_subplot

mpl_toolkits.axes_grid1.parasite_axes.**host_subplot**(**args*, *axes_class=<class 'mpl_toolkits.axes_grid1.mpl_axes.Axes'>*, *figure=None*, ***kwargs*)

Create a subplot that can act as a host to parasitic axes.

Parameters

figure

[*matplotlib.figure.Figure*] Figure to which the subplot will be added. Defaults to the current figure *pyplot.gcf()*.

***args, **kwargs**

Will be passed on to the underlying `Axes` object creation.

Examples using `mpl_toolkits.axes_grid1.parasite_axes.host_subplot`

mpl_toolkits.axes_grid1.parasite_axes.host_subplot_class_factory

`mpl_toolkits.axes_grid1.parasite_axes.`**`host_subplot_class_factory`**(*axes_class*)

Examples using `mpl_toolkits.axes_grid1.parasite_axes.host_subplot_class_factory`

mpl_toolkits.axes_grid1.parasite_axes.parasite_axes_auxtrans_class_factory

`mpl_toolkits.axes_grid1.parasite_axes.`**`parasite_axes_auxtrans_class_factory`**(*axes_class=None*)

Examples using `mpl_toolkits.axes_grid1.parasite_axes.parasite_axes_auxtrans_class_factory`

mpl_toolkits.axes_grid1.parasite_axes.parasite_axes_class_factory

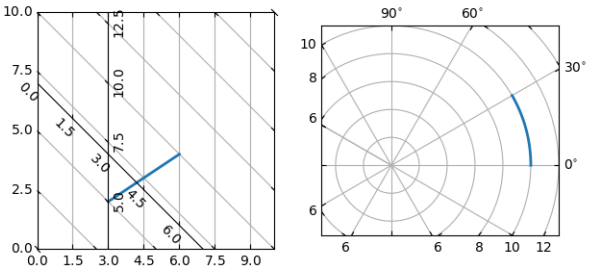
`mpl_toolkits.axes_grid1.parasite_axes.`**`parasite_axes_class_factory`**(*axes_class=None*)

Examples using `mpl_toolkits.axes_grid1.parasite_axes.parasite_axes_class_factory`

19.1.3 Matplotlib axisartist Toolkit

The *axisartist* namespace includes a derived Axes implementation (`mpl_toolkits.axisartist.Axes`). The biggest difference is that the artists that are responsible for drawing axis lines, ticks, ticklabels, and axis labels are separated out from the mpl's Axis class. This change was strongly motivated to support curvilinear grid.

You can find a tutorial describing usage of axisartist at the *axisartist* user guide.



The submodules of the axisartist API are:

axisartist.angle_helper	
axisartist.axes_divider	
axisartist.axes_grid	
axisartist.axes_rgb	
axisartist.axis_artist	axis_artist.py module provides axis-related artists.
axisartist.axisline_style	
axisartist.axislines	Axislines includes modified implementation of the Axes class.
axisartist.clip_path	
axisartist.floating_axes	An experimental support for curvilinear grid.
axisartist.grid_finder	
axisartist.grid_helper_curvelinear	An experimental support for curvilinear grid.
axisartist.parasite_axes	

mpl_toolkits.axisartist.angle_helper

Classes

ExtremeFinderCycle(nx, ny[, lon_cycle, ...])	This subclass handles the case where one or both coordinates should be taken modulo 360, or be restricted to not exceed a specific range.
FormatterDMS()	
FormatterHMS()	
LocatorBase(**kwargs)	
LocatorD(**kwargs)	
LocatorDM(**kwargs)	
LocatorDMS(**kwargs)	
LocatorH(**kwargs)	
LocatorHM(**kwargs)	
LocatorHMS(**kwargs)	

mpl_toolkits.axisartist.angle_helper.ExtremeFinderCycle

class mpl_toolkits.axisartist.angle_helper.**ExtremeFinderCycle**(*nx*, *ny*, *lon_cycle=360.0*, *lat_cycle=None*, *lon_minmax=None*, *lat_minmax=-90, 90*)

Bases: *mpl_toolkits.axisartist.grid_finder.ExtremeFinderSimple*

This subclass handles the case where one or both coordinates should be taken modulo 360, or be restricted to not exceed a specific range.

Parameters

nx, ny

[int] The number of samples in each direction.

lon_cycle, lat_cycle

[360 or None] If not None, values in the corresponding direction are taken modulo *lon_cycle* or *lat_cycle*; in theory this can be any number but the implementation actually assumes that it is 360 (if not None); other values give nonsensical results.

This is done by "unwrapping" the transformed grid coordinates so that jumps are less than a half-cycle; then normalizing the span to no more than a full cycle.

For example, if values are in the union of the [0, 2] and [358, 360] intervals (typically, angles measured modulo 360), the values in the second interval are normalized to [-2, 0] instead so that the values now cover [-2, 2]. If values are in a range of [5, 1000], this gets normalized to [5, 365].

lon_minmax, lat_minmax

[(float, float) or None] If not None, the computed bounding box is clipped to the given range in the corresponding direction.

__call__(*self*, *transform_xy*, *x1*, *y1*, *x2*, *y2*)

Compute an approximation of the bounding box obtained by applying *transform_xy* to the box delimited by `(x1, y1, x2, y2)`.

The intended use is to have `(x1, y1, x2, y2)` in axes coordinates, and have *transform_xy* be the transform from axes coordinates to data coordinates; this method then returns the range of data coordinates that span the actual axes.

The computation is done by sampling `nx * ny` equispaced points in the `(x1, y1, x2, y2)` box and finding the resulting points with extremal coordinates; then adding some padding to take into account the finite sampling.

As each sampling step covers a relative range of *1/nx* or *1/ny*, the padding is computed by expanding the span covered by the extremal coordinates by these fractions.

`__init__`(*self*, *nx*, *ny*, *lon_cycle=360.0*, *lat_cycle=None*, *lon_minmax=None*, *lat_minmax=- 90, 90*)

This subclass handles the case where one or both coordinates should be taken modulo 360, or be restricted to not exceed a specific range.

Parameters

nx, ny

[int] The number of samples in each direction.

lon_cycle, lat_cycle

[360 or None] If not None, values in the corresponding direction are taken modulo *lon_cycle* or *lat_cycle*; in theory this can be any number but the implementation actually assumes that it is 360 (if not None); other values give nonsensical results.

This is done by "unwrapping" the transformed grid coordinates so that jumps are less than a half-cycle; then normalizing the span to no more than a full cycle.

For example, if values are in the union of the [0, 2] and [358, 360] intervals (typically, angles measured modulo 360), the values in the second interval are normalized to [-2, 0] instead so that the values now cover [-2, 2]. If values are in a range of [5, 1000], this gets normalized to [5, 365].

lon_minmax, lat_minmax

[(float, float) or None] If not None, the computed bounding box is clipped to the given range in the corresponding direction.

`__module__ = 'mpl_toolkits.axisartist.angle_helper'`

Examples using `mpl_toolkits.axisartist.angle_helper.ExtremeFinderCycle`

- sphx_glr_gallery_axisartist_demo_axis_direction.py
- sphx_glr_gallery_axisartist_demo_curvelinear_grid.py
- sphx_glr_gallery_axisartist_demo_floating_axis.py
- sphx_glr_gallery_axisartist_simple_axis_pad.py

mpl_toolkits.axisartist.angle_helper.FormatterDMS

class mpl_toolkits.axisartist.angle_helper.FormatterDMS

Bases: object

__call__(*self*, *direction*, *factor*, *values*)

Call self as a function.

__dict__ = mappingproxy({'__module__': 'mpl_toolkits.axisartist.angle_helper', 'deg_mark'

__module__ = 'mpl_toolkits.axisartist.angle_helper'

__weakref__

list of weak references to the object (if defined)

deg_mark = '^{\\circ}'

fmt_d = '$%d^{\\circ}$'

fmt_d_m = '$%s%d^{\\circ}\\,%02d^{\\prime}$'

fmt_d_m_partial = '$%s%d^{\\circ}\\,%02d^{\\prime}\\,'

fmt_d_ms = '$%s%d^{\\circ}\\,%02d.%s^{\\prime}$'

fmt_ds = '$%d.%s^{\\circ}$'

fmt_s_partial = '%02d^{\\prime\\prime}$'

fmt_ss_partial = '%02d.%s^{\\prime\\prime}$'

min_mark = '^{\\prime}'

sec_mark = '^{\\prime\\prime}'

Examples using mpl_toolkits.axisartist.angle_helper.FormatterDMS

- sphx_glr_gallery_axisartist_demo_axis_direction.py
- sphx_glr_gallery_axisartist_demo_curvelinear_grid.py
- sphx_glr_gallery_axisartist_demo_floating_axes.py
- sphx_glr_gallery_axisartist_demo_floating_axis.py
- sphx_glr_gallery_axisartist_simple_axis_pad.py

mpl_toolkits.axisartist.angle_helper.FormatterHMS

class mpl_toolkits.axisartist.angle_helper.FormatterHMS

Bases: *mpl_toolkits.axisartist.angle_helper.FormatterDMS*

__call__(*self*, *direction*, *factor*, *values*)

Call self as a function.

__module__ = 'mpl_toolkits.axisartist.angle_helper'

deg_mark = '^\\mathrm{h}'

fmt_d = '$%d^\\mathrm{h}$'

fmt_d_m = '$%s%d^\\mathrm{h}\\,%02d^\\mathrm{m}$'

fmt_d_m_partial = '$%s%d^\\mathrm{h}\\,%02d^\\mathrm{m}\\,'

fmt_d_ms = '$%s%d^\\mathrm{h}\\,%02d.%s^\\mathrm{m}$'

fmt_ds = '$%d.%s^\\mathrm{h}$'

fmt_s_partial = '%02d^\\mathrm{s}$'

fmt_ss_partial = '%02d.%s^\\mathrm{s}$'

min_mark = '^\\mathrm{m}'

sec_mark = '^\\mathrm{s}'

Examples using mpl_toolkits.axisartist.angle_helper.FormatterHMS

- sphx_glr_gallery_axisartist_demo_floating_axes.py

mpl_toolkits.axisartist.angle_helper.LocatorBase

class mpl_toolkits.axisartist.angle_helper.LocatorBase(**kwargs)

Bases: object

__dict__ = mappingproxy({'__module__': 'mpl_toolkits.axisartist.angle_helper', '__init__'

__init__(*self*, *nbins*, *include_last=True*)

Initialize self. See help(type(self)) for accurate signature.

__module__ = 'mpl_toolkits.axisartist.angle_helper'

__weakref__

list of weak references to the object (if defined)

property den

set_params(*self*, *nbins=None*)

Examples using `mpl_toolkits.axisartist.angle_helper.LocatorBase`

- sphx_glr_gallery_axisartist_demo_axis_direction.py
- sphx_glr_gallery_axisartist_demo_curvelinear_grid.py
- sphx_glr_gallery_axisartist_demo_floating_axes.py
- sphx_glr_gallery_axisartist_demo_floating_axis.py
- sphx_glr_gallery_axisartist_simple_axis_pad.py

mpl_toolkits.axisartist.angle_helper.LocatorD

class mpl_toolkits.axisartist.angle_helper.LocatorD(**kwargs)

Bases: *mpl_toolkits.axisartist.angle_helper.LocatorBase*

__call__(*self*, *v1*, *v2*)

Call self as a function.

__module__ = 'mpl_toolkits.axisartist.angle_helper'

Examples using `mpl_toolkits.axisartist.angle_helper.LocatorD`

mpl_toolkits.axisartist.angle_helper.LocatorDM

class mpl_toolkits.axisartist.angle_helper.LocatorDM(**kwargs)

Bases: *mpl_toolkits.axisartist.angle_helper.LocatorBase*

__call__(*self*, *v1*, *v2*)

Call self as a function.

__module__ = 'mpl_toolkits.axisartist.angle_helper'

Examples using `mpl_toolkits.axisartist.angle_helper.LocatorDM`

mpl_toolkits.axisartist.angle_helper.LocatorDMS

class mpl_toolkits.axisartist.angle_helper.LocatorDMS(**kwargs)

Bases: *mpl_toolkits.axisartist.angle_helper.LocatorBase*

__call__(*self*, *v1*, *v2*)

Call self as a function.

__module__ = 'mpl_toolkits.axisartist.angle_helper'

Examples using `mpl_toolkits.axisartist.angle_helper.LocatorDMS`

- sphx_glr_gallery_axisartist_demo_axis_direction.py
- sphx_glr_gallery_axisartist_demo_curvelinear_grid.py
- sphx_glr_gallery_axisartist_demo_floating_axis.py
- sphx_glr_gallery_axisartist_simple_axis_pad.py

mpl_toolkits.axisartist.angle_helper.LocatorH

class mpl_toolkits.axisartist.angle_helper.LocatorH(**kwargs)

Bases: *mpl_toolkits.axisartist.angle_helper.LocatorBase*

__call__(*self*, *v1*, *v2*)

Call self as a function.

__module__ = 'mpl_toolkits.axisartist.angle_helper'

Examples using `mpl_toolkits.axisartist.angle_helper.LocatorH`

mpl_toolkits.axisartist.angle_helper.LocatorHM

class mpl_toolkits.axisartist.angle_helper.LocatorHM(**kwargs)

Bases: *mpl_toolkits.axisartist.angle_helper.LocatorBase*

__call__(*self*, *v1*, *v2*)

Call self as a function.

__module__ = 'mpl_toolkits.axisartist.angle_helper'

Examples using `mpl_toolkits.axisartist.angle_helper.LocatorHM`

mpl_toolkits.axisartist.angle_helper.LocatorHMS

class mpl_toolkits.axisartist.angle_helper.LocatorHMS(**kwargs)

Bases: *mpl_toolkits.axisartist.angle_helper.LocatorBase*

__call__(*self*, *v1*, *v2*)

Call self as a function.

__module__ = 'mpl_toolkits.axisartist.angle_helper'

Examples using `mpl_toolkits.axisartist.angle_helper.LocatorHMS`

- sphx_glr_gallery_axisartist_demo_floating_axes.py

Functions

select_step(v1, v2, nv[, hour, ...])	
select_step24(v1, v2, nv[, include_last, ...])	
select_step360(v1, v2, nv[, include_last, ...])	
select_step_degree(dv)	
select_step_hour(dv)	
select_step_sub(dv)	

mpl_toolkits.axisartist.angle_helper.select_step

mpl_toolkits.axisartist.angle_helper.**select_step**(*v1*, *v2*, *nv*, *hour=False*, *include_last=True*, *threshold_factor=3600.0*)

Examples using `mpl_toolkits.axisartist.angle_helper.select_step`

mpl_toolkits.axisartist.angle_helper.select_step24

mpl_toolkits.axisartist.angle_helper.**select_step24**(*v1*, *v2*, *nv*, *include_last=True*, *threshold_factor=3600*)

Examples using `mpl_toolkits.axisartist.angle_helper.select_step24`

mpl_toolkits.axisartist.angle_helper.select_step360

mpl_toolkits.axisartist.angle_helper.**select_step360**(*v1*, *v2*, *nv*, *include_last=True*, *threshold_factor=3600*)

Examples using `mpl_toolkits.axisartist.angle_helper.select_step360`

mpl_toolkits.axisartist.angle_helper.select_step_degree

mpl_toolkits.axisartist.angle_helper.**select_step_degree**(*dv*)

Examples using `mpl_toolkits.axisartist.angle_helper.select_step_degree`

mpl_toolkits.axisartist.angle_helper.select_step_hour

mpl_toolkits.axisartist.angle_helper.**select_step_hour**(*dv*)

Examples using `mpl_toolkits.axisartist.angle_helper.select_step_hour`

mpl_toolkits.axisartist.angle_helper.select_step_sub

mpl_toolkits.axisartist.angle_helper.**select_step_sub**(*dv*)

Examples using `mpl_toolkits.axisartist.angle_helper.select_step_sub`

mpl_toolkits.axisartist.axes_divider

mpl_toolkits.axisartist.axes_grid

Classes

AxesGrid	alias of *mpl_toolkits.axisartist.axes_grid.ImageGrid*
CbarAxes(*args, orientation, **kwargs)	
Grid(fig, rect, nrows_ncols[, ngrids, ...])	**Parameters**
ImageGrid(fig, rect, nrows_ncols[, ngrids, ...])	**Parameters**

mpl_toolkits.axisartist.axes_grid.AxesGrid

mpl_toolkits.axisartist.axes_grid.**AxesGrid**
alias of *mpl_toolkits.axisartist.axes_grid.ImageGrid*

mpl_toolkits.axisartist.axes_grid.CbarAxes

class mpl_toolkits.axisartist.axes_grid.**CbarAxes**(**args*, *orientation*, ***kwargs*)
Bases: *mpl_toolkits.axes_grid1.axes_grid.CbarAxesBase*, *mpl_toolkits.axisartist.axislines.Axes*

__module__ = 'mpl_toolkits.axisartist.axes_grid'

Examples using mpl_toolkits.axisartist.axes_grid.CbarAxes

mpl_toolkits.axisartist.axes_grid.Grid

class mpl_toolkits.axisartist.axes_grid.**Grid**(*fig*, *rect*, *nrows_ncols*, *ngrids=None*, *direction='row'*, *axes_pad=0.02*, *add_all=<deprecated parameter>*, *share_all=False*, *share_x=True*, *share_y=True*, *label_mode='L'*, *axes_class=None*, ***, *aspect=False*)
Bases: *mpl_toolkits.axes_grid1.axes_grid.Grid*

Parameters

fig

[*Figure*] The parent figure.

rect

[(float, float, float, float) or int] The axes position, as a `(left, bottom, width, height)` tuple or as a three-digit subplot position code (e.g., "121").

nrows_ncols

[(int, int)] Number of rows and columns in the grid.

ngrids

[int or None, default: None] If not None, only the first *ngrids* axes in the grid are created.

direction

[{"row", "column"}, default: "row"] Whether axes are created in row-major ("row by row") or column-major order ("column by column").

axes_pad

[float or (float, float), default: 0.02] Padding or (horizontal padding, vertical padding) between axes, in inches.

add_all

[bool, default: True] Whether to add the axes to the figure using *Figure.add_axes*. This parameter is deprecated.

share_all

[bool, default: False] Whether all axes share their x- and y-axis. Overrides *share_x* and *share_y*.

share_x

[bool, default: True] Whether all axes of a column share their x-axis.

share_y

[bool, default: True] Whether all axes of a row share their y-axis.

label_mode

[{"L", "1", "all"}, default: "L"] Determines which axes will get tick labels:

- "L": All axes on the left column get vertical tick labels; all axes on the bottom row get horizontal tick labels.
- "1": Only the bottom left axes is labelled.
- "all": all axes are labelled.

axes_class

[subclass of *matplotlib.axes.Axes*, default: None]

aspect

[bool, default: False] Whether the axes aspect ratio follows the aspect ratio of the data limits.

```
__module__ = 'mpl_toolkits.axisartist.axes_grid'
```

Examples using `mpl_toolkits.axisartist.axes_grid.Grid`

mpl_toolkits.axisartist.axes_grid.ImageGrid

class mpl_toolkits.axisartist.axes_grid.**ImageGrid**(*fig*, *rect*, *nrows_ncols*, *ngrids=None*, *direction='row'*, *axes_pad=0.02*, *add_all=<deprecated parameter>*, *share_all=False*, *aspect=True*, *label_mode='L'*, *cbar_mode=None*, *cbar_location='right'*, *cbar_pad=None*, *cbar_size='5%'*, *cbar_set_cax=True*, *axes_class=None*)

Bases: *mpl_toolkits.axes_grid1.axes_grid.ImageGrid*

Parameters

fig

[*Figure*] The parent figure.

rect

[(float, float, float, float) or int] The axes position, as a `(left, bottom, width, height)` tuple or as a three-digit subplot position code (e.g., "121").

nrows_ncols

[(int, int)] Number of rows and columns in the grid.

ngrids

[int or None, default: None] If not None, only the first *ngrids* axes in the grid are created.

direction

[{"row", "column"}, default: "row"] Whether axes are created in row-major ("row by row") or column-major order ("column by column"). This also affects the order in which axes are accessed using indexing (`grid[index]`).

axes_pad

[float or (float, float), default: 0.02in] Padding or (horizontal padding, vertical padding) between axes, in inches.

add_all

[bool, default: True] Whether to add the axes to the figure using *Figure.add_axes*. This parameter is deprecated.

share_all

[bool, default: False] Whether all axes share their x- and y-axis.

aspect

[bool, default: True] Whether the axes aspect ratio follows the aspect ratio of the data limits.

label_mode

[{"L", "1", "all"}, default: "L"] Determines which axes will get tick labels:

- "L": All axes on the left column get vertical tick labels; all axes on the bottom row get horizontal tick labels.
- "1": Only the bottom left axes is labelled.
- "all": all axes are labelled.

cbar_mode

[{"each", "single", "edge", None}, default: None] Whether to create a colorbar for "each" axes, a "single" colorbar for the entire grid, colorbars only for axes on the "edge" determined by *cbar_location*, or no colorbars. The colorbars are stored in the `cbar_axes` attribute.

cbar_location

[{"left", "right", "bottom", "top"}, default: "right"]

cbar_pad

[float, default: None] Padding between the image axes and the colorbar axes.

cbar_size

[size specification (see `Size.from_any`), default: "5%"] Colorbar size.

cbar_set_cax

[bool, default: True] If True, each axes in the grid has a *cax* attribute that is bound to associated *cbar_axes*.

axes_class

[subclass of *matplotlib.axes.Axes*, default: None]

```
__module__ = 'mpl_toolkits.axisartist.axes_grid'
```

Examples using `mpl_toolkits.axisartist.axes_grid.ImageGrid`

mpl_toolkits.axisartist.axes_rgb

Classes

RGBAxes(*args[, pad, add_all])	**Parameters**

mpl_toolkits.axisartist.axes_rgb.RGBAxes

class mpl_toolkits.axisartist.axes_rgb.**RGBAxes**(**args*, *pad=0*, *add_all=<deprecated parameter>*, ***kwargs*)

Bases: *mpl_toolkits.axes_grid1.axes_rgb.RGBAxes*

Parameters

pad

[float, default: 0] fraction of the axes height to put as padding.

add_all

[bool, default: True] Whether to add the {rgb, r, g, b} axes to the figure. This parameter is deprecated.

axes_class

[matplotlib.axes.Axes]

***args**

Unpacked into axes_class() init for RGB

****kwargs**

Unpacked into axes_class() init for RGB, R, G, B axes

```
__module__ = 'mpl_toolkits.axisartist.axes_rgb'
```

Examples using `mpl_toolkits.axisartist.axes_rgb.RGBAxes`

mpl_toolkits.axisartist.axis_artist

axis_artist.py module provides axis-related artists. They are

- axis line
- tick lines
- tick labels
- axis label
- grid lines

The main artist classes are *AxisArtist* and *GridlinesCollection*. While *GridlinesCollection* is responsible for drawing grid lines, *AxisArtist* is responsible for all other artists. *AxisArtist* has attributes that are associated with each type of artists:

- line: axis line
- major_ticks: major tick lines
- major_ticklabels: major tick labels
- minor_ticks: minor tick lines
- minor_ticklabels: minor tick labels
- label: axis label

Typically, the *AxisArtist* associated with an axes will be accessed with the *axis* dictionary of the axes, i.e., the *AxisArtist* for the bottom axis is

```
ax.axis["bottom"]
```

where *ax* is an instance of `mpl_toolkits.axislines.Axes`. Thus, `ax.axis["bottom"].line` is an artist associated with the axis line, and `ax.axis["bottom"].major_ticks` is an artist associated with the major tick lines.

You can change the colors, fonts, line widths, etc. of these artists by calling suitable set method. For example, to change the color of the major ticks of the bottom axis to red, use

```
ax.axis["bottom"].major_ticks.set_color("r")
```

However, things like the locations of ticks, and their ticklabels need to be changed from the side of the grid_helper.

axis_direction

AxisArtist, *AxisLabel*, *TickLabels* have an *axis_direction* attribute, which adjusts the location, angle, etc. The *axis_direction* must be one of "left", "right", "bottom", "top", and follows the Matplotlib convention for rectangular axis.

For example, for the *bottom* axis (the left and right is relative to the direction of the increasing coordinate),

- ticklabels and axislabel are on the right
- ticklabels and axislabel have text angle of 0
- ticklabels are baseline, center-aligned
- axislabel is top, center-aligned

The text angles are actually relative to (90 + angle of the direction to the ticklabel), which gives 0 for bottom axis.

Parameter	left	bottom	right	top
ticklabels location	left	right	right	left
axislabel location	left	right	right	left
ticklabels angle	90	0	-90	180
axislabel angle	180	0	0	180
ticklabel va	center	baseline	center	baseline
axislabel va	center	top	center	bottom
ticklabel ha	right	center	right	center
axislabel ha	right	center	right	center

Ticks are by default direct opposite side of the ticklabels. To make ticks to the same side of the ticklabels,

```
ax.axis["bottom"].major_ticks.set_ticks_out(True)
```

The following attributes can be customized (use the `set_xxx` methods):

- *Ticks*: ticksize, tick_out
- *TickLabels*: pad
- *AxisLabel*: pad

Classes

AttributeCopier(**kwargs)	[*Deprecated*]
AxisArtist(axes, helper[, offset, ...])	An artist which draws axis (a line along which the n-th axes coord is constant) line, ticks, ticklabels, and axis label.
AxisLabel(*args[, axis_direction, axis])	Axis Label.
BezierPath(**kwargs)	[*Deprecated*]
GridlinesCollection(*args[, which, axis])	**Parameters**
LabelBase(*args, **kwargs)	A base class for AxisLabel and TickLabels.
TickLabels(*[, axis_direction])	Tick Labels.
Ticks(ticksize[, tick_out, axis])	Ticks are derived from Line2D, and note that ticks themselves are markers.

mpl_toolkits.axisartist.axis_artist.AttributeCopier

class mpl_toolkits.axisartist.axis_artist.**AttributeCopier**(***kwargs*)

Bases: object

[*Deprecated*]

Notes

Deprecated since version 3.2:

__dict__ = mappingproxy({'__module__': 'mpl_toolkits.axisartist.axis_artist', '__init__':

__init__(*self*, *ref_artist*, *klass=<class 'matplotlib.artist.Artist'>*)

[*Deprecated*]

Notes

Deprecated since version 3.2:

__module__ = 'mpl_toolkits.axisartist.axis_artist'

__weakref__

list of weak references to the object (if defined)

get_attribute_from_ref_artist(*self*, *attr_name*, *default_value=<deprecated parameter>*)

get_ref_artist(*self*)
Return the underlying artist that actually defines some properties (e.g., color) of this artist.

set_ref_artist(*self*, *artist*)
[*Deprecated*]

Notes

Deprecated since version 3.2:

Examples using mpl_toolkits.axisartist.axis_artist.AttributeCopier

mpl_toolkits.axisartist.axis_artist.AxisArtist

class mpl_toolkits.axisartist.axis_artist.**AxisArtist**(*axes*, *helper*, *offset=None*, *axis_direction='bottom'*, ***kwargs*)

Bases: *matplotlib.artist.Artist*

An artist which draws axis (a line along which the n-th axes coord is constant) line, ticks, ticklabels, and axis label.

Parameters

axes

[*mpl_toolkits.axisartist.axislines.Axes*]

helper

[*AxisArtistHelper*]

property LABELPAD

ZORDER = 2.5

__init__(*self*, *axes*, *helper*, *offset=None*, *axis_direction='bottom'*, ***kwargs*)

Parameters

axes

[*mpl_toolkits.axisartist.axislines.Axes*]

helper

[*AxisArtistHelper*]

__module__ = 'mpl_toolkits.axisartist.axis_artist'

property dpi_transform

draw(*self*, *renderer*)
Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

Parameters

renderer

[*RendererBase* subclass.]

Notes

This method is overridden in the Artist subclasses.

get_axisline_style(*self*)
Return the current axisline style.

get_helper(*self*)
Return axis artist helper instance.

get_tightbbox(*self*, *renderer*)
Like *Artist.get_window_extent*, but includes any clipping.

Parameters

renderer

[*RendererBase* subclass] renderer that will be used to draw the figures (i.e. `fig.canvas.get_renderer()`)

Returns

Bbox

The enclosing bounding box (in figure pixel coordinates).

get_transform(*self*)
Return the *Transform* instance used by this artist.

invert_ticklabel_direction(*self*)

set_axis_direction(*self*, *axis_direction*)
Adjust the direction, text angle, text alignment of ticklabels, labels following the matplotlib convention for the rectangle axes.

The *axis_direction* must be one of [left, right, bottom, top].

property	left	bottom	right	top
ticklabels location	"-"	"+"	"+"	"-"
axislabel location	"-"	"+"	"+"	"-"
ticklabels angle	90	0	-90	180
ticklabel va	center	baseline	center	baseline
ticklabel ha	right	center	right	center
axislabel angle	180	0	0	180
axislabel va	center	top	center	bottom
axislabel ha	right	center	right	center

Note that the direction "+" and "-" are relative to the direction of the increasing coordinate. Also, the text angles are actually relative to (90 + angle of the direction to the ticklabel), which gives 0 for bottom axis.

`set_axislabel_direction`(*self*, *label_direction*)

Adjust the direction of the axislabel.

Note that the *label_direction*s '+' and '-' are relative to the direction of the increasing coordinate.

Parameters

tick_direction

[{"+", "-"}]

`set_axisline_style`(*self*, *axisline_style=None*, ***kwargs*)

Set the axisline style.

The new style is completely defined by the passed attributes. Existing style attributes are forgotten.

Parameters

axisline_style

[str or None] The line style, e.g. '->', optionally followed by a comma-separated list of attributes. Alternatively, the attributes can be provided as keywords.

If *None* this returns a string containing the available styles.

Examples

The following two commands are equal: >>> set_axisline_style("->,size=1.5") >>> set_axisline_style("->", size=1.5)

`set_label`(*self*, *s*)
Set a label that will be displayed in the legend.

Parameters

s

[object] *s* will be converted to a string by calling `str`.

`set_ticklabel_direction`(*self*, *tick_direction*)
Adjust the direction of the ticklabel.

Note that the *label_direction*s '+' and '-' are relative to the direction of the increasing coordinate.

Parameters

tick_direction

[{"+", "-"}]

`toggle`(*self*, *all=None*, *ticks=None*, *ticklabels=None*, *label=None*)
Toggle visibility of ticks, ticklabels, and (axis) label. To turn all off,

```
axis.toggle(all=False)
```

To turn all off but ticks on

```
axis.toggle(all=False, ticks=True)
```

To turn all on but (axis) label off

```
axis.toggle(all=True, label=False))
```

`zorder = 2.5`

Examples using `mpl_toolkits.axisartist.axis_artist.AxisArtist`

mpl_toolkits.axisartist.axis_artist.AxisLabel

`class mpl_toolkits.axisartist.axis_artist.AxisLabel`(**args*, *axis_direction='bottom'*, *axis=None*, ***kwargs*)
Bases: *mpl_toolkits.axisartist.axis_artist.AttributeCopier*, *mpl_toolkits.axisartist.axis_artist.LabelBase*

Axis Label. Derived from Text. The position of the text is updated in the fly, so changing text position has no effect. Otherwise, the properties can be changed as a normal Text.

To change the pad between ticklabels and axis label, use set_pad.

[*Deprecated*]

Notes

Deprecated since version 3.2:

`__init__`(*self*, **args*, *axis_direction='bottom'*, *axis=None*, ***kwargs*)
[*Deprecated*]

Notes

Deprecated since version 3.2:

`__module__ = 'mpl_toolkits.axisartist.axis_artist'`

`draw`(*self*, *renderer*)
Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

Parameters

renderer

[*RendererBase* subclass.]

Notes

This method is overridden in the Artist subclasses.

`get_color`(*self*)
Return the color of the text.

`get_pad`(*self*)
Return the internal pad in points.

See *set_pad* for more details.

`get_ref_artist`(*self*)
Return the underlying artist that actually defines some properties (e.g., color) of this artist.

`get_text`(*self*)
Return the text string.

`get_window_extent`(*self*, *renderer*)

Return the *Bbox* bounding the text, in display units.

In addition to being used internally, this is useful for specifying clickable regions in a png file on a web page.

Parameters

renderer

[Renderer, optional] A renderer is needed to compute the bounding box. If the artist has already been drawn, the renderer is cached; thus, it is only necessary to pass this argument when calling *get_window_extent* before the first *draw*. In practice, it is usually easier to trigger a draw first (e.g. by saving the figure).

dpi

[float, optional] The dpi value for computing the bbox, defaults to `self.figure.dpi` (*not* the renderer dpi); should be set e.g. if to match regions with a figure saved with a custom dpi value.

`set_axis_direction`(*self*, *d*)

Adjust the text angle and text alignment of axis label according to the matplotlib convention.

property	left	bottom	right	top
axislabel angle	180	0	0	180
axislabel va	center	top	center	bottom
axislabel ha	right	center	right	center

Note that the text angles are actually relative to (90 + angle of the direction to the ticklabel), which gives 0 for bottom axis.

`set_default_alignment`(*self*, *d*)

`set_default_angle`(*self*, *d*)

`set_pad`(*self*, *pad*)

Set the internal pad in points.

The actual pad will be the sum of the internal pad and the external pad (the latter is set automatically by the AxisArtist).

Examples using `mpl_toolkits.axisartist.axis_artist.AxisLabel`

mpl_toolkits.axisartist.axis_artist.BezierPath

class mpl_toolkits.axisartist.axis_artist.**BezierPath**(***kwargs*)

Bases: *matplotlib.lines.Line2D*

[*Deprecated*]

Notes

Deprecated since version 3.2:

Parameters

path

[*Path*] The path to draw.

****kwargs**

All remaining keyword arguments are passed to *Line2D*.

__init__(*self*, *path*, **args*, ***kwargs*)

Parameters

path

[*Path*] The path to draw.

****kwargs**

All remaining keyword arguments are passed to *Line2D*.

__module__ = 'mpl_toolkits.axisartist.axis_artist'

draw(*self*, *renderer*)

Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

Parameters

renderer

[*RendererBase* subclass.]

Notes

This method is overridden in the Artist subclasses.

`recache`(*self*)

`set_path`(*self*, *path*)

Examples using `mpl_toolkits.axisartist.axis_artist.BezierPath`

mpl_toolkits.axisartist.axis_artist.GridlinesCollection

`class mpl_toolkits.axisartist.axis_artist.GridlinesCollection`(**args*, *which='major'*, *axis='both'*, ***kwargs*)

Bases: *matplotlib.collections.LineCollection*

Parameters

which

[{"major", "minor"}]

axis

[{"both", "x", "y"}]

`__init__`(*self*, **args*, *which='major'*, *axis='both'*, ***kwargs*)

Parameters

which

[{"major", "minor"}]

axis

[{"both", "x", "y"}]

`__module__ = 'mpl_toolkits.axisartist.axis_artist'`

`draw`(*self*, *renderer*)

Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

Parameters

renderer

[*RendererBase* subclass.]

Notes

This method is overridden in the Artist subclasses.

`set_axis`(*self*, *axis*)

`set_grid_helper`(*self*, *grid_helper*)

`set_which`(*self*, *which*)

Examples using `mpl_toolkits.axisartist.axis_artist.GridlinesCollection`

mpl_toolkits.axisartist.axis_artist.LabelBase

`class mpl_toolkits.axisartist.axis_artist.LabelBase`(**args*, ***kwargs*)

Bases: *matplotlib.text.Text*

A base class for AxisLabel and TickLabels. The position and angle of the text are calculated by to offset_ref_angle, text_ref_angle, and offset_radius attributes.

Create a *Text* instance at *x*, *y* with string *text*.

Valid keyword arguments are:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array an
alpha	float or None
animated	bool
backgroundcolor	color
bbox	dict with properties for *patches.FancyBboxPatch*
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color or c	color
contains	unknown
figure	*Figure*
fontfamily or family	{FONTNAME, 'serif', 'sans-serif', 'cursive', 'fantasy', '
fontproperties or font or font_properties	*font_manager.FontProperties* or `str` or `pathlib.Path`
fontsize or size	float or {'xx-small', 'x-small', 'small', 'medium', 'large',
fontstretch or stretch	{a numeric value in range 0-1000, 'ultra-condensed',
fontstyle or style	{'normal', 'italic', 'oblique'}
fontvariant or variant	{'normal', 'small-caps'}
fontweight or weight	{a numeric value in range 0-1000, 'ultralight', 'light',
gid	str
horizontalalignment or ha	{'center', 'right', 'left'}
in_layout	bool
label	object

Tab

Property	Description
linespacing	float (multiple of font size)
multialignment or ma	{'left', 'right', 'center'}
path_effects	*AbstractPathEffect*
picker	None or bool or callable
position	(float, float)
rasterized	bool or None
rotation	float or {'vertical', 'horizontal'}
rotation_mode	{None, 'default', 'anchor'}
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
text	object
transform	*Transform*
url	str
usetex	bool or None
verticalalignment or va	{'center', 'top', 'bottom', 'baseline', 'center_baseline'}
visible	bool
wrap	bool
x	float
y	float
zorder	float

__init__(*self*, **args*, ***kwargs*)

Create a *Text* instance at *x*, *y* with string *text*.

Valid keyword arguments are:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array an
alpha	float or None
animated	bool
backgroundcolor	color
bbox	dict with properties for *patches.FancyBboxPatch*
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color or c	color
contains	unknown
figure	*Figure*
fontfamily or family	{FONTNAME, 'serif', 'sans-serif', 'cursive', 'fantasy', '
fontproperties or font or font_properties	*font_manager.FontProperties* or `str` or `pathlib.Path`
fontsize or size	float or {'xx-small', 'x-small', 'small', 'medium', 'large',
fontstretch or stretch	{a numeric value in range 0-1000, 'ultra-condensed',

Tab

Property	Description
fontstyle or style	{'normal', 'italic', 'oblique'}
fontvariant or variant	{'normal', 'small-caps'}
fontweight or weight	{a numeric value in range 0-1000, 'ultralight', 'light',
gid	str
horizontalalignment or ha	{'center', 'right', 'left'}
in_layout	bool
label	object
linespacing	float (multiple of font size)
multialignment or ma	{'left', 'right', 'center'}
path_effects	*AbstractPathEffect*
picker	None or bool or callable
position	(float, float)
rasterized	bool or None
rotation	float or {'vertical', 'horizontal'}
rotation_mode	{None, 'default', 'anchor'}
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
text	object
transform	*Transform*
url	str
usetex	bool or None
verticalalignment or va	{'center', 'top', 'bottom', 'baseline', 'center_baseline'}
visible	bool
wrap	bool
x	float
y	float
zorder	float

`__module__ = 'mpl_toolkits.axisartist.axis_artist'`

`draw`(*self*, *renderer*)

Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

Parameters

renderer

[*RendererBase* subclass.]

Notes

This method is overridden in the Artist subclasses.

`get_window_extent`(*self*, *renderer*)
Return the *Bbox* bounding the text, in display units.

In addition to being used internally, this is useful for specifying clickable regions in a png file on a web page.

Parameters

renderer

[Renderer, optional] A renderer is needed to compute the bounding box. If the artist has already been drawn, the renderer is cached; thus, it is only necessary to pass this argument when calling *get_window_extent* before the first *draw*. In practice, it is usually easier to trigger a draw first (e.g. by saving the figure).

dpi

[float, optional] The dpi value for computing the bbox, defaults to `self.figure.dpi` (*not* the renderer dpi); should be set e.g. if to match regions with a figure saved with a custom dpi value.

Examples using `mpl_toolkits.axisartist.axis_artist.LabelBase`

mpl_toolkits.axisartist.axis_artist.TickLabels

`class mpl_toolkits.axisartist.axis_artist.`**`TickLabels`**(*, *axis_direction='bottom'*, ***kwargs*)
Bases: *mpl_toolkits.axisartist.axis_artist.AxisLabel*

Tick Labels. While derived from Text, this single artist draws all ticklabels. As in AxisLabel, the position of the text is updated in the fly, so changing text position has no effect. Otherwise, the properties can be changed as a normal Text. Unlike the ticklabels of the mainline matplotlib, properties of single ticklabel alone cannot modified.

To change the pad between ticks and ticklabels, use set_pad.

[*Deprecated*]

Notes

Deprecated since version 3.2:

`__init__`(*self*, ***, *axis_direction='bottom'*, ***kwargs*)
[*Deprecated*]

Notes

Deprecated since version 3.2:

`__module__ = 'mpl_toolkits.axisartist.axis_artist'`

`draw`(*self*, *renderer*)
Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

Parameters

renderer

[*RendererBase* subclass.]

Notes

This method is overridden in the Artist subclasses.

`get_ref_artist`(*self*)
Return the underlying artist that actually defines some properties (e.g., color) of this artist.

`get_texts_widths_heights_descents`(*self*, *renderer*)
Return a list of `(width, height, descent)` tuples for ticklabels.

Empty labels are left out.

`get_window_extents`(*self*, *renderer*)

`invert_axis_direction`(*self*)

`set_axis_direction`(*self*, *label_direction*)
Adjust the text angle and text alignment of ticklabels according to the matplotlib convention.

The *label_direction* must be one of [left, right, bottom, top].

property	left	bottom	right	top
ticklabels angle	90	0	-90	180
ticklabel va	center	baseline	center	baseline
ticklabel ha	right	center	right	center

Note that the text angles are actually relative to (90 + angle of the direction to the ticklabel), which gives 0 for bottom axis.

`set_locs_angles_labels`(*self*, *locs_angles_labels*)

Examples using `mpl_toolkits.axisartist.axis_artist.TickLabels`

mpl_toolkits.axisartist.axis_artist.Ticks

`class mpl_toolkits.axisartist.axis_artist.Ticks`(*ticksize*, *tick_out=False*, *, *axis=None*, ***kwargs*)

Bases: *mpl_toolkits.axisartist.axis_artist.AttributeCopier*, *matplotlib.lines.Line2D*

Ticks are derived from Line2D, and note that ticks themselves are markers. Thus, you should use set_mec, set_mew, etc.

To change the tick size (length), you need to use set_ticksize. To change the direction of the ticks (ticks are in opposite direction of ticklabels by default), use set_tick_out(False).

[*Deprecated*]

Notes

Deprecated since version 3.2:

`__init__`(*self*, *ticksize*, *tick_out=False*, *, *axis=None*, ***kwargs*)

[*Deprecated*]

Notes

Deprecated since version 3.2:

`__module__ = 'mpl_toolkits.axisartist.axis_artist'`

`draw`(*self*, *renderer*)

Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

Parameters

renderer

[*RendererBase* subclass.]

Notes

This method is overridden in the Artist subclasses.

`get_color`(*self*)
Return the line color.

See also *set_color*.

`get_markeredgecolor`(*self*)
Return the marker edge color.

See also *set_markeredgecolor*.

`get_markeredgewidth`(*self*)
Return the marker edge width in points.

See also *set_markeredgewidth*.

`get_ref_artist`(*self*)
Return the underlying artist that actually defines some properties (e.g., color) of this artist.

`get_tick_out`(*self*)
Return whether ticks are drawn inside or outside the axes.

`get_ticksize`(*self*)
Return length of the ticks in points.

`set_locs_angles`(*self*, *locs_angles*)

`set_tick_out`(*self*, *b*)
Set whether ticks are drawn inside or outside the axes.

`set_ticksize`(*self*, *ticksize*)
Set length of the ticks in points.

Examples using `mpl_toolkits.axisartist.axis_artist.Ticks`

mpl_toolkits.axisartist.axisline_style

Classes

AxislineStyle(stylename, **kw)	A container class which defines style classes for AxisArtists.

mpl_toolkits.axisartist.axisline_style.AxislineStyle

class mpl_toolkits.axisartist.axisline_style.**AxislineStyle**(*stylename*, ***kw*)

Bases: matplotlib.patches._Style

A container class which defines style classes for AxisArtists.

An instance of any axisline style class is an callable object, whose call signature is

```
__call__(self, axis_artist, path, transform)
```

When called, this should return an *Artist* with the following methods:

```
def set_path(self, path):
    # set the path for axisline.

def set_line_mutation_scale(self, scale):
    # set the scale

def draw(self, renderer):
    # draw
```

Return the instance of the subclass with the given style name.

class **FilledArrow**(*size=1*)

Bases: *mpl_toolkits.axisartist.axisline_style.AxislineStyle.SimpleArrow*

Parameters

size

[float] Size of the arrow as a fraction of the ticklabel size.

ArrowAxisClass

alias of _FancyAxislineStyle.FilledArrow

__module__ = 'mpl_toolkits.axisartist.axisline_style'

class **SimpleArrow**(*size=1*)

Bases: mpl_toolkits.axisartist.axisline_style.AxislineStyle._Base

A simple arrow.

Parameters

size

[float] Size of the arrow as a fraction of the ticklabel size.

ArrowAxisClass

alias of _FancyAxislineStyle.SimpleArrow

__init__(*self*, *size=1*)

Parameters

size

[float] Size of the arrow as a fraction of the ticklabel size.

`__module__ = 'mpl_toolkits.axisartist.axisline_style'`

`new_line`(*self*, *axis_artist*, *transform*)

`__module__ = 'mpl_toolkits.axisartist.axisline_style'`

Examples using `mpl_toolkits.axisartist.axisline_style.AxislineStyle`

mpl_toolkits.axisartist.axislines

Axislines includes modified implementation of the Axes class. The biggest difference is that the artists responsible for drawing the axis spine, ticks, ticklabels and axis labels are separated out from Matplotlib's Axis class. Originally, this change was motivated to support curvilinear grid. Here are a few reasons that I came up with a new axes class:

- "top" and "bottom" x-axis (or "left" and "right" y-axis) can have different ticks (tick locations and labels). This is not possible with the current Matplotlib, although some twin axes trick can help.
- Curvilinear grid.
- angled ticks.

In the new axes class, xaxis and yaxis is set to not visible by default, and new set of artist (AxisArtist) are defined to draw axis line, ticks, ticklabels and axis label. Axes.axis attribute serves as a dictionary of these artists, i.e., ax.axis["left"] is a AxisArtist instance responsible to draw left y-axis. The default Axes.axis contains "bottom", "left", "top" and "right".

AxisArtist can be considered as a container artist and has following children artists which will draw ticks, labels, etc.

- line
- major_ticks, major_ticklabels
- minor_ticks, minor_ticklabels
- offsetText
- label

Note that these are separate artists from *`matplotlib.axis.Axis`*, thus most tick-related functions in Matplotlib won't work. For example, color and markerwidth of the `ax.axis["bottom"].major_ticks` will follow those of Axes.xaxis unless explicitly specified.

In addition to AxisArtist, the Axes will have *gridlines* attribute, which obviously draws grid lines. The gridlines needs to be separated from the axis as some gridlines can never pass any axis.

Classes

Axes(*args[, grid_helper])	Build an axes in a figure.
AxesZero(*args[, grid_helper])	Build an axes in a figure.
AxisArtistHelper()	AxisArtistHelper should define following method with given APIs.
AxisArtistHelperRectlinear()	
GridHelperBase()	
GridHelperRectlinear(axes)	

mpl_toolkits.axisartist.axislines.Axes

class mpl_toolkits.axisartist.axislines.**Axes**(**args*, *grid_helper=None*, ***kwargs*)

Bases: matplotlib.axes._axes.Axes

Build an axes in a figure.

Parameters

fig

[*Figure*] The axes is build in the *Figure* *fig*.

rect

[[left, bottom, width, height]] The axes is build in the rectangle *rect*. *rect* is in *Figure* coordinates.

sharex, sharey

[*Axes*, optional] The x or y *axis* is shared with the x or y axis in the input *Axes*.

frameon

[bool, default: True] Whether the axes frame is visible.

box_aspect

[None, or a number, optional] Sets the aspect of the axes box. See *set_box_aspect* for details.

****kwargs**

Other optional keyword arguments:

Property	Description
adjustable	{'box', 'datalim'}
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and r
alpha	float or None
anchor	2-tuple of floats or {'C', 'SW', 'S', 'SE', ...}
animated	bool
aspect	{'auto'} or num
autoscale_on	bool
autoscalex_on	bool
autoscaley_on	bool
axes_locator	Callable[[Axes, Renderer], Bbox]
axisbelow	bool or 'line'
box_aspect	None, or a number
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
contains	unknown
facecolor or fc	color
figure	*Figure*
frame_on	bool
gid	str
in_layout	bool
label	object
navigate	bool
navigate_mode	unknown
path_effects	*AbstractPathEffect*
picker	None or bool or callable
position	[left, bottom, width, height] or *Bbox*
prop_cycle	unknown
rasterization_zorder	float or None
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
title	str
transform	*Transform*
url	str
visible	bool
xbound	unknown
xlabel	str
xlim	(bottom: float, top: float)
xmargin	float greater than -0.5
xscale	{"linear", "log", "symlog", "logit", ...}
xticklabels	unknown
xticks	unknown

Table 35 – continued from previous page

Property	Description
ybound	unknown
ylabel	str
ylim	(bottom: float, top: float)
ymargin	float greater than -0.5
yscale	{"linear", "log", "symlog", "logit", ...}
yticklabels	unknown
yticks	unknown
zorder	float

Returns

Axes

The new *Axes* object.

`__call__`(*self*, **args*, ***kwargs*)
Call self as a function.

`__init__`(*self*, **args*, *grid_helper=None*, ***kwargs*)
Build an axes in a figure.

Parameters

fig

[*Figure*] The axes is build in the *Figure* *fig*.

rect

[[left, bottom, width, height]] The axes is build in the rectangle *rect*. *rect* is in *Figure* coordinates.

sharex, sharey

[*Axes*, optional] The x or y *axis* is shared with the x or y axis in the input *Axes*.

frameon

[bool, default: True] Whether the axes frame is visible.

box_aspect

[None, or a number, optional] Sets the aspect of the axes box. See *set_box_aspect* for details.

****kwargs**

Other optional keyword arguments:

Property	Description
adjustable	{'box', 'datalim'}
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and r
alpha	float or None
anchor	2-tuple of floats or {'C', 'SW', 'S', 'SE', ...}
animated	bool
aspect	{'auto'} or num
autoscale_on	bool
autoscalex_on	bool
autoscaley_on	bool
axes_locator	Callable[[Axes, Renderer], Bbox]
axisbelow	bool or 'line'
box_aspect	None, or a number
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
contains	unknown
facecolor or fc	color
figure	*Figure*
frame_on	bool
gid	str
in_layout	bool
label	object
navigate	bool
navigate_mode	unknown
path_effects	*AbstractPathEffect*
picker	None or bool or callable
position	[left, bottom, width, height] or *Bbox*
prop_cycle	unknown
rasterization_zorder	float or None
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
title	str
transform	*Transform*
url	str
visible	bool
xbound	unknown
xlabel	str
xlim	(bottom: float, top: float)
xmargin	float greater than -0.5
xscale	{"linear", "log", "symlog", "logit", ...}
xticklabels	unknown
xticks	unknown
ybound	unknown

Table 36 – continued from previous page

Property	Description
ylabel	str
ylim	(bottom: float, top: float)
ymargin	float greater than -0.5
yscale	{"linear", "log", "symlog", "logit", ...}
yticklabels	unknown
yticks	unknown
zorder	float

Returns

Axes

The new *Axes* object.

__module__ = 'mpl_toolkits.axisartist.axislines'

property axis

Convenience method to get or set some axis properties.

Call signatures:

```
xmin, xmax, ymin, ymax = axis()
xmin, xmax, ymin, ymax = axis([xmin, xmax, ymin, ymax])
xmin, xmax, ymin, ymax = axis(option)
xmin, xmax, ymin, ymax = axis(**kwargs)
```

Parameters

xmin, xmax, ymin, ymax

[float, optional] The axis limits to be set. This can also be achieved using

```
ax.set(xlim=(xmin, xmax), ylim=(ymin, ymax))
```

option

[bool or str] If a bool, turns axis lines and labels on or off. If a string, possible values are:

Value	Description
'on'	Turn on axis lines and labels. Same as `True`.
'off'	Turn off axis lines and labels. Same as `False`.
'equal'	Set equal scaling (i.e., make circles circular) by changing axis limits. This is the same as `ax.set_aspect('equal', adjustable='datalim')`. Explicit data limits may not be respected in this case.
'scaled'	Set equal scaling (i.e., make circles circular) by changing dimensions of the plot box. This is the same as `ax.set_aspect('equal', adjustable='box', anchor='C')`. Additionally, further autoscaling will be disabled.
'tight'	Set limits just large enough to show all data, then disable further autoscaling.
'auto'	Automatic scaling (fill plot box with data).
'image'	'scaled' with axis limits equal to data limits.
'square'	Square plot; similar to 'scaled', but initially forcing `xmax-xmin == ymax-ymin`.

emit

[bool, default: True] Whether observers are notified of the axis limit change. This option is passed on to *set_xlim* and *set_ylim*.

Returns

xmin, xmax, ymin, ymax

[float] The axis limits.

See also:

matplotlib.axes.Axes.set_xlim

matplotlib.axes.Axes.set_ylim

`cla`(*self*)
Clear the current axes.

`get_children`(*self*)
Return a list of the child *Artist*s of this *Artist*.

`get_grid_helper`(*self*)

`grid`(*self*, *b=None*, *which='major'*, *axis='both'*, ***kwargs*)
Toggle the gridlines, and optionally set the properties of the lines.

`invalidate_grid_helper`(*self*)

`new_fixed_axis`(*self*, *loc*, *offset=None*)

`new_floating_axis`(*self*, *nth_coord*, *value*, *axis_direction='bottom'*)

`new_gridlines`(*self*, *grid_helper=None*)
Create and return a new GridlineCollection instance.

which : "major" or "minor" *axis* : "both", "x" or "y"

`toggle_axisline`(*self*, *b=None*)

Examples using `mpl_toolkits.axisartist.axislines.Axes`

mpl_toolkits.axisartist.axislines.AxesZero

`class mpl_toolkits.axisartist.axislines.AxesZero`(**args*, *grid_helper=None*, ***kwargs*)

Bases: *mpl_toolkits.axisartist.axislines.Axes*

Build an axes in a figure.

Parameters

fig

[*Figure*] The axes is build in the *Figure* *fig*.

rect

[[left, bottom, width, height]] The axes is build in the rectangle *rect*. *rect* is in *Figure* coordinates.

sharex, sharey

[*Axes*, optional] The x or y *axis* is shared with the x or y axis in the input *Axes*.

frameon

[bool, default: True] Whether the axes frame is visible.

box_aspect

[None, or a number, optional] Sets the aspect of the axes box. See *set_box_aspect* for details.

****kwargs**

Other optional keyword arguments:

Property	Description
adjustable	{'box', 'datalim'}
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and re

Table 37 – continued from previous page

Property	Description
alpha	float or None
anchor	2-tuple of floats or {'C', 'SW', 'S', 'SE', ...}
animated	bool
aspect	{'auto'} or num
autoscale_on	bool
autoscalex_on	bool
autoscaley_on	bool
axes_locator	Callable[[Axes, Renderer], Bbox]
axisbelow	bool or 'line'
box_aspect	None, or a number
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
contains	unknown
facecolor or fc	color
figure	*Figure*
frame_on	bool
gid	str
in_layout	bool
label	object
navigate	bool
navigate_mode	unknown
path_effects	*AbstractPathEffect*
picker	None or bool or callable
position	[left, bottom, width, height] or *Bbox*
prop_cycle	unknown
rasterization_zorder	float or None
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
title	str
transform	*Transform*
url	str
visible	bool
xbound	unknown
xlabel	str
xlim	(bottom: float, top: float)
xmargin	float greater than -0.5
xscale	{"linear", "log", "symlog", "logit", ...}
xticklabels	unknown
xticks	unknown
ybound	unknown
ylabel	str

Table 37 – continued from previous page

Property	Description
ylim	(bottom: float, top: float)
ymargin	float greater than -0.5
yscale	{"linear", "log", "symlog", "logit", ...}
yticklabels	unknown
yticks	unknown
zorder	float

Returns

Axes

The new *Axes* object.

```
__module__ = 'mpl_toolkits.axisartist.axislines'
```

Examples using `mpl_toolkits.axisartist.axislines.AxesZero`

mpl_toolkits.axisartist.axislines.AxisArtistHelper

class mpl_toolkits.axisartist.axislines.**AxisArtistHelper**

Bases: `object`

AxisArtistHelper should define following method with given APIs. Note that the first axes argument will be axes attribute of the caller artist.:

```
# LINE (spinal line?)

def get_line(self, axes):
    # path : Path
    return path

def get_line_transform(self, axes):
    # ...
    # trans : transform
    return trans

# LABEL

def get_label_pos(self, axes):
    # x, y : position
    return (x, y), trans

def get_label_offset_transform(self,
        axes,
        pad_points, fontprops, renderer,
        bboxes,
        ):
```

(continues on next page)

(continued from previous page)

```
    # va : vertical alignment
    # ha : horizontal alignment
    # a : angle
    return trans, va, ha, a

# TICK

def get_tick_transform(self, axes):
    return trans

def get_tick_iterators(self, axes):
    # iter : iterable object that yields (c, angle, l) where
    # c, angle, l is position, tick angle, and label

    return iter_major, iter_minor
```

class Fixed(*loc*, *nth_coord=None*)

Bases: mpl_toolkits.axisartist.axislines.AxisArtistHelper._Base

Helper class for a fixed (in the axes coordinate) axis.

nth_coord = along which coordinate value varies in 2d, nth_coord = 0 -> x axis, nth_coord = 1 -> y axis

__init__(*self*, *loc*, *nth_coord=None*)

nth_coord = along which coordinate value varies in 2d, nth_coord = 0 -> x axis, nth_coord = 1 -> y axis

__module__ = 'mpl_toolkits.axisartist.axislines'

get_axislabel_pos_angle(*self*, *axes*)

Return the label reference position in transAxes.

get_label_transform() returns a transform of (transAxes+offset)

get_axislabel_transform(*self*, *axes*)

get_line(*self*, *axes*)

get_line_transform(*self*, *axes*)

get_nth_coord(*self*)

get_tick_transform(*self*, *axes*)

class Floating(*nth_coord*, *value*)

Bases: mpl_toolkits.axisartist.axislines.AxisArtistHelper._Base

__init__(*self*, *nth_coord*, *value*)

Initialize self. See help(type(self)) for accurate signature.

__module__ = 'mpl_toolkits.axisartist.axislines'

get_line(*self*, *axes*)

get_nth_coord(*self*)

__dict__ = mappingproxy({'__module__': 'mpl_toolkits.axisartist.axislines', '__doc__': '\

__module__ = 'mpl_toolkits.axisartist.axislines'

__weakref__
list of weak references to the object (if defined)

Examples using `mpl_toolkits.axisartist.axislines.AxisArtistHelper`

mpl_toolkits.axisartist.axislines.AxisArtistHelperRectlinear

class mpl_toolkits.axisartist.axislines.**AxisArtistHelperRectlinear**
Bases: `object`

class Fixed(*axes*, *loc*, *nth_coord=None*)
Bases: *mpl_toolkits.axisartist.axislines.AxisArtistHelper.Fixed*

nth_coord = along which coordinate value varies in 2d, nth_coord = 0 -> x axis, nth_coord = 1 -> y axis

__init__(*self*, *axes*, *loc*, *nth_coord=None*)
nth_coord = along which coordinate value varies in 2d, nth_coord = 0 -> x axis, nth_coord = 1 -> y axis

__module__ = 'mpl_toolkits.axisartist.axislines'

get_tick_iterators(*self*, *axes*)
tick_loc, tick_angle, tick_label

class Floating(*axes*, *nth_coord*, *passingthrough_point*, *axis_direction='bottom'*)
Bases: *mpl_toolkits.axisartist.axislines.AxisArtistHelper.Floating*

__init__(*self*, *axes*, *nth_coord*, *passingthrough_point*, *axis_direction='bottom'*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'mpl_toolkits.axisartist.axislines'

get_axislabel_pos_angle(*self*, *axes*)
Return the label reference position in transAxes.

get_label_transform() returns a transform of (transAxes+offset)

get_axislabel_transform(*self*, *axes*)

get_line(*self*, *axes*)

get_line_transform(*self*, *axes*)

get_tick_iterators(*self*, *axes*)
tick_loc, tick_angle, tick_label

get_tick_transform(*self*, *axes*)

__dict__ = mappingproxy({'__module__': 'mpl_toolkits.axisartist.axislines', 'Fixed': <cla

__module__ = 'mpl_toolkits.axisartist.axislines'

__weakref__
list of weak references to the object (if defined)

Examples using `mpl_toolkits.axisartist.axislines.AxisArtistHelperRectlinear`

mpl_toolkits.axisartist.axislines.GridHelperBase

class mpl_toolkits.axisartist.axislines.**GridHelperBase**
Bases: `object`

__dict__ = mappingproxy({'__module__': 'mpl_toolkits.axisartist.axislines', '__init__': <

__init__(*self*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'mpl_toolkits.axisartist.axislines'

__weakref__
list of weak references to the object (if defined)

get_gridlines(*self*, *which*, *axis*)
Return list of grid lines as a list of paths (list of points).

which : "major" or "minor" *axis* : "both", "x" or "y"

invalidate(*self*)

new_gridlines(*self*, *ax*)
Create and return a new GridlineCollection instance.

which : "major" or "minor" *axis* : "both", "x" or "y"

update_lim(*self*, *axes*)

valid(*self*)

Examples using `mpl_toolkits.axisartist.axislines.GridHelperBase`

mpl_toolkits.axisartist.axislines.GridHelperRectlinear

class mpl_toolkits.axisartist.axislines.**GridHelperRectlinear**(*axes*)
Bases: *mpl_toolkits.axisartist.axislines.GridHelperBase*

__init__(*self*, *axes*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'mpl_toolkits.axisartist.axislines'

`get_gridlines`(*self*, *which='major'*, *axis='both'*)
Return list of gridline coordinates in data coordinates.

which : "major" or "minor" *axis* : "both", "x" or "y"

`new_fixed_axis`(*self*, *loc*, *nth_coord=None*, *axis_direction=None*, *offset=None*, *axes=None*)

`new_floating_axis`(*self*, *nth_coord*, *value*, *axis_direction='bottom'*, *axes=None*)

Examples using `mpl_toolkits.axisartist.axislines.GridHelperRectlinear`

mpl_toolkits.axisartist.clip_path

Functions

atan2(dy, dx)	
clip(xlines, ylines, x0[, clip, xdir, ydir])	
clip_line_to_rect(xline, yline, bbox)	

mpl_toolkits.axisartist.clip_path.atan2

`mpl_toolkits.axisartist.clip_path.`**`atan2`**(*dy*, *dx*)

Examples using `mpl_toolkits.axisartist.clip_path.atan2`

mpl_toolkits.axisartist.clip_path.clip

`mpl_toolkits.axisartist.clip_path.`**`clip`**(*xlines*, *ylines*, *x0*, *clip='right'*, *xdir=True*, *ydir=True*)

Examples using `mpl_toolkits.axisartist.clip_path.clip`

mpl_toolkits.axisartist.clip_path.clip_line_to_rect

`mpl_toolkits.axisartist.clip_path.`**`clip_line_to_rect`**(*xline*, *yline*, *bbox*)

Examples using `mpl_toolkits.axisartist.clip_path.clip_line_to_rect`

mpl_toolkits.axisartist.floating_axes

An experimental support for curvilinear grid.

Classes

ExtremeFinderFixed(extremes)	This subclass always returns the same bounding box.
FixedAxisArtistHelper(grid_helper, side[, ...])	nth_coord = along which coordinate value varies.
FloatingAxes	alias of `mpl_toolkits.axisartist.floating_axes.Floating AxesHostAxes`
FloatingAxesBase(*args, **kwargs)	
FloatingAxisArtistHelper(grid_helper, ...[, ...])	nth_coord = along which coordinate value varies.
GridHelperCurveLinear(aux_trans, extremes[, ...])	aux_trans : a transform from the source (curved) coordinate to target (rectilinear) coordinate.

mpl_toolkits.axisartist.floating_axes.ExtremeFinderFixed

class mpl_toolkits.axisartist.floating_axes.**ExtremeFinderFixed**(*extremes*)

Bases: *mpl_toolkits.axisartist.grid_finder.ExtremeFinderSimple*

This subclass always returns the same bounding box.

Parameters

extremes

[(float, float, float, float)] The bounding box that this helper always returns.

__call__(*self*, *transform_xy*, *x1*, *y1*, *x2*, *y2*)

Compute an approximation of the bounding box obtained by applying *transform_xy* to the box delimited by `(x1, y1, x2, y2)`.

The intended use is to have `(x1, y1, x2, y2)` in axes coordinates, and have *transform_xy* be the transform from axes coordinates to data coordinates; this method then returns the range of data coordinates that span the actual axes.

The computation is done by sampling `nx * ny` equispaced points in the `(x1, y1, x2, y2)` box and finding the resulting points with extremal coordinates;

then adding some padding to take into account the finite sampling.

As each sampling step covers a relative range of *1/nx* or *1/ny*, the padding is computed by expanding the span covered by the extremal coordinates by these fractions.

`__init__`(*self*, *extremes*)
This subclass always returns the same bounding box.

Parameters

extremes

[(float, float, float, float)] The bounding box that this helper always returns.

`__module__ = 'mpl_toolkits.axisartist.floating_axes'`

Examples using `mpl_toolkits.axisartist.floating_axes.ExtremeFinderFixed`

mpl_toolkits.axisartist.floating_axes.FixedAxisArtistHelper

`class mpl_toolkits.axisartist.floating_axes.FixedAxisArtistHelper`(*grid_helper*, *side*, *nth_coord_ticks=None*)

Bases: *mpl_toolkits.axisartist.grid_helper_curvelinear.FloatingAxisArtistHelper*

nth_coord = along which coordinate value varies.

nth_coord = 0 -> x axis, nth_coord = 1 -> y axis

`__init__`(*self*, *grid_helper*, *side*, *nth_coord_ticks=None*)

nth_coord = along which coordinate value varies.

nth_coord = 0 -> x axis, nth_coord = 1 -> y axis

`__module__ = 'mpl_toolkits.axisartist.floating_axes'`

`get_line`(*self*, *axes*)

`get_tick_iterators`(*self*, *axes*)
tick_loc, tick_angle, tick_label, (optionally) tick_label

`update_lim`(*self*, *axes*)

Examples using mpl_toolkits.axisartist.floating_axes.FixedAxisArtistHelper

mpl_toolkits.axisartist.floating_axes.FloatingAxes

mpl_toolkits.axisartist.floating_axes.**FloatingAxes**
alias of mpl_toolkits.axisartist.floating_axes.Floating AxesHostAxes

mpl_toolkits.axisartist.floating_axes.FloatingAxesBase

class mpl_toolkits.axisartist.floating_axes.**FloatingAxesBase**(**args*, ***kwargs*)

Bases: object

__dict__ = mappingproxy({'__module__': 'mpl_toolkits.axisartist.floating_axes', '__init__

__init__(*self*, **args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'mpl_toolkits.axisartist.floating_axes'

__weakref__
list of weak references to the object (if defined)

adjust_axes_lim(*self*)

cla(*self*)

Examples using mpl_toolkits.axisartist.floating_axes.FloatingAxesBase

- sphx_glr_gallery_axisartist_demo_floating_axes.py

mpl_toolkits.axisartist.floating_axes.FloatingAxisArtistHelper

class mpl_toolkits.axisartist.floating_axes.**FloatingAxisArtistHelper**(*grid_helper*, *nth_coord*, *value*, *axis_direction=None*)

Bases: *mpl_toolkits.axisartist.grid_helper_curvelinear.FloatingAxisArtistHelper*

nth_coord = along which coordinate value varies.

nth_coord = 0 -> x axis, nth_coord = 1 -> y axis

__module__ = 'mpl_toolkits.axisartist.floating_axes'

Examples using `mpl_toolkits.axisartist.floating_axes.FloatingAxisArtistHelper`

mpl_toolkits.axisartist.floating_axes.GridHelperCurveLinear

class mpl_toolkits.axisartist.floating_axes.**GridHelperCurveLinear**(*aux_trans*, *extremes*, *grid_locator1=None*, *grid_locator2=None*, *tick_formatter1=None*, *tick_formatter2=None*)

Bases: *mpl_toolkits.axisartist.grid_helper_curvelinear.GridHelperCurveLinear*

aux_trans : a transform from the source (curved) coordinate to target (rectilinear) coordinate. An instance of MPL's Transform (inverse transform should be defined) or a tuple of two callable objects which defines the transform and its inverse. The callables need take two arguments of array of source coordinates and should return two target coordinates.

e.g., `x2, y2 = trans(x1, y1)`

__init__(*self*, *aux_trans*, *extremes*, *grid_locator1=None*, *grid_locator2=None*, *tick_formatter1=None*, *tick_formatter2=None*)

aux_trans : a transform from the source (curved) coordinate to target (rectilinear) coordinate. An instance of MPL's Transform (inverse transform should be defined) or a tuple of two callable objects which defines the transform and its inverse. The callables need take two arguments of array of source coordinates and should return two target coordinates.

e.g., `x2, y2 = trans(x1, y1)`

__module__ = 'mpl_toolkits.axisartist.floating_axes'

get_boundary(*self*)

Return (N, 2) array of (x, y) coordinate of the boundary.

get_data_boundary(*self*, *side*)

Return v=0, nth=1.

get_gridlines(*self*, *which='major'*, *axis='both'*)

Return list of grid lines as a list of paths (list of points).

which : "major" or "minor" *axis* : "both", "x" or "y"

new_fixed_axis(*self*, *loc*, *nth_coord=None*, *axis_direction=None*, *offset=None*, *axes=None*)

Examples using `mpl_toolkits.axisartist.floating_axes.GridHelperCurveLinear`

- sphx_glr_gallery_axisartist_demo_floating_axes.py

Functions

floatingaxes_class_factory(axes_class)	

mpl_toolkits.axisartist.floating_axes.floatingaxes_class_factory

mpl_toolkits.axisartist.floating_axes.**floatingaxes_class_factory**(*axes_class*)

Examples using `mpl_toolkits.axisartist.floating_axes.floatingaxes_class_factory`

mpl_toolkits.axisartist.grid_finder

Classes

DictFormatter(format_dict[, formatter])	format_dict : dictionary for format strings to be used.
ExtremeFinderSimple(nx, ny)	A helper class to figure out the range of grid lines that need to be drawn.
FixedLocator(locs)	
FormatterPrettyPrint([useMathText])	
GridFinder(transform[, extreme_finder, ...])	transform : transform from the image coordinate (which will be the transData of the axes to the world coordinate.
GridFinderBase(**kwargs)	[*Deprecated*]
MaxNLocator([nbins, steps, trim, integer, ...])	**Parameters**

mpl_toolkits.axisartist.grid_finder.DictFormatter

class mpl_toolkits.axisartist.grid_finder.DictFormatter(*format_dict*, *formatter=None*)

Bases: object

format_dict : dictionary for format strings to be used. formatter : fall-back formatter

__call__(*self*, *direction*, *factor*, *values*)
factor is ignored if value is found in the dictionary

__dict__ = mappingproxy({'__module__': 'mpl_toolkits.axisartist.grid_finder', '__init__':

__init__(*self*, *format_dict*, *formatter=None*)
format_dict : dictionary for format strings to be used. formatter : fall-back formatter

__module__ = 'mpl_toolkits.axisartist.grid_finder'

__weakref__
list of weak references to the object (if defined)

Examples using mpl_toolkits.axisartist.grid_finder.DictFormatter

- sphx_glr_gallery_axisartist_demo_floating_axes.py

mpl_toolkits.axisartist.grid_finder.ExtremeFinderSimple

class mpl_toolkits.axisartist.grid_finder.ExtremeFinderSimple(*nx*, *ny*)

Bases: object

A helper class to figure out the range of grid lines that need to be drawn.

Parameters

nx, ny

[int] The number of samples in each direction.

__call__(*self*, *transform_xy*, *x1*, *y1*, *x2*, *y2*)
Compute an approximation of the bounding box obtained by applying *transform_xy* to the box delimited by `(x1, y1, x2, y2)`.

The intended use is to have `(x1, y1, x2, y2)` in axes coordinates, and have *transform_xy* be the transform from axes coordinates to data coordinates; this method then returns the range of data coordinates that span the actual axes.

The computation is done by sampling `nx * ny` equispaced points in the `(x1, y1, x2, y2)` box and finding the resulting points with extremal coordinates; then adding some padding to take into account the finite sampling.

As each sampling step covers a relative range of *1/nx* or *1/ny*, the padding is computed by expanding the span covered by the extremal coordinates by these fractions.

`__dict__ = mappingproxy({'__module__': 'mpl_toolkits.axisartist.grid_finder', '__doc__':`

`__init__`(*self*, *nx*, *ny*)

Parameters

nx, ny

[int] The number of samples in each direction.

`__module__ = 'mpl_toolkits.axisartist.grid_finder'`

`__weakref__`
list of weak references to the object (if defined)

Examples using `mpl_toolkits.axisartist.grid_finder.ExtremeFinderSimple`

- sphx_glr_gallery_axisartist_demo_axis_direction.py
- sphx_glr_gallery_axisartist_demo_curvelinear_grid.py
- sphx_glr_gallery_axisartist_demo_curvelinear_grid2.py
- sphx_glr_gallery_axisartist_demo_floating_axis.py
- sphx_glr_gallery_axisartist_simple_axis_pad.py

mpl_toolkits.axisartist.grid_finder.FixedLocator

`class mpl_toolkits.axisartist.grid_finder.FixedLocator`(*locs*)
Bases: `object`

`__call__`(*self*, *v1*, *v2*)
Call self as a function.

`__dict__ = mappingproxy({'__module__': 'mpl_toolkits.axisartist.grid_finder', '__init__':`

`__init__`(*self*, *locs*)
Initialize self. See help(type(self)) for accurate signature.

`__module__ = 'mpl_toolkits.axisartist.grid_finder'`

`__weakref__`
list of weak references to the object (if defined)

set_factor(*self*, *f*)
[*Deprecated*]

Notes

Deprecated since version 3.3:

Examples using `mpl_toolkits.axisartist.grid_finder.FixedLocator`

- sphx_glr_gallery_axisartist_demo_floating_axes.py

mpl_toolkits.axisartist.grid_finder.FormatterPrettyPrint

class mpl_toolkits.axisartist.grid_finder.FormatterPrettyPrint(*useMathText=True*)
Bases: `object`

__call__(*self*, *direction*, *factor*, *values*)
Call self as a function.

__dict__ = mappingproxy({'__module__': 'mpl_toolkits.axisartist.grid_finder', '__init__':

__init__(*self*, *useMathText=True*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'mpl_toolkits.axisartist.grid_finder'

__weakref__
list of weak references to the object (if defined)

Examples using `mpl_toolkits.axisartist.grid_finder.FormatterPrettyPrint`

mpl_toolkits.axisartist.grid_finder.GridFinder

class mpl_toolkits.axisartist.grid_finder.GridFinder(*transform*, *extreme_finder=None*, *grid_locator1=None*, *grid_locator2=None*, *tick_formatter1=None*, *tick_formatter2=None*)
Bases: `object`

transform : transform from the image coordinate (which will be the transData of the axes to the world coordinate.

or transform = (transform_xy, inv_transform_xy)

locator1, locator2 : grid locator for 1st and 2nd axis.

__dict__ = mappingproxy({'__module__': 'mpl_toolkits.axisartist.grid_finder', '__init__':

__init__(*self*, *transform*, *extreme_finder=None*, *grid_locator1=None*, *grid_locator2=None*, *tick_formatter1=None*, *tick_formatter2=None*)

transform : transform from the image coordinate (which will be the trans-Data of the axes to the world coordinate.

or transform = (transform_xy, inv_transform_xy)

locator1, locator2 : grid locator for 1st and 2nd axis.

__module__ = 'mpl_toolkits.axisartist.grid_finder'

__weakref__

list of weak references to the object (if defined)

get_grid_info(*self*, *x1*, *y1*, *x2*, *y2*)

lon_values, lat_values

[list of grid values. if integer is given,] rough number of grids in each direction.

update(*self*, ***kw*)

update_transform(*self*, *aux_trans*)

Examples using mpl_toolkits.axisartist.grid_finder.GridFinder

mpl_toolkits.axisartist.grid_finder.GridFinderBase

class mpl_toolkits.axisartist.grid_finder.GridFinderBase(***kwargs*)

Bases: *mpl_toolkits.axisartist.grid_finder.GridFinder*

[*Deprecated*]

Notes

Deprecated since version 3.2:

transform : transform from the image coordinate (which will be the transData of the axes to the world coordinate.

or transform = (transform_xy, inv_transform_xy)

locator1, locator2 : grid locator for 1st and 2nd axis.

__init__(*self*, *extreme_finder*, *grid_locator1=None*, *grid_locator2=None*, *tick_formatter1=None*, *tick_formatter2=None*)

transform : transform from the image coordinate (which will be the trans-Data of the axes to the world coordinate.

or transform = (transform_xy, inv_transform_xy)

locator1, locator2 : grid locator for 1st and 2nd axis.

```
__module__ = 'mpl_toolkits.axisartist.grid_finder'
```

Examples using `mpl_toolkits.axisartist.grid_finder.GridFinderBase`

mpl_toolkits.axisartist.grid_finder.MaxNLocator

```
class mpl_toolkits.axisartist.grid_finder.MaxNLocator(nbins=10, steps=None,
                                                      trim=True, integer=False,
                                                      symmetric=False,
                                                      prune=None)
```

Bases: *matplotlib.ticker.MaxNLocator*

Parameters

nbins

[int or 'auto', default: 10] Maximum number of intervals; one less than max number of ticks. If the string 'auto', the number of bins will be automatically determined based on the length of the axis.

steps

[array-like, optional] Sequence of nice numbers starting with 1 and ending with 10; e.g., [1, 2, 4, 5, 10], where the values are acceptable tick multiples. i.e. for the example, 20, 40, 60 would be an acceptable set of ticks, as would 0.4, 0.6, 0.8, because they are multiples of 2. However, 30, 60, 90 would not be allowed because 3 does not appear in the list of steps.

integer

[bool, default: False] If True, ticks will take only integer values, provided at least *min_n_ticks* integers are found within the view limits.

symmetric

[bool, default: False] If True, autoscaling will result in a range symmetric about zero.

prune

[{'lower', 'upper', 'both', None}, default: None] Remove edge ticks -- useful for stacked or ganged plots where the upper tick of one axes overlaps with the lower tick of the axes above it, primarily when `rcParams["axes.autolimit_mode"]` (default: `'data'`) is

'round_numbers'. If prune=='lower', the smallest tick will be removed. If prune == 'upper', the largest tick will be removed. If prune == 'both', the largest and smallest ticks will be removed. If *prune* is *None*, no ticks will be removed.

min_n_ticks

[int, default: 2] Relax *nbins* and *integer* constraints if necessary to obtain this minimum number of ticks.

__call__(*self*, *v1*, *v2*)

Return the locations of the ticks.

__init__(*self*, *nbins=10*, *steps=None*, *trim=True*, *integer=False*, *symmetric=False*, *prune=None*)

Parameters

nbins

[int or 'auto', default: 10] Maximum number of intervals; one less than max number of ticks. If the string 'auto', the number of bins will be automatically determined based on the length of the axis.

steps

[array-like, optional] Sequence of nice numbers starting with 1 and ending with 10; e.g., [1, 2, 4, 5, 10], where the values are acceptable tick multiples. i.e. for the example, 20, 40, 60 would be an acceptable set of ticks, as would 0.4, 0.6, 0.8, because they are multiples of 2. However, 30, 60, 90 would not be allowed because 3 does not appear in the list of steps.

integer

[bool, default: False] If True, ticks will take only integer values, provided at least *min_n_ticks* integers are found within the view limits.

symmetric

[bool, default: False] If True, autoscaling will result in a range symmetric about zero.

prune

[{'lower', 'upper', 'both', None}, default: None] Remove edge ticks -- useful for stacked or ganged plots where the upper tick of one axes overlaps with the lower tick of the axes above it, primarily when rcParams["axes.autolimit_mode"] (default: 'data') is 'round_numbers'. If prune=='lower', the smallest tick will be removed. If prune == 'upper', the largest tick will be removed.

If `prune == 'both'`, the largest and smallest ticks will be removed. If *prune* is *None*, no ticks will be removed.

min_n_ticks

[int, default: 2] Relax *nbins* and *integer* constraints if necessary to obtain this minimum number of ticks.

`__module__ = 'mpl_toolkits.axisartist.grid_finder'`

`set_factor`(*self*, *f*)
[*Deprecated*]

Notes

Deprecated since version 3.3:

Examples using `mpl_toolkits.axisartist.grid_finder.MaxNLocator`

- sphx_glr_gallery_axisartist_demo_axis_direction.py
- sphx_glr_gallery_axisartist_demo_curvelinear_grid2.py
- sphx_glr_gallery_axisartist_demo_floating_axes.py
- sphx_glr_gallery_axisartist_simple_axis_pad.py

mpl_toolkits.axisartist.grid_helper_curvelinear

An experimental support for curvilinear grid.

Classes

FixedAxisArtistHelper(grid_helper, side[, ...])	Helper class for a fixed axis.
FloatingAxisArtistHelper(grid_helper, ...[, ...])	nth_coord = along which coordinate value varies.
GridHelperCurveLinear(aux_trans[, ...])	aux_trans : a transform from the source (curved) coordinate to target (rectilinear) coordinate.

mpl_toolkits.axisartist.grid_helper_curvelinear.FixedAxisArtistHelper

class mpl_toolkits.axisartist.grid_helper_curvelinear.**FixedAxisArtistHelper**(*grid_helper*, *side*, *nth_coord_ticks=N*

Bases: *mpl_toolkits.axisartist.axislines.AxisArtistHelper.Fixed*

Helper class for a fixed axis.

nth_coord = along which coordinate value varies.

nth_coord = 0 -> x axis, nth_coord = 1 -> y axis

`__init__`(*self*, *grid_helper*, *side*, *nth_coord_ticks=None*)

nth_coord = along which coordinate value varies.

nth_coord = 0 -> x axis, nth_coord = 1 -> y axis

`__module__ = 'mpl_toolkits.axisartist.grid_helper_curvelinear'`

`change_tick_coord`(*self*, *coord_number=None*)

`get_tick_iterators`(*self*, *axes*)
tick_loc, tick_angle, tick_label

`get_tick_transform`(*self*, *axes*)

`update_lim`(*self*, *axes*)

Examples using `mpl_toolkits.axisartist.grid_helper_curvelinear.FixedAxisArtistHelper`

mpl_toolkits.axisartist.grid_helper_curvelinear.FloatingAxisArtistHelper

class mpl_toolkits.axisartist.grid_helper_curvelinear.**FloatingAxisArtistHelper**(*grid_helper*, *nth_coord*, *value*, *axis_direction=*

Bases: *mpl_toolkits.axisartist.axislines.AxisArtistHelper.Floating*

nth_coord = along which coordinate value varies.

nth_coord = 0 -> x axis, nth_coord = 1 -> y axis

`__init__`(*self*, *grid_helper*, *nth_coord*, *value*, *axis_direction=None*)

nth_coord = along which coordinate value varies.

nth_coord = 0 -> x axis, nth_coord = 1 -> y axis

`__module__ = 'mpl_toolkits.axisartist.grid_helper_curvelinear'`

`get_axislabel_pos_angle`(*self*, *axes*)

`get_axislabel_transform`(*self*, *axes*)

`get_line`(*self*, *axes*)

`get_line_transform`(*self*, *axes*)

`get_tick_iterators`(*self*, *axes*)
tick_loc, tick_angle, tick_label, (optionally) tick_label

`get_tick_transform`(*self*, *axes*)

`set_extremes`(*self*, *e1*, *e2*)

`update_lim`(*self*, *axes*)

Examples using `mpl_toolkits.axisartist.grid_helper_curvelinear.FloatingAxisArtistHelper`

mpl_toolkits.axisartist.grid_helper_curvelinear.GridHelperCurveLinear

class mpl_toolkits.axisartist.grid_helper_curvelinear.**GridHelperCurveLinear**(*aux_trans*, *extreme_finder=None*, *grid_locator1=None*, *grid_locator2=None*, *tick_formatter1=None*, *tick_formatter2=None*)

Bases: *mpl_toolkits.axisartist.axislines.GridHelperBase*

aux_trans : a transform from the source (curved) coordinate to target (rectilinear) coordinate. An instance of MPL's Transform (inverse transform should be defined) or a tuple of two callable objects which defines the transform and its inverse. The callables need take two arguments of array of source coordinates and should return two target coordinates.

e.g., `x2, y2 = trans(x1, y1)`

`__init__`(*self*, *aux_trans*, *extreme_finder=None*, *grid_locator1=None*, *grid_locator2=None*, *tick_formatter1=None*, *tick_formatter2=None*)
aux_trans : a transform from the source (curved) coordinate to target (rectilinear) coordinate. An instance of MPL's Transform (inverse transform should be defined) or a tuple of two callable objects which defines the transform and its inverse. The callables need take two arguments of array of source coordinates and should return two target coordinates.

e.g., `x2, y2 = trans(x1, y1)`

`__module__ = 'mpl_toolkits.axisartist.grid_helper_curvelinear'`

`get_gridlines`(*self*, *which='major'*, *axis='both'*)
Return list of grid lines as a list of paths (list of points).

which : "major" or "minor" *axis* : "both", "x" or "y"

`get_tick_iterator`(*self*, *nth_coord*, *axis_side*, *minor=False*)

`new_fixed_axis`(*self*, *loc*, *nth_coord=None*, *axis_direction=None*, *offset=None*, *axes=None*)

`new_floating_axis`(*self*, *nth_coord*, *value*, *axes=None*, *axis_direction='bottom'*)

`update_grid_finder`(*self*, *aux_trans=None*, ***kw*)

Examples using `mpl_toolkits.axisartist.grid_helper_curvelinear.GridHelperCurveLinear`

mpl_toolkits.axisartist.parasite_axes

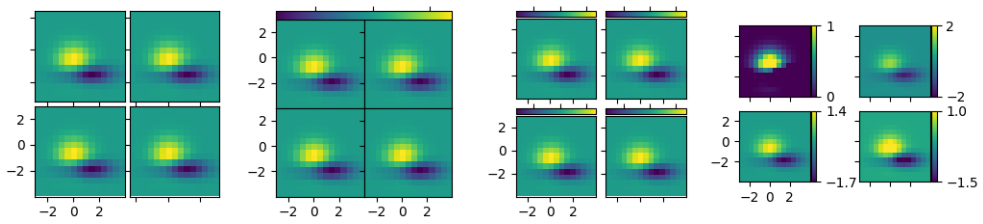
19.1.4 Matplotlib axes_grid Toolkit

Note: AxesGrid toolkit has been a part of matplotlib since v 0.99. Originally, the toolkit had a single namespace of *axes_grid*. In more recent version, the toolkit has divided into two separate namespace (*axes_grid1* and *axisartist*). While *axes_grid* namespace is maintained for the backward compatibility, use of *axes_grid1* and *axisartist* is recommended. For the documentation on `axes_grid`, see the previous version of the docs.

Matplotlib axes_grid1 Toolkit

The matplotlib *mpl_toolkits.axes_grid1* toolkit is a collection of helper classes to ease displaying multiple images in matplotlib. While the aspect parameter in matplotlib adjust the position of the single axes, axes_grid1 toolkit provides a framework to adjust the position of multiple axes according to their aspects.

See *What is axes_grid1 toolkit?* for a guide on the usage of axes_grid1.



The submodules of the axes_grid1 API are:

axes_grid1.anchored_artists

continues on next page

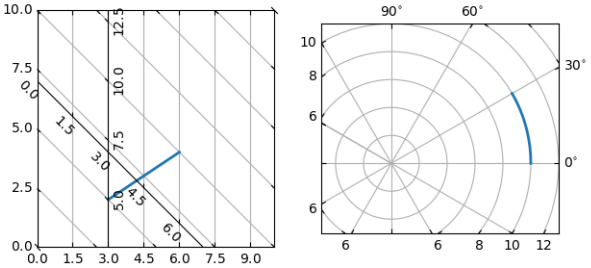
Table 43 – continued from previous page

axes_grid1.axes_divider	Helper classes to adjust the positions of multiple axes at drawing time.
axes_grid1.axes_grid	
axes_grid1.axes_rgb	
axes_grid1.axes_size	Provides classes of simple units that will be used with AxesDivider class (or others) to determine the size of each axes.
axes_grid1.colorbar	Colorbar toolkit with two classes and a function:
axes_grid1.inset_locator	A collection of functions and objects for creating or placing inset axes.
axes_grid1.mpl_axes	
axes_grid1.parasite_axes	

Matplotlib axisartist Toolkit

The *axisartist* namespace includes a derived Axes implementation (`mpl_toolkits.axisartist.Axes`). The biggest difference is that the artists that are responsible for drawing axis lines, ticks, ticklabels, and axis labels are separated out from the mpl's Axis class. This change was strongly motivated to support curvilinear grid.

You can find a tutorial describing usage of axisartist at the *axisartist* user guide.



The submodules of the axisartist API are:

axisartist.angle_helper	
axisartist.axes_divider	
axisartist.axes_grid	
axisartist.axes_rgb	
axisartist.axis_artist	axis_artist.py module provides axis-related artists.
axisartist.axisline_style	

continues on next page

Table 44 – continued from previous page

axisartist.axislines	Axislines includes modified implementation of the Axes class.
axisartist.clip_path	
axisartist.floating_axes	An experimental support for curvilinear grid.
axisartist.grid_finder	
axisartist.grid_helper_curvelinear	An experimental support for curvilinear grid.
axisartist.parasite_axes	

19.2 mplot3d API

Contents

19.2.1 axes3d

Note: Significant effort went into bringing axes3d to feature-parity with regular axes objects for version 1.1.0. However, more work remains. Please report any functions that do not behave as expected as a bug. In addition, help and patches would be greatly appreciated!

axes3d.Axes3D(fig[, rect, azim, elev, ...])	3D axes object.

mpl_toolkits.mplot3d.axes3d.Axes3D

class mpl_toolkits.mplot3d.axes3d.**Axes3D**(*fig*, *rect=None*, **args*, *azim=-60*, *elev=30*, *sharez=None*, *proj_type='persp'*, *box_aspect=None*, ***kwargs*)

Bases: `matplotlib.axes._axes.Axes`

3D axes object.

Parameters

fig

[Figure] The parent figure.

rect

[(float, float, float, float)] The `(left, bottom, width, height)` axes position.

azim

[float, default: -60] Azimuthal viewing angle.

elev

[float, default: 30] Elevation viewing angle.

sharez

[Axes3D, optional] Other axes to share z-limits with.

proj_type

[{'persp', 'ortho'}] The projection type, default 'persp'.

****kwargs**

Other optional keyword arguments:

Property	Description
adjustable	{'box', 'datalim'}
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and r
alpha	float or None
anchor	2-tuple of floats or {'C', 'SW', 'S', 'SE', ...}
animated	bool
aspect	{'auto'}
autoscale_on	bool
autoscalex_on	bool
autoscaley_on	bool
autoscalez_on	bool
axes_locator	Callable[[Axes, Renderer], Bbox]

Table 46 – continued from previous page

Property	Description
axisbelow	bool or 'line'
box_aspect	3-tuple of floats or None
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
contains	unknown
facecolor or fc	color
figure	*Figure*
frame_on	bool
gid	str
in_layout	bool
label	object
navigate	bool
navigate_mode	unknown
path_effects	*AbstractPathEffect*
picker	None or bool or callable
position	[left, bottom, width, height] or *Bbox*
proj_type	{'persp', 'ortho'}
prop_cycle	unknown
rasterization_zorder	float or None
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
title	str
transform	*Transform*
url	str
visible	bool
xbound	unknown
xlabel	str
xlim3d or xlim	unknown
xmargin	float greater than -0.5
xscale	{"linear"}
xticklabels	unknown
xticks	unknown
ybound	unknown
ylabel	str
ylim3d or ylim	unknown
ymargin	float greater than -0.5
yscale	{"linear"}
yticklabels	unknown
yticks	unknown
zbound	unknown
zlabel	unknown

Table 46 – continued from previous page

Property	Description
zlim3d or zlim	unknown
zmargin	unknown
zorder	float
zscale	{"linear"}
zticklabels	unknown
zticks	unknown

Notes

New in version 1.2.1: The *sharez* parameter.

`__init__`(*self*, *fig*, *rect=None*, **args*, *azim=- 60*, *elev=30*, *sharez=None*, *proj_type='persp'*, *box_aspect=None*, ***kwargs*)

Parameters

fig

[Figure] The parent figure.

rect

[(float, float, float, float)] The `(left, bottom, width, height)` axes position.

azim

[float, default: -60] Azimuthal viewing angle.

elev

[float, default: 30] Elevation viewing angle.

sharez

[Axes3D, optional] Other axes to share z-limits with.

proj_type

[{'persp', 'ortho'}] The projection type, default 'persp'.

****kwargs**

Other optional keyword arguments:

Property	Description
adjustable	{'box', 'datalim'}
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and re
alpha	float or None
anchor	2-tuple of floats or {'C', 'SW', 'S', 'SE', ...}

Table 47 – continued from previous page

Property	Description
animated	bool
aspect	{'auto'}
autoscale_on	bool
autoscalex_on	bool
autoscaley_on	bool
autoscalez_on	bool
axes_locator	Callable[[Axes, Renderer], Bbox]
axisbelow	bool or 'line'
box_aspect	3-tuple of floats or None
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
contains	unknown
facecolor or fc	color
figure	*Figure*
frame_on	bool
gid	str
in_layout	bool
label	object
navigate	bool
navigate_mode	unknown
path_effects	*AbstractPathEffect*
picker	None or bool or callable
position	[left, bottom, width, height] or *Bbox*
proj_type	{'persp', 'ortho'}
prop_cycle	unknown
rasterization_zorder	float or None
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
title	str
transform	*Transform*
url	str
visible	bool
xbound	unknown
xlabel	str
xlim3d or xlim	unknown
xmargin	float greater than -0.5
xscale	{"linear"}
xticklabels	unknown
xticks	unknown
ybound	unknown
ylabel	str

Table 47 – continued from previous page

Property	Description
ylim3d or ylim	unknown
ymargin	float greater than -0.5
yscale	{"linear"}
yticklabels	unknown
yticks	unknown
zbound	unknown
zlabel	unknown
zlim3d or zlim	unknown
zmargin	unknown
zorder	float
zscale	{"linear"}
zticklabels	unknown
zticks	unknown

Notes

New in version 1.2.1: The *sharez* parameter.

`__module__ = 'mpl_toolkits.mplot3d.axes3d'`

`add_collection3d`(*self*, *col*, *zs=0*, *zdir='z'*)
Add a 3D collection object to the plot.

2D collection types are converted to a 3D version by modifying the object and adding z coordinate information.

Supported are:

- PolyCollection
- LineCollection
- PatchCollection

`add_contour_set`(*self*, *cset*, *extend3d=False*, *stride=5*, *zdir='z'*, *offset=None*)

`add_contourf_set`(*self*, *cset*, *zdir='z'*, *offset=None*)

`apply_aspect`(*self*, *position=None*)
Adjust the Axes for a specified data aspect ratio.

Depending on *get_adjustable* this will modify either the Axes box (position) or the view limits. In the former case, *get_anchor* will affect the position.

See also:

matplotlib.axes.Axes.set_aspect

For a description of aspect ratio handling.

matplotlib.axes.Axes.set_adjustable

Set how the Axes adjusts to achieve the required aspect ratio.

matplotlib.axes.Axes.set_anchor

Set the position in case of extra space.

Notes

This is called automatically when each Axes is drawn. You may need to call it yourself if you need to update the Axes position and/or view limits before the Figure is drawn.

`auto_scale_xyz`(*self*, *X*, *Y*, *Z=None*, *had_data=None*)

`autoscale`(*self*, *enable=True*, *axis='both'*, *tight=None*)
Convenience method for simple axis view autoscaling. See *matplotlib.axes.Axes.autoscale()* for full explanation. Note that this function behaves the same, but for all three axes. Therefore, 'z' can be passed for *axis*, and 'both' applies to all three axes.

New in version 1.1.0.

`autoscale_view`(*self*, *tight=None*, *scalex=True*, *scaley=True*, *scalez=True*)
Autoscale the view limits using the data limits. See *matplotlib.axes.Axes.autoscale_view()* for documentation. Note that this function applies to the 3D axes, and as such adds the *scalez* to the function arguments.

Changed in version 1.1.0: Function signature was changed to better match the 2D version. *tight* is now explicitly a kwarg and placed first.

Changed in version 1.2.1: This is now fully functional.

`bar`(*self*, *left*, *height*, *zs=0*, *zdir='z'*, **args*, ***kwargs*)
Add 2D bar(s).

Parameters

left

[1D array-like] The x coordinates of the left sides of the bars.

height

[1D array-like] The height of the bars.

zs

[float or 1D array-like] Z coordinate of bars; if a single value is specified, it will be used for all bars.

zdir

[{'x', 'y', 'z'}, default: 'z'] When plotting 2D data, the direction to use as z ('x', 'y' or 'z').

****kwargs**

Other arguments are forwarded to *matplotlib.axes.Axes.bar*.

Returns

mpl_toolkits.mplot3d.art3d.Patch3DCollection

`bar3d`(*self*, *x*, *y*, *z*, *dx*, *dy*, *dz*, *color=None*, *zsort='average'*, *shade=True*, *lightsource=None*, **args*, ***kwargs*)

Generate a 3D barplot.

This method creates three dimensional barplot where the width, depth, height, and color of the bars can all be uniquely set.

Parameters

x, y, z

[array-like] The coordinates of the anchor point of the bars.

dx, dy, dz

[float or array-like] The width, depth, and height of the bars, respectively.

color

[sequence of colors, optional] The color of the bars can be specified globally or individually. This parameter can be:

- A single color, to color all bars the same color.
- An array of colors of length N bars, to color each bar independently.
- An array of colors of length 6, to color the faces of the bars similarly.
- An array of colors of length 6 * N bars, to color each face independently.

When coloring the faces of the boxes specifically, this is the order of the coloring:

1. -Z (bottom of box)
2. +Z (top of box)
3. -Y
4. +Y
5. -X
6. +X

zsort

[str, optional] The z-axis sorting scheme passed onto *Poly3DCollection*

shade

[bool, default: True] When true, this shades the dark sides of the bars (relative to the plot's source of light).

lightsource

[*LightSource*] The lightsource to use when *shade* is True.

****kwargs**

Any additional keyword arguments are passed onto *Poly3DCollection*.

Returns

collection

[*Poly3DCollection*] A collection of three dimensional polygons representing the bars.

`can_pan`(*self*)
Return *True* if this axes supports the pan/zoom button functionality.

3D axes objects do not use the pan/zoom button.

`can_zoom`(*self*)
Return *True* if this axes supports the zoom box button functionality.

3D axes objects do not use the zoom box button.

`cla`(*self*)
Clear the current axes.

`clabel`(*self*, **args*, ***kwargs*)
Currently not implemented for 3D axes, and returns *None*.

`contour`(*self*, *X*, *Y*, *Z*, **args*, *extend3d=False*, *stride=5*, *zdir='z'*, *offset=None*, ***kwargs*)
Create a 3D contour plot.

Parameters

X, Y, Z

[array-like] Input data.

extend3d

[bool, default: False] Whether to extend contour in 3D.

stride

[int] Step size for extending contour.

zdir

[{'x', 'y', 'z'}, default: 'z'] The direction to use.

offset

[float, optional] If specified, plot a projection of the contour lines at this position in a plane normal to zdir.

***args, **kwargs**

Other arguments are forwarded to *matplotlib.axes.Axes.contour*.

Returns

matplotlib.contour.QuadContourSet

`contour3D`(*self*, *X*, *Y*, *Z*, **args*, *extend3d=False*, *stride=5*, *zdir='z'*, *offset=None*, ***kwargs*)

Create a 3D contour plot.

Parameters

X, Y, Z

[array-like] Input data.

extend3d

[bool, default: False] Whether to extend contour in 3D.

stride

[int] Step size for extending contour.

zdir

[{'x', 'y', 'z'}, default: 'z'] The direction to use.

offset

[float, optional] If specified, plot a projection of the contour lines at this position in a plane normal to zdir.

***args, **kwargs**

Other arguments are forwarded to *matplotlib.axes.Axes.contour*.

Returns

matplotlib.contour.QuadContourSet

contourf(*self*, *X*, *Y*, *Z*, **args*, *zdir='z'*, *offset=None*, ***kwargs*)

Create a 3D filled contour plot.

Parameters

X, Y, Z

[array-like] Input data.

zdir

[{'x', 'y', 'z'}, default: 'z'] The direction to use.

offset

[float, optional] If specified, plot a projection of the contour lines at this position in a plane normal to zdir.

***args, **kwargs**

Other arguments are forwarded to *matplotlib.axes.Axes.contourf*.

Returns

matplotlib.contour.QuadContourSet

Notes

New in version 1.1.0: The *zdir* and *offset* parameters.

contourf3D(*self*, *X*, *Y*, *Z*, **args*, *zdir='z'*, *offset=None*, ***kwargs*)

Create a 3D filled contour plot.

Parameters

X, Y, Z

[array-like] Input data.

zdir

[{'x', 'y', 'z'}, default: 'z'] The direction to use.

offset

[float, optional] If specified, plot a projection of the contour lines at this position in a plane normal to zdir.

***args, **kwargs**

Other arguments are forwarded to *matplotlib.axes.Axes.contourf*.

Returns

matplotlib.contour.QuadContourSet

Notes

New in version 1.1.0: The *zdir* and *offset* parameters.

`convert_zunits`(*self*, *z*)
For artists in an axes, if the zaxis has units support, convert *z* using zaxis unit type

New in version 1.2.1.

`disable_mouse_rotation`(*self*)
Disable mouse buttons for 3D rotation and zooming.

`draw`(*self*, *renderer*)
Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

Parameters

renderer

[*RendererBase* subclass.]

Notes

This method is overridden in the Artist subclasses.

`format_coord`(*self*, *xd*, *yd*)
Given the 2D view coordinates attempt to guess a 3D coordinate. Looks for the nearest edge to the point and then assumes that the point is at the same z location as the nearest point on the edge.

`format_zdata`(*self*, *z*)
Return *z* string formatted. This function will use the `fmt_zdata` attribute if it is callable, else will fall back on the zaxis major formatter

`get_autoscale_on`(*self*)
Get whether autoscaling is applied for all axes on plot commands

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

`get_autoscalez_on`(*self*)
Get whether autoscaling for the z-axis is applied on plot commands

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

`get_axis_position`(*self*)

`get_frame_on`(*self*)
Get whether the 3D axes panels are drawn.

`get_proj`(*self*)
Create the projection matrix from the current viewing position.

`get_tightbbox`(*self*, *renderer*, *call_axes_locator=True*, *bbox_extra_artists=None*, *, *for_layout_only=False*)
Return the tight bounding box of the axes, including axis and their decorators (xlabel, title, etc).

Artists that have `artist.set_in_layout(False)` are not included in the bbox.

Parameters

renderer

[*RendererBase* subclass] renderer that will be used to draw the figures (i.e. `fig.canvas.get_renderer()`)

bbox_extra_artists

[list of *Artist* or `None`] List of artists to include in the tight bounding box. If `None` (default), then all artist children of the axes are included in the tight bounding box.

call_axes_locator

[bool, default: True] If *call_axes_locator* is `False`, it does not call the `_axes_locator` attribute, which is necessary to get the correct bounding box. `call_axes_locator=False` can be used if the caller is only interested in the relative size of the tightbbox compared to the axes bbox.

for_layout_only

[default: False] The bounding box will *not* include the x-extent of the title and the xlabel, or the y-extent of the ylabel.

Returns

BboxBase

Bounding box in figure pixel coordinates.

See also:

matplotlib.axes.Axes.get_window_extent

matplotlib.axis.Axis.get_tightbbox

matplotlib.spines.Spine.get_window_extent

`get_w_lims`(*self*)
Get 3D world limits.

get_xlim(*self*)
Alias for *get_xlim3d*.

get_xlim3d(*self*)
Return the x-axis view limits.

Returns

left, right

[(float, float)] The current x-axis limits in data coordinates.

See also:

set_xlim

set_xbound, get_xbound

invert_xaxis, xaxis_inverted

Notes

The x-axis may be inverted, in which case the *left* value will be greater than the *right* value.

Changed in version 1.1.0: This function now correctly refers to the 3D x-limits

get_ylim(*self*)
Alias for *get_ylim3d*.

get_ylim3d(*self*)
Return the y-axis view limits.

Returns

bottom, top

[(float, float)] The current y-axis limits in data coordinates.

See also:

set_ylim

set_ybound, get_ybound

invert_yaxis, yaxis_inverted

Notes

The y-axis may be inverted, in which case the *bottom* value will be greater than the *top* value.

Changed in version 1.1.0: This function now correctly refers to the 3D y-limits.

`get_zaxis`(*self*)
Return the `ZAxis` (*Axis*) instance.

`get_zbound`(*self*)
Return the lower and upper z-axis bounds, in increasing order.

New in version 1.1.0.

`get_zgridlines`(*self*)
Return the zaxis' grid lines as a list of *Line2D*s.

`get_zlabel`(*self*)
Get the z-label text string.

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

`get_zlim`(*self*)
Alias for *get_zlim3d*.

`get_zlim3d`(*self*)
Get 3D z limits.

`get_zmajorticklabels`(*self*)
Return the zaxis' major tick labels, as a list of *Text*.

`get_zminorticklabels`(*self*)
Return the zaxis' minor tick labels, as a list of *Text*.

`get_zscale`(*self*)

`get_zticklabels`(*self*, *minor=False*, *which=None*)
Get the zaxis' tick labels.

Parameters

minor

[bool] Whether to return the minor or the major ticklabels.

which

[None, ('minor', 'major', 'both')] Overrides *minor*.

Selects which ticklabels to return

Returns

list of *Text*

Notes

The tick label strings are not populated until a `draw` method has been called.

See also: *draw* and *draw*.

`get_zticklines`(*self*, *minor=False*)
Return the zaxis' tick lines as a list of *Line2D*s.

`get_zticks`(*self*, *, *minor=False*)
Return the zaxis' tick locations in data coordinates.

`grid`(*self*, *b=True*, ***kwargs*)
Set / unset 3D grid.

Note: Currently, this function does not behave the same as *matplotlib.axes.Axes.grid()*, but it is intended to eventually support that behavior.

New in version 1.1.0.

`invert_zaxis`(*self*)
Invert the z-axis.

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

`locator_params`(*self*, *axis='both'*, *tight=None*, ***kwargs*)
Convenience method for controlling tick locators.

See *matplotlib.axes.Axes.locator_params()* for full documentation. Note that this is for Axes3D objects, therefore, setting *axis* to 'both' will result in the parameters being set for all three axes. Also, *axis* can also take a value of 'z' to apply parameters to the z axis.

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

`margins`(*self*, **margins*, *x=None*, *y=None*, *z=None*, *tight=True*)
Convenience method to set or retrieve autoscaling margins.

Call signatures:

```
margins()
```

returns xmargin, ymargin, zmargin

```
margins(margin)

margins(xmargin, ymargin, zmargin)
```

(continues on next page)

(continued from previous page)

```
margins(x=xmargin, y=ymargin, z=zmargin)

margins(..., tight=False)
```

All forms above set the xmargin, ymargin and zmargin parameters. All keyword parameters are optional. A single positional argument specifies xmargin, ymargin and zmargin. Passing both positional and keyword arguments for xmargin, ymargin, and/or zmargin is invalid.

The *tight* parameter is passed to *autoscale_view()*, which is executed after a margin is changed; the default here is *True*, on the assumption that when margins are specified, no additional padding to match tick marks is usually desired. Setting *tight* to *None* will preserve the previous setting.

Specifying any margin changes only the autoscaling; for example, if *xmargin* is not None, then *xmargin* times the X data interval will be added to each end of that interval before it is used in autoscaling.

New in version 1.1.0.

`mouse_init`(*self*, *rotate_btn=1*, *zoom_btn=3*)

Set the mouse buttons for 3D rotation and zooming.

Parameters

rotate_btn

[int or list of int, default: 1] The mouse button or buttons to use for 3D rotation of the axes.

zoom_btn

[int or list of int, default: 3] The mouse button or buttons to use to zoom the 3D axes.

`name = '3d'`

`plot`(*self*, *xs*, *ys*, **args*, *zdir='z'*, ***kwargs*)

Plot 2D or 3D data.

Parameters

xs

[1D array-like] x coordinates of vertices.

ys

[1D array-like] y coordinates of vertices.

zs

[float or 1D array-like] z coordinates of vertices; either one for all points or one for each point.

zdir

[{'x', 'y', 'z'}, default: 'z'] When plotting 2D data, the direction to use as z ('x', 'y' or 'z').

****kwargs**

Other arguments are forwarded to *matplotlib.axes.Axes.plot*.

`plot3D`(*self*, *xs*, *ys*, **args*, *zdir='z'*, ***kwargs*)

Plot 2D or 3D data.

Parameters

xs

[1D array-like] x coordinates of vertices.

ys

[1D array-like] y coordinates of vertices.

zs

[float or 1D array-like] z coordinates of vertices; either one for all points or one for each point.

zdir

[{'x', 'y', 'z'}, default: 'z'] When plotting 2D data, the direction to use as z ('x', 'y' or 'z').

****kwargs**

Other arguments are forwarded to *matplotlib.axes.Axes.plot*.

`plot_surface`(*self*, *X*, *Y*, *Z*, **args*, *norm=None*, *vmin=None*, *vmax=None*, *lightsource=None*, ***kwargs*)

Create a surface plot.

By default it will be colored in shades of a solid color, but it also supports color mapping by supplying the *cmap* argument.

Note: The *rcount* and *ccount* kwargs, which both default to 50, determine the maximum number of samples used in each direction. If the input data is larger, it will be downsampled (by slicing) to these numbers of points.

Note: To maximize rendering speed consider setting *rstride* and *cstride* to divisors of the number of rows minus 1 and columns minus 1 respectively. For example, given 51 rows rstride can be any of the divisors of 50.

Similarly, a setting of *rstride* and *cstride* equal to 1 (or *rcount* and *ccount* equal the number of rows and columns) can use the optimized path.

Parameters

X, Y, Z

[2d arrays] Data values.

rcount, ccount

[int] Maximum number of samples used in each direction. If the input data is larger, it will be downsampled (by slicing) to these numbers of points. Defaults to 50.

New in version 2.0.

rstride, cstride

[int] Downsampling stride in each direction. These arguments are mutually exclusive with *rcount* and *ccount*. If only one of *rstride* or *cstride* is set, the other defaults to 10.

'classic' mode uses a default of `rstride = cstride = 10` instead of the new default of `rcount = ccount = 50`.

color

[color-like] Color of the surface patches.

cmap

[Colormap] Colormap of the surface patches.

facecolors

[array-like of colors.] Colors of each individual patch.

norm

[Normalize] Normalization for the colormap.

vmin, vmax

[float] Bounds for the normalization.

shade

[bool, default: True] Whether to shade the facecolors. Shading is always disabled when *cmap* is specified.

lightsource

[*LightSource*] The lightsource to use when *shade* is True.

****kwargs**

Other arguments are forwarded to *Poly3DCollection*.

`plot_trisurf`(*self*, **args*, *color=None*, *norm=None*, *vmin=None*, *vmax=None*, *lightsource=None*, ***kwargs*)

Plot a triangulated surface.

The (optional) triangulation can be specified in one of two ways; either:

```
plot_trisurf(triangulation, ...)
```

where triangulation is a *Triangulation* object, or:

```
plot_trisurf(X, Y, ...)
plot_trisurf(X, Y, triangles, ...)
plot_trisurf(X, Y, triangles=triangles, ...)
```

in which case a Triangulation object will be created. See *Triangulation* for a explanation of these possibilities.

The remaining arguments are:

```
plot_trisurf(..., Z)
```

where *Z* is the array of values to contour, one per point in the triangulation.

Parameters

X, Y, Z

[array-like] Data values as 1D arrays.

color

Color of the surface patches.

cmap

A colormap for the surface patches.

norm

[Normalize] An instance of Normalize to map values to colors.

vmin, vmax

[float, default: None] Minimum and maximum value to map.

shade

[bool, default: True] Whether to shade the facecolors. Shading is always disabled when *cmap* is specified.

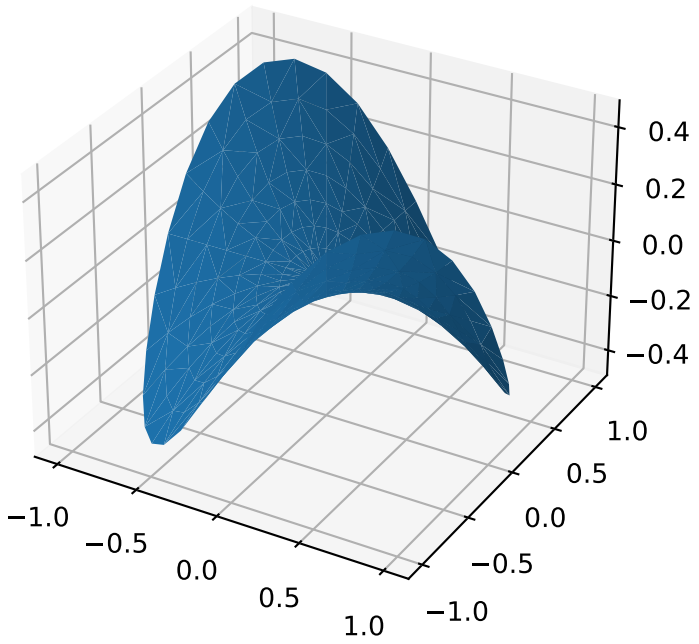
lightsource

[*LightSource*] The lightsource to use when *shade* is True.

****kwargs**

All other arguments are passed on to *Poly3DCollection*

Examples



New in version 1.2.0.

`plot_wireframe`(*self*, *X*, *Y*, *Z*, **args*, ***kwargs*)

Plot a 3D wireframe.

Note: The *rcount* and *ccount* kwargs, which both default to 50, determine the maximum number of samples used in each direction. If the input data is larger, it will be downsampled (by slicing) to these numbers of points.

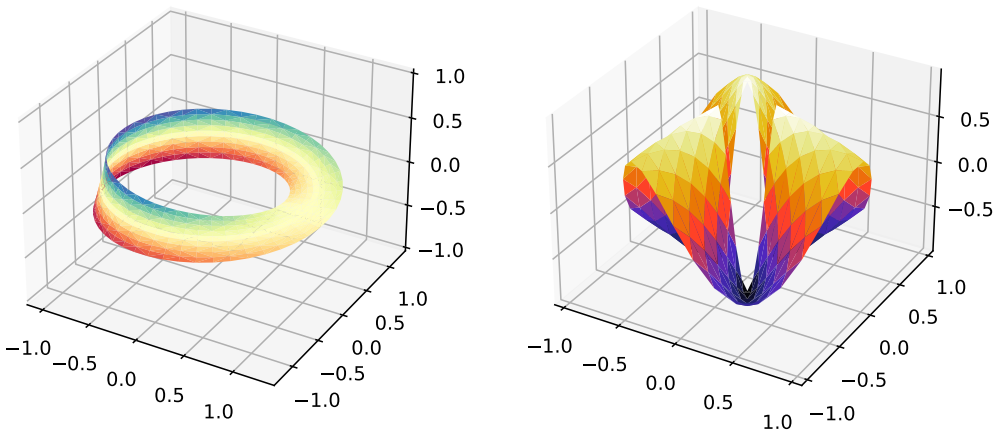
Parameters

X, Y, Z

[2d arrays] Data values.

rcount, ccount

[int] Maximum number of samples used in each direction. If the input data is larger, it will be downsampled (by slicing) to these numbers of points. Setting a count to zero causes the data to



be not sampled in the corresponding direction, producing a 3D line plot rather than a wireframe plot. Defaults to 50.

New in version 2.0.

rstride, cstride

[int] Downsampling stride in each direction. These arguments are mutually exclusive with *rcount* and *ccount*. If only one of *rstride* or *cstride* is set, the other defaults to 1. Setting a stride to zero causes the data to be not sampled in the corresponding direction, producing a 3D line plot rather than a wireframe plot.

'classic' mode uses a default of `rstride = cstride = 1` instead of the new default of `rcount = ccount = 50`.

****kwargs**

Other arguments are forwarded to *Line3DCollection*.

`quiver`(*X*, *Y*, *Z*, *U*, *V*, *W*, */*, *length=1*, *arrow_length_ratio=0.3*, *pivot='tail'*, *normalize=False*, ***kwargs*)

Plot a 3D field of arrows.

The arguments could be array-like or scalars, so long as they they can be broadcast together. The arguments can also be masked arrays. If an element in any of argument is masked, then that corresponding quiver element will not be plotted.

Parameters

X, Y, Z

[array-like] The x, y and z coordinates of the arrow locations (default is tail of arrow; see *pivot* kwarg).

U, V, W

[array-like] The x, y and z components of the arrow vectors.

length

[float, default: 1] The length of each quiver.

arrow_length_ratio

[float, default: 0.3] The ratio of the arrow head with respect to the quiver.

pivot

[{'tail', 'middle', 'tip'}, default: 'tail'] The part of the arrow that is at the grid point; the arrow rotates about this point, hence the name *pivot*.

normalize

[bool, default: False] Whether all arrows are normalized to have the same length, or keep the lengths defined by *u*, *v*, and *w*.

****kwargs**

Any additional keyword arguments are delegated to *LineCollection*

`quiver3D`(*X*, *Y*, *Z*, *U*, *V*, *W*, */*, *length=1*, *arrow_length_ratio=0.3*, *pivot='tail'*, *normalize=False*, ***kwargs*)

Plot a 3D field of arrows.

The arguments could be array-like or scalars, so long as they they can be broadcast together. The arguments can also be masked arrays. If an element in any of argument is masked, then that corresponding quiver element will not be plotted.

Parameters

X, Y, Z

[array-like] The x, y and z coordinates of the arrow locations (default is tail of arrow; see *pivot* kwarg).

U, V, W

[array-like] The x, y and z components of the arrow vectors.

length

[float, default: 1] The length of each quiver.

arrow_length_ratio

[float, default: 0.3] The ratio of the arrow head with respect to the quiver.

pivot

[{'tail', 'middle', 'tip'}, default: 'tail'] The part of the arrow that is at the grid point; the arrow rotates about this point, hence the name *pivot*.

normalize

[bool, default: False] Whether all arrows are normalized to have the same length, or keep the lengths defined by *u*, *v*, and *w*.

****kwargs**

Any additional keyword arguments are delegated to *LineCollection*

`scatter`(*self*, *xs*, *ys*, *zs=0*, *zdir='z'*, *s=20*, *c=None*, *depthshade=True*, **args*, ***kwargs*)

Create a scatter plot.

Parameters

xs, ys

[array-like] The data positions.

zs

[float or array-like, default: 0] The z-positions. Either an array of the same length as *xs* and *ys* or a single value to place all points in the same plane.

zdir

[{'x', 'y', 'z', '-x', '-y', '-z'}, default: 'z'] The axis direction for the *zs*. This is useful when plotting 2D data on a 3D Axes. The data must be passed as *xs*, *ys*. Setting *zdir* to 'y' then plots the data to the x-z-plane.

See also /gallery/mplot3d/2dcollections3d.

s

[float or array-like, default: 20] The marker size in points**2. Either an array of the same length as *xs* and *ys* or a single value to make all markers the same size.

c

[color, sequence, or sequence of colors, optional] The marker color. Possible values:

- A single color format string.
- A sequence of colors of length n.
- A sequence of n numbers to be mapped to colors using *cmap* and *norm*.
- A 2-D array in which the rows are RGB or RGBA.

For more details see the *c* argument of *scatter*.

depthshade

[bool, default: True] Whether to shade the scatter markers to give the appearance of depth. Each call to `scatter()` will perform its depthshading independently.

****kwargs**

All other arguments are passed on to *scatter*.

Returns

paths

[*PathCollection*]

`scatter3D`(*self*, *xs*, *ys*, *zs=0*, *zdir='z'*, *s=20*, *c=None*, *depthshade=True*, **args*, ***kwargs*)

Create a scatter plot.

Parameters

xs, ys

[array-like] The data positions.

zs

[float or array-like, default: 0] The z-positions. Either an array of the same length as *xs* and *ys* or a single value to place all points in the same plane.

zdir

[{'x', 'y', 'z', '-x', '-y', '-z'}, default: 'z'] The axis direction for the *zs*. This is useful when plotting 2D data on a 3D Axes. The data must be passed as *xs*, *ys*. Setting *zdir* to 'y' then plots the data to the x-z-plane.

See also /gallery/mplot3d/2dcollections3d.

s

[float or array-like, default: 20] The marker size in points**2. Either an array of the same length as *xs* and *ys* or a single value to make all markers the same size.

c

[color, sequence, or sequence of colors, optional] The marker color. Possible values:

- A single color format string.
- A sequence of colors of length n.
- A sequence of n numbers to be mapped to colors using *cmap* and *norm*.
- A 2-D array in which the rows are RGB or RGBA.

For more details see the *c* argument of *scatter*.

depthshade

[bool, default: True] Whether to shade the scatter markers to give the appearance of depth. Each call to `scatter()` will perform its depthshading independently.

****kwargs**

All other arguments are passed on to *scatter*.

Returns

paths

[*PathCollection*]

`set_anchor`(*self*, *anchor*, *share=False*)

Define the anchor location.

The actual drawing area (active position) of the Axes may be smaller than the Bbox (original position) when a fixed aspect is required. The anchor defines where the drawing area will be located within the available space.

Parameters

anchor

[2-tuple of floats or {'C', 'SW', 'S', 'SE', ...}] The anchor position may be either:

- a sequence (*cx*, *cy*). *cx* and *cy* may range from 0 to 1, where 0 is left or bottom and 1 is right or top.
- a string using cardinal directions as abbreviation:
  - 'C' for centered
  - 'S' (south) for bottom-center
  - 'SW' (south west) for bottom-left
  - etc.

Here is an overview of the possible positions:

'NW'	'N'	'NE'
'W'	'C'	'E'
'SW'	'S'	'SE'

share

[bool, default: False] If `True`, apply the settings to all shared Axes.

See also:

matplotlib.axes.Axes.set_aspect

for a description of aspect handling.

`set_aspect`(*self*, *aspect*, *adjustable=None*, *anchor=None*, *share=False*)
Set the aspect ratios.

Axes 3D does not current support any aspect but 'auto' which fills the axes with the data limits.

To simulate having equal aspect in data space, set the ratio of your data limits to match the value of *get_box_aspect*. To control box aspect ratios use *set_box_aspect*.

Parameters

aspect

[{'auto'}] Possible values:

value	description
'auto'	automatic; fill the position rectangle with data.

adjustable

[None] Currently ignored by Axes3D

If not *None*, this defines which parameter will be adjusted to meet the required aspect. See *set_adjustable* for further details.

anchor

[None or str or 2-tuple of float, optional] If not *None*, this defines where the Axes will be drawn if there is extra space due to aspect constraints. The most common way to to specify the anchor are abbreviations of cardinal directions:

value	description
'C'	centered
'SW'	lower left corner
'S'	middle of bottom edge
'SE'	lower right corner
etc.	

See *set_anchor* for further details.

share

[bool, default: False] If `True`, apply the settings to all shared Axes.

See also:

mpl_toolkits.mplot3d.axes3d.Axes3D.set_box_aspect

`set_autoscale_on`(*self*, *b*)
Set whether autoscaling is applied on plot commands

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

Parameters

b

[bool]

`set_autoscalez_on`(*self*, *b*)
Set whether autoscaling for the z-axis is applied on plot commands

New in version 1.1.0.

Parameters

b

[bool]

`set_axis_off`(*self*)
Turn the x- and y-axis off.

This affects the axis lines, ticks, ticklabels, grid and axis labels.

`set_axis_on`(*self*)
Turn the x- and y-axis on.

This affects the axis lines, ticks, ticklabels, grid and axis labels.

`set_box_aspect`(*self*, *aspect*, *, *zoom=1*)
Set the axes box aspect.

The box aspect is the ratio of height to width in display units for each face of the box when viewed perpendicular to that face. This is not to be confused with the data aspect (which for Axes3D is always 'auto'). The default ratios are 4:4:3 (x:y:z).

To simulate having equal aspect in data space, set the box aspect to match your data range in each dimension.

zoom controls the overall size of the Axes3D in the figure.

Parameters

aspect

[3-tuple of floats or None] Changes the physical dimensions of the Axes3D, such that the ratio of the axis lengths in display units is x:y:z.

If None, defaults to 4:4:3

zoom

[float] Control overall size of the Axes3D in the figure.

`set_frame_on`(*self*, *b*)
Set whether the 3D axes panels are drawn.

Parameters

b

[bool]

`set_proj_type`(*self*, *proj_type*)
Set the projection type.

Parameters

proj_type

[{'persp', 'ortho'}]

`set_title`(*self*, *label*, *fontdict=None*, *loc='center'*, ***kwargs*)
Set a title for the axes.

Set one of the three available axes titles. The available titles are positioned above the axes in the center, flush with the left edge, and flush with the right edge.

Parameters

label

[str] Text to use for the title

fontdict

[dict] A dictionary controlling the appearance of the title text, the default *fontdict* is:

```
{'fontsize': rcParams['axes.titlesize'],
 'fontweight': rcParams['axes.titleweight'],
 'color': rcParams['axes.titlecolor'],
 'verticalalignment': 'baseline',
 'horizontalalignment': loc}
```

loc

[{'center', 'left', 'right'}, default: `rcParams["axes.titlelocation"]` (default: `'center'`)] Which title to set.

y

[float, default: `rcParams["axes.titley"]` (default: `None`)] Vertical axes loation for the title (1.0 is the top). If None (the default), y is determined automatically to avoid decorators on the axes.

pad

[float, default: `rcParams["axes.titlepad"]` (default: `6.0`)] The offset of the title from the top of the axes, in points.

Returns

Text

The matplotlib text instance representing the title

Other Parameters

****kwargs**

[*Text* properties] Other keyword arguments are text properties, see *Text* for a list of valid text properties.

`set_top_view`(*self*)

`set_xlim`(*self*, *left=None*, *right=None*, *emit=True*, *auto=False*, *, *xmin=None*, *xmax=None*)

Alias for *set_xlim3d*.

`set_xlim3d`(*self*, *left=None*, *right=None*, *emit=True*, *auto=False*, *, *xmin=None*, *xmax=None*)

Set 3D x limits.

See *matplotlib.axes.Axes.set_xlim()* for full documentation.

set_xscale(*self*, *value*, ***kwargs*)
Set the x-axis scale.

Parameters

value

[{"linear"}] The axis scale type to apply. 3D axes currently only support linear scales; other scales yield nonsensical results.

****kwargs**

Keyword arguments are nominally forwarded to the scale class, but none of them is applicable for linear scales.

set_ylim(*self*, *bottom=None*, *top=None*, *emit=True*, *auto=False*, *, *ymin=None*, *ymax=None*)
Alias for *set_ylim3d*.

set_ylim3d(*self*, *bottom=None*, *top=None*, *emit=True*, *auto=False*, *, *ymin=None*, *ymax=None*)
Set 3D y limits.

See *matplotlib.axes.Axes.set_ylim()* for full documentation.

set_yscale(*self*, *value*, ***kwargs*)
Set the y-axis scale.

Parameters

value

[{"linear"}] The axis scale type to apply. 3D axes currently only support linear scales; other scales yield nonsensical results.

****kwargs**

Keyword arguments are nominally forwarded to the scale class, but none of them is applicable for linear scales.

set_zbound(*self*, *lower=None*, *upper=None*)
Set the lower and upper numerical bounds of the z-axis.

This method will honor axes inversion regardless of parameter order. It will not change the autoscaling setting (*get_autoscalez_on()*).

New in version 1.1.0.

set_zlabel(*self*, *zlabel*, *fontdict=None*, *labelpad=None*, ***kwargs*)
Set zlabel. See doc for *set_ylabel* for description.

set_zlim(*self*, *bottom=None*, *top=None*, *emit=True*, *auto=False*, *, *zmin=None*, *zmax=None*)
Alias for *set_zlim3d*.

`set_zlim3d`(*self*, *bottom=None*, *top=None*, *emit=True*, *auto=False*, ***, *zmin=None*, *zmax=None*)
Set 3D z limits.

See *matplotlib.axes.Axes.set_ylim()* for full documentation

`set_zmargin`(*self*, *m*)
Set padding of Z data limits prior to autoscaling.

m times the data interval will be added to each end of that interval before it is used in autoscaling.

accepts: float in range 0 to 1

New in version 1.1.0.

`set_zscale`(*self*, *value*, ***kwargs*)
Set the z-axis scale.

Parameters

value

[{"linear"}] The axis scale type to apply. 3D axes currently only support linear scales; other scales yield nonsensical results.

****kwargs**

Keyword arguments are nominally forwarded to the scale class, but none of them is applicable for linear scales.

`set_zticklabels`(*self*, *labels*, ***, *fontdict=None*, *minor=False*, ***kwargs*)
Set the zaxis' labels with list of string labels.

Warning: This method should only be used after fixing the tick positions using *Axes3D.set_zticks*. Otherwise, the labels may end up in unexpected positions.

Parameters

labels

[list of str] The label texts.

fontdict

[dict, optional] A dictionary controlling the appearance of the ticklabels. The default *fontdict* is:

```
{'fontsize': rcParams['axes.titlesize'],
 'fontweight': rcParams['axes.titleweight'],
 'verticalalignment': 'baseline',
 'horizontalalignment': loc}
```

minor

[bool, default: False] Whether to set the minor ticklabels rather than the major ones.

Returns

list of *Text*

The labels.

Other Parameters

****kwargs**

[*Text* properties.]

`set_zticks`(*self*, *ticks*, *, *minor=False*)
Set the zaxis' tick locations.

Parameters

ticks

[list of floats] List of tick locations.

minor

[bool, default: False] If `False`, set the major ticks; if `True`, the minor ticks.

`text`(*self*, *x*, *y*, *z*, *s*, *zdir=None*, ***kwargs*)
Add text to the plot. kwargs will be passed on to Axes.text, except for the *zdir* keyword, which sets the direction to be used as the z direction.

`text2D`(*self*, *x*, *y*, *s*, *fontdict=None*, ***kwargs*)
Add text to the axes.

Add the text *s* to the axes at location *x*, *y* in data coordinates.

Parameters

x, y

[float] The position to place the text. By default, this is in data coordinates. The coordinate system can be changed using the *transform* parameter.

s

[str] The text.

fontdict

[dict, default: None] A dictionary to override the default text properties. If fontdict is None, the defaults are determined by *rcParams*.

Returns

Text

The created *Text* instance.

Other Parameters

****kwargs**

[*Text* properties.] Other miscellaneous text parameters.

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array an
alpha	float or None
animated	bool
backgroundcolor	color
bbox	dict with properties for *patches.FancyBboxPatch*
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color or c	color
contains	unknown
figure	*Figure*
fontfamily or family	{FONTNAME, 'serif', 'sans-serif', 'cursive', 'fantasy', '
fontproperties or font or font_properties	*font_manager.FontProperties* or `str` or `pathlib.Path`
fontsize or size	float or {'xx-small', 'x-small', 'small', 'medium', 'large',
fontstretch or stretch	{a numeric value in range 0-1000, 'ultra-condensed',
fontstyle or style	{'normal', 'italic', 'oblique'}
fontvariant or variant	{'normal', 'small-caps'}
fontweight or weight	{a numeric value in range 0-1000, 'ultralight', 'light',
gid	str
horizontalalignment or ha	{'center', 'right', 'left'}
in_layout	bool
label	object
linespacing	float (multiple of font size)
multialignment or ma	{'left', 'right', 'center'}
path_effects	*AbstractPathEffect*
picker	None or bool or callable
position	(float, float)
rasterized	bool or None

Tab

Property	Description
rotation	float or {'vertical', 'horizontal'}
rotation_mode	{None, 'default', 'anchor'}
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
text	object
transform	*Transform*
url	str
usetex	bool or None
verticalalignment or va	{'center', 'top', 'bottom', 'baseline', 'center_baseline'}
visible	bool
wrap	bool
x	float
y	float
zorder	float

Examples

Individual keyword arguments can be used to override any given parameter:

```
>>> text(x, y, s, fontsize=12)
```

The default transform specifies that text is in data coords, alternatively, you can specify text in axis coords ((0, 0) is lower-left and (1, 1) is upper-right). The example below places text in the center of the axes:

```
>>> text(0.5, 0.5, 'matplotlib', horizontalalignment='center',
...      verticalalignment='center', transform=ax.transAxes)
```

You can put a rectangular box around the text instance (e.g., to set a background color) by using the keyword *bbox*. *bbox* is a dictionary of *Rectangle* properties. For example:

```
>>> text(x, y, s, bbox=dict(facecolor='red', alpha=0.5))
```

`text3D`(*self*, *x*, *y*, *z*, *s*, *zdir=None*, ***kwargs*)
Add text to the plot. kwargs will be passed on to Axes.text, except for the *zdir* keyword, which sets the direction to be used as the z direction.

`tick_params`(*self*, *axis='both'*, ***kwargs*)
Convenience method for changing the appearance of ticks and tick labels.

See *matplotlib.axes.Axes.tick_params()* for more complete documentation.

The only difference is that setting *axis* to 'both' will mean that the settings are applied to all three axes. Also, the *axis* parameter also accepts a value of 'z', which would mean to apply to only the z-axis.

Also, because of how Axes3D objects are drawn very differently from regular 2D axes, some of these settings may have ambiguous meaning. For simplicity, the 'z' axis will accept settings as if it was like the 'y' axis.

Note: Axes3D currently ignores some of these settings.

New in version 1.1.0.

`tricontour`(*self*, **args*, *extend3d=False*, *stride=5*, *zdir='z'*, *offset=None*, ***kwargs*)

Create a 3D contour plot.

Changed in version 1.3.0: Added support for custom triangulations

---

Note: This method currently produces incorrect output due to a longstanding bug in 3D PolyCollection rendering.

Parameters

X, Y, Z

[array-like] Input data.

extend3d

[bool, default: False] Whether to extend contour in 3D.

stride

[int] Step size for extending contour.

zdir

[{'x', 'y', 'z'}, default: 'z'] The direction to use.

offset

[float, optional] If specified, plot a projection of the contour lines at this position in a plane normal to zdir.

***args, **kwargs**

Other arguments are forwarded to *matplotlib.axes.Axes.tricontour*.

Returns

matplotlib.tri.tricontour.TriContourSet

`tricontourf`(*self*, **args*, *zdir='z'*, *offset=None*, ***kwargs*)

Create a 3D filled contour plot.

Note: This method currently produces incorrect output due to a longstanding bug in 3D PolyCollection rendering.

Parameters

X, Y, Z

[array-like] Input data.

zdir

[{'x', 'y', 'z'}, default: 'z'] The direction to use.

offset

[float, optional] If specified, plot a projection of the contour lines at this position in a plane normal to zdir.

***args, **kwargs**

Other arguments are forwarded to *matplotlib.axes.Axes.tricontourf*.

Returns

matplotlib.tri.tricontour.TriContourSet

Notes

New in version 1.1.0: The *zdir* and *offset* parameters.

Changed in version 1.3.0: Added support for custom triangulations

`tunit_cube`(*self*, *vals=None*, *M=None*)

`tunit_edges`(*self*, *vals=None*, *M=None*)

`unit_cube`(*self*, *vals=None*)

`update_datalim`(*self*, *xys*, ***kwargs*)

Extend the `dataLim` Bbox to include the given points.

If no data is set currently, the Bbox will ignore its limits and set the bound to be the bounds of the xydata (*xys*). Otherwise, it will compute the bounds of the union of its current data and the data in *xys*.

Parameters

xys

[2D array-like] The points to include in the data limits Bbox. This can be either a list of (x, y) tuples or a Nx2 array.

updatex, updatey

[bool, default: True] Whether to update the x/y limits.

`view_init`(*self*, *elev=None*, *azim=None*)
Set the elevation and azimuth of the axes in degrees (not radians).

This can be used to rotate the axes programmatically.

'elev' stores the elevation angle in the z plane (in degrees). 'azim' stores the azimuth angle in the (x, y) plane (in degrees).

if 'elev' or 'azim' are None (default), then the initial value is used which was specified in the *Axes3D* constructor.

`voxels`([*x*, *y*, *z*], */*, *filled*, *facecolors=None*, *edgecolors=None*, ***kwargs*)
Plot a set of filled voxels

All voxels are plotted as 1x1x1 cubes on the axis, with `filled[0, 0, 0]` placed with its lower corner at the origin. Occluded faces are not plotted.

New in version 2.1.

Parameters

filled

[3D np.array of bool] A 3d array of values, with truthy values indicating which voxels to fill

x, y, z

[3D np.array, optional] The coordinates of the corners of the voxels. This should broadcast to a shape one larger in every dimension than the shape of *filled*. These can be used to plot non-cubic voxels.

If not specified, defaults to increasing integers along each axis, like those returned by `indices()`. As indicated by the `/` in the function signature, these arguments can only be passed positionally.

facecolors, edgecolors

[array-like, optional] The color to draw the faces and edges of the voxels. Can only be passed as keyword arguments. This parameter can be:

- A single color value, to color all voxels the same color. This can be either a string, or a 1D rgb/rgba array
- `None`, the default, to use a single color for the faces, and the style default for the edges.
- A 3D ndarray of color names, with each item the color for the corresponding voxel. The size must match the voxels.

- A 4D ndarray of rgb/rgba data, with the components along the last axis.

shade

[bool, default: True] Whether to shade the facecolors. Shading is always disabled when *cmap* is specified.

New in version 3.1.

lightsource

[*LightSource*] The lightsource to use when *shade* is True.

New in version 3.1.

****kwargs**

Additional keyword arguments to pass onto *Poly3DCollection*.

Returns

faces

[dict] A dictionary indexed by coordinate, where `faces[i, j, k]` is a *Poly3DCollection* of the faces drawn for the voxel `filled[i, j, k]`. If no faces were drawn for a given voxel, either because it was not asked to be drawn, or it is fully occluded, then `(i, j, k) not in faces`.

Examples

`property w_xaxis`

`property w_yaxis`

`property w_zaxis`

`zaxis_date`(*self*, *tz=None*)

Sets up axis ticks and labels to treat data along the zaxis as dates.

Parameters

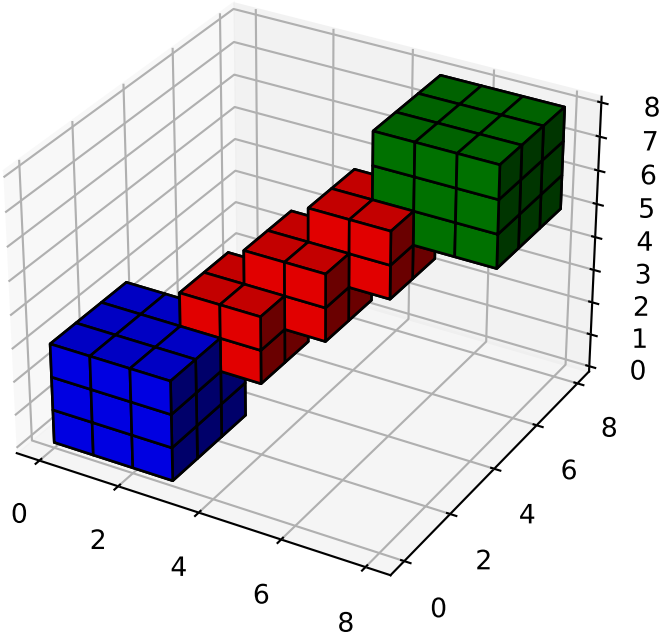
tz

[str or `datetime.tzinfo`, default: `rcParams["timezone"]` (default: `'UTC'`)] The timezone used to create date labels.

`zaxis_inverted`(*self*)

Returns True if the z-axis is inverted.

New in version 1.1.0.



Examples using `mpl_toolkits.mplot3d.axes3d.Axes3D`

19.2.2 axis3d

Note: See `mpl_toolkits.mplot3d.axis3d._axinfo` for a dictionary containing constants that may be modified for controlling the look and feel of mplot3d axes (e.g., label spacing, font colors and panel colors). Historically, axis3d has suffered from having hard-coded constants that precluded user adjustments, and this dictionary was implemented in version 1.1 as a stop-gap measure.

axis3d.Axis(adir, v_intervalx, d_intervalx, ...)	An Axis class for the 3D plots.

mpl_toolkits.mplot3d.axis3d.Axis

class mpl_toolkits.mplot3d.axis3d.**Axis**(*adir*, *v_intervalx*, *d_intervalx*, *axes*, **args*, *rotate_label=None*, ***kwargs*)

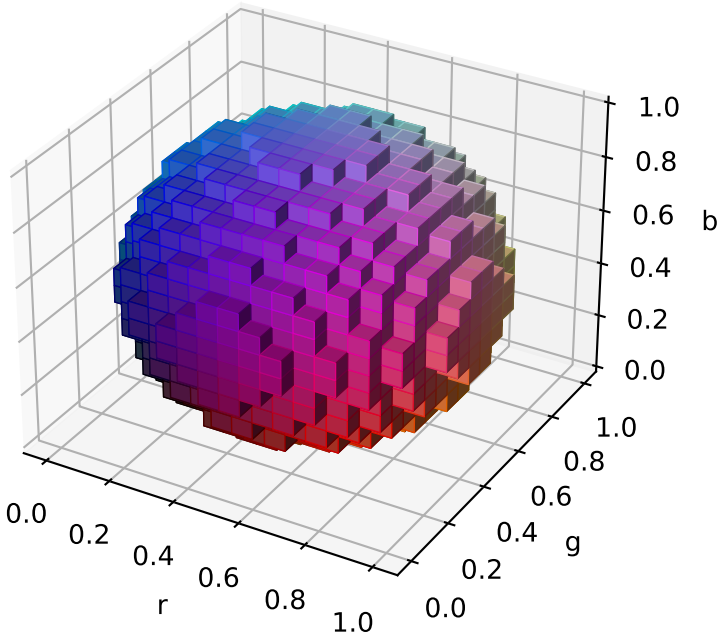
Bases: *matplotlib.axis.XAxis*

An Axis class for the 3D plots.

Parameters

axes

[*matplotlib.axes.Axes*] The *Axes* to which the created Axis be-



`__module__ = 'mpl_toolkits.mplot3d.axis3d'`

`property d_interval`

`draw`(*self*, *renderer*)

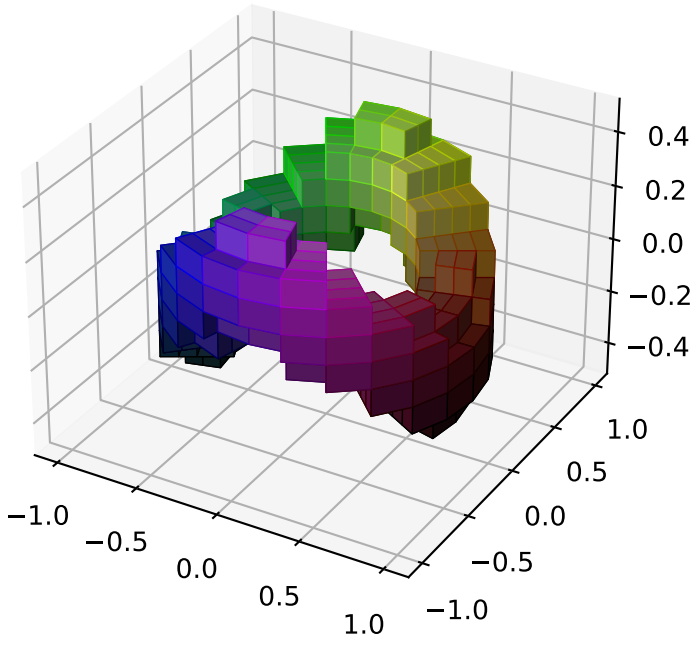
Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

Parameters

renderer

[*RendererBase* subclass.]



Notes

This method is overridden in the Artist subclasses.

`draw_pane`(*self*, *renderer*)

`get_major_ticks`(*self*, *numticks=None*)
Return the list of major *Tick*s.

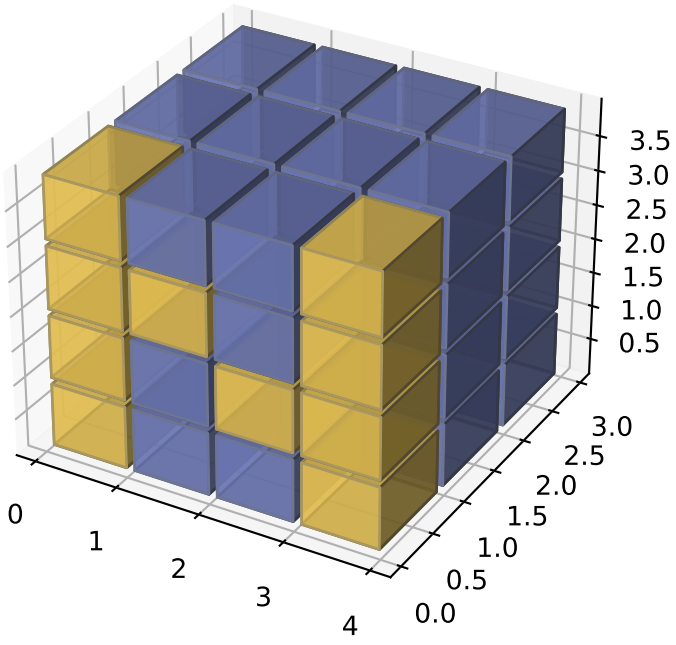
`get_minor_ticks`(*self*, *numticks=None*)
Return the list of minor *Tick*s.

`get_rotate_label`(*self*, *text*)

`get_tightbbox`(*self*, *renderer*, *, *for_layout_only=False*)
Return a bounding box that encloses the axis. It only accounts tick labels, axis label, and offsetText.

If *for_layout_only* is True, then the width of the label (if this is an x-axis) or the height of the label (if this is a y-axis) is collapsed to near zero. This allows tight/constrained_layout to ignore too-long labels when doing their layout.

`init3d`(*self*)



`set_pane_color`(*self*, *color*)
Set pane color to a RGBA tuple.

`set_pane_pos`(*self*, *xys*)

`set_rotate_label`(*self*, *val*)
Whether to rotate the axis label: True, False or None. If set to None the label will be rotated if longer than 4 chars.

`property v_interval`

Examples using `mpl_toolkits.mplot3d.axis3d.Axis`

19.2.3 art3d

art3d.Line3D(xs, ys, zs, *args, **kwargs)	3D line object.
art3d.Line3DCollection(segments[, ...])	A collection of 3D lines.
art3d.Patch3D(*args[, zs, zdir])	3D patch object.
art3d.Patch3DCollection(*args[, zs, zdir, ...])	A collection of 3D patches.

continues on next page

Table 50 – continued from previous page

art3d.Path3DCollection(*args[, zs, zdir, ...])	A collection of 3D paths.
art3d.PathPatch3D(path, *[, zs, zdir])	3D PathPatch object.
art3d.Poly3DCollection(verts, *args[, zsort])	A collection of 3D polygons.
art3d.Text3D([x, y, z, text, zdir])	Text object with 3D position and direction.
art3d.get_dir_vector(zdir)	Return a direction vector.
art3d.juggle_axes(xs, ys, zs, zdir)	Reorder coordinates so that 2D xs, ys can be plotted in the plane orthogonal to zdir.
art3d.line_2d_to_3d(line[, zs, zdir])	Convert a 2D line to 3D.
art3d.line_collection_2d_to_3d(col[, zs, zdir])	Convert a LineCollection to a Line3DCollection object.
art3d.patch_2d_to_3d(patch[, z, zdir])	Convert a Patch to a Patch3D object.
art3d.patch_collection_2d_to_3d(col[, zs, ...])	Convert a *PatchCollection* into a `Patch3DCollection` object (or a *PathCollection* into a `Path3DCollection` object).
art3d.pathpatch_2d_to_3d(pathpatch[, z, zdir])	Convert a PathPatch to a PathPatch3D object.
art3d.poly_collection_2d_to_3d(col[, zs, zdir])	Convert a PolyCollection to a Poly3DCollection object.
art3d.rotate_axes(xs, ys, zs, zdir)	Reorder coordinates so that the axes are rotated with zdir along the original z axis.
art3d.text_2d_to_3d(obj[, z, zdir])	Convert a Text to a Text3D object.

mpl_toolkits.mplot3d.art3d.Line3D

`class mpl_toolkits.mplot3d.art3d.Line3D(`*xs*, *ys*, *zs*, **args*, ***kwargs*`)`

Bases: *matplotlib.lines.Line2D*

3D line object.

Keyword arguments are passed onto *Line2D()*.

`__init__(`*self*, *xs*, *ys*, *zs*, **args*, ***kwargs*`)`

Keyword arguments are passed onto *Line2D()*.

`__module__ = 'mpl_toolkits.mplot3d.art3d'`

`draw(`*self*, *renderer*`)`

Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

Parameters

renderer

[*RendererBase* subclass.]

Notes

This method is overridden in the Artist subclasses.

`get_data_3d`(*self*)
Get the current data

Returns

verts3d

[length-3 tuple or array-like] The current data as a tuple or array-like.

`set_3d_properties`(*self*, *zs=0*, *zdir='z'*)

`set_data_3d`(*self*, **args*)
Set the x, y and z data

Parameters

x

[array-like] The x-data to be plotted.

y

[array-like] The y-data to be plotted.

z

[array-like] The z-data to be plotted.

Notes

Accepts x, y, z arguments or a single array-like (x, y, z)

Examples using `mpl_toolkits.mplot3d.art3d.Line3D`

mpl_toolkits.mplot3d.art3d.Line3DCollection

class mpl_toolkits.mplot3d.art3d.**Line3DCollection**(*segments*, *linewidths=None*, *colors=None*, *antialiaseds=None*, *linestyles='solid'*, *offsets=None*, *transOffset=None*, *norm=None*, *cmap=None*, *pickradius=5*, *zorder=2*, *facecolors='none'*, ***kwargs*)

Bases: *matplotlib.collections.LineCollection*

A collection of 3D lines.

Parameters

segments: list of array-like

A sequence of (*line0*, *line1*, *line2*), where:

```
linen = (x0, y0), (x1, y1), ... (xm, ym)
```

or the equivalent numpy array with two columns. Each line can have a different number of segments.

linewidths

[float or list of float, default: `rcParams["lines.linewidth"]` (default: `1.5`)] The width of each line in points.

colors

[color or list of color, default: `rcParams["lines.color"]` (default: `'C0'`)] A sequence of RGBA tuples (e.g., arbitrary color strings, etc, not allowed).

antialiaseds

[bool or list of bool, default: `rcParams["lines.antialiased"]` (default: `True`)] Whether to use antialiasing for each line.

zorder

[int, default: 2] zorder of the lines once drawn.

facecolors

[color or list of color, default: 'none'] The facecolors of the LineCollection. Setting to a value other than 'none' will lead to each line being "filled in" as if there was an implicit line segment

joining the last and first points of that line back around to each other. In order to manually specify what should count as the "interior" of each line, please use *PathCollection* instead, where the "interior" can be specified by appropriate usage of *CLOSEPOLY*.

****kwargs**

Forwareded to *Collection*.

`__module__ = 'mpl_toolkits.mplot3d.art3d'`

`do_3d_projection`(*self*, *renderer*)
Project the points according to renderer matrix.

`draw`(*self*, *renderer*, *project=False*)
Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

Parameters

renderer

[*RendererBase* subclass.]

Notes

This method is overridden in the Artist subclasses.

`set_segments`(*self*, *segments*)
Set 3D segments.

`set_sort_zpos`(*self*, *val*)
Set the position to use for z-sorting.

Examples using `mpl_toolkits.mplot3d.art3d.Line3DCollection`

mpl_toolkits.mplot3d.art3d.Patch3D

`class mpl_toolkits.mplot3d.art3d.Patch3D`(**args*, *zs=()*, *zdir='z'*, ***kwargs*)
Bases: *matplotlib.patches.Patch*

3D patch object.

The following kwarg properties are supported

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and retur

Table 51 – continued from previous page

Property	Description
alpha	float or None
animated	bool
antialiased or aa	unknown
capstyle	{'butt', 'round', 'projecting'}
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color	color
contains	unknown
edgecolor or ec	color or None or 'auto'
facecolor or fc	color or None
figure	*Figure*
fill	bool
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float or None
path_effects	*AbstractPathEffect*
picker	None or bool or callable
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
visible	bool
zorder	float

__init__(*self*, **args*, *zs=()*, *zdir='z'*, ***kwargs*)

The following kwarg properties are supported

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and retur
alpha	float or None
animated	bool
antialiased or aa	unknown
capstyle	{'butt', 'round', 'projecting'}
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color	color

Table 52 – continued from previous page

Property	Description
contains	unknown
edgecolor or ec	color or None or 'auto'
facecolor or fc	color or None
figure	*Figure*
fill	bool
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float or None
path_effects	*AbstractPathEffect*
picker	None or bool or callable
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
visible	bool
zorder	float

`__module__ = 'mpl_toolkits.mplot3d.art3d'`

`do_3d_projection`(*self*, *renderer*)

`get_facecolor`(*self*)
Return the face color.

`get_path`(*self*)
Return the path of this patch.

`set_3d_properties`(*self*, *verts*, *zs=0*, *zdir='z'*)

Examples using `mpl_toolkits.mplot3d.art3d.Patch3D`

mpl_toolkits.mplot3d.art3d.Patch3DCollection

`class mpl_toolkits.mplot3d.art3d.Patch3DCollection`(**args*, *zs=0*, *zdir='z'*, *depthshade=True*, ***kwargs*)

Bases: *matplotlib.collections.PatchCollection*

A collection of 3D patches.

Create a collection of flat 3D patches with its normal vector pointed in *zdir* direction, and located at *zs* on the *zdir* axis. 'zs' can be a scalar or an array-like of

the same length as the number of patches in the collection.

Constructor arguments are the same as for *PatchCollection*. In addition, keywords *zs=0* and *zdir='z'* are available.

Also, the keyword argument "depthshade" is available to indicate whether or not to shade the patches in order to give the appearance of depth (default is *True*). This is typically desired in scatter plots.

`__init__`(*self*, **args*, *zs=0*, *zdir='z'*, *depthshade=True*, ***kwargs*)

Create a collection of flat 3D patches with its normal vector pointed in *zdir* direction, and located at *zs* on the *zdir* axis. 'zs' can be a scalar or an array-like of the same length as the number of patches in the collection.

Constructor arguments are the same as for *PatchCollection*. In addition, keywords *zs=0* and *zdir='z'* are available.

Also, the keyword argument "depthshade" is available to indicate whether or not to shade the patches in order to give the appearance of depth (default is *True*). This is typically desired in scatter plots.

`__module__ = 'mpl_toolkits.mplot3d.art3d'`

`do_3d_projection`(*self*, *renderer*)

`set_3d_properties`(*self*, *zs*, *zdir*)

`set_sort_zpos`(*self*, *val*)

Set the position to use for z-sorting.

Examples using `mpl_toolkits.mplot3d.art3d.Patch3DCollection`

mpl_toolkits.mplot3d.art3d.Path3DCollection

`class mpl_toolkits.mplot3d.art3d.Path3DCollection`(**args*, *zs=0*, *zdir='z'*, *depthshade=True*, ***kwargs*)

Bases: *matplotlib.collections.PathCollection*

A collection of 3D paths.

Create a collection of flat 3D paths with its normal vector pointed in *zdir* direction, and located at *zs* on the *zdir* axis. 'zs' can be a scalar or an array-like of the same length as the number of paths in the collection.

Constructor arguments are the same as for *PathCollection*. In addition, keywords *zs=0* and *zdir='z'* are available.

Also, the keyword argument "depthshade" is available to indicate whether or not to shade the patches in order to give the appearance of depth (default is *True*). This is typically desired in scatter plots.

`__init__`(*self*, **args*, *zs=0*, *zdir='z'*, *depthshade=True*, ***kwargs*)
Create a collection of flat 3D paths with its normal vector pointed in *zdir* direction, and located at *zs* on the *zdir* axis. 'zs' can be a scalar or an array-like of the same length as the number of paths in the collection.

Constructor arguments are the same as for *PathCollection*. In addition, keywords *zs=0* and *zdir='z'* are available.

Also, the keyword argument "depthshade" is available to indicate whether or not to shade the patches in order to give the appearance of depth (default is *True*). This is typically desired in scatter plots.

`__module__ = 'mpl_toolkits.mplot3d.art3d'`

`do_3d_projection`(*self*, *renderer*)

`set_3d_properties`(*self*, *zs*, *zdir*)

`set_sort_zpos`(*self*, *val*)
Set the position to use for z-sorting.

Examples using `mpl_toolkits.mplot3d.art3d.Path3DCollection`

mpl_toolkits.mplot3d.art3d.PathPatch3D

`class mpl_toolkits.mplot3d.art3d.PathPatch3D`(*path*, ***, *zs=()*, *zdir='z'*, ***kwargs*)
Bases: *mpl_toolkits.mplot3d.art3d.Patch3D*

3D PathPatch object.

The following kwarg properties are supported

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and retur
alpha	float or None
animated	bool
antialiased or aa	unknown
capstyle	{'butt', 'round', 'projecting'}
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color	color
contains	unknown
edgecolor or ec	color or None or 'auto'
facecolor or fc	color or None
figure	*Figure*
fill	bool
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}

Table 53 – continued from previous page

Property	Description
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float or None
path_effects	*AbstractPathEffect*
picker	None or bool or callable
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
visible	bool
zorder	float

`__init__`(*self*, *path*, *, *zs=()*, *zdir='z'*, ***kwargs*)

The following kwarg properties are supported

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and retur
alpha	float or None
animated	bool
antialiased or aa	unknown
capstyle	{'butt', 'round', 'projecting'}
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color	color
contains	unknown
edgecolor or ec	color or None or 'auto'
facecolor or fc	color or None
figure	*Figure*
fill	bool
gid	str
hatch	{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
in_layout	bool
joinstyle	{'miter', 'round', 'bevel'}
label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float or None
path_effects	*AbstractPathEffect*
picker	None or bool or callable
rasterized	bool or None

Table 54 – continued from previous page

Property	Description
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
transform	*Transform*
url	str
visible	bool
zorder	float

__module__ = 'mpl_toolkits.mplot3d.art3d'

do_3d_projection(*self*, *renderer*)

set_3d_properties(*self*, *path*, *zs=0*, *zdir='z'*)

Examples using mpl_toolkits.mplot3d.art3d.PathPatch3D

- sphx_glr_gallery_mplot3d_pathpatch3d.py

mpl_toolkits.mplot3d.art3d.Poly3DCollection

class mpl_toolkits.mplot3d.art3d.Poly3DCollection(*verts*, **args*, *zsort='average'*, ***kwargs*)

Bases: *matplotlib.collections.PolyCollection*

A collection of 3D polygons.

Note: Filling of 3D polygons

There is no simple definition of the enclosed surface of a 3D polygon unless the polygon is planar.

In practice, Matplotlib fills the 2D projection of the polygon. This gives a correct filling appearance only for planar polygons. For all other polygons, you'll find orientations in which the edges of the polygon intersect in the projection. This will lead to an incorrect visualization of the 3D area.

If you need filled areas, it is recommended to create them via *plot_trisurf*, which creates a triangulation and thus generates consistent surfaces.

Parameters

verts

[list of array-like Nx3] Each element describes a polygon as a sequence of `N_i` points `(x, y, z)`.

zsort

[{'average', 'min', 'max'}, default: 'average'] The calculation method for the z-order. See *set_zsort* for details.

***args, **kwargs**

All other parameters are forwarded to *PolyCollection*.

Notes

Note that this class does a bit of magic with the _facecolors and _edgecolors properties.

`__init__`(*self*, *verts*, **args*, *zsort='average'*, ***kwargs*)

Parameters

verts

[list of array-like Nx3] Each element describes a polygon as a sequence of `N_i` points `(x, y, z)`.

zsort

[{'average', 'min', 'max'}, default: 'average'] The calculation method for the z-order. See *set_zsort* for details.

***args, **kwargs**

All other parameters are forwarded to *PolyCollection*.

Notes

Note that this class does a bit of magic with the _facecolors and _edgecolors properties.

`__module__ = 'mpl_toolkits.mplot3d.art3d'`

`do_3d_projection`(*self*, *renderer*)
Perform the 3D projection for this object.

`get_edgecolor`(*self*)

`get_facecolor`(*self*)

`get_vector`(*self*, *segments3d*)
Optimize points for projection.

`set_3d_properties`(*self*)

`set_alpha`(*self*, *alpha*)
Set the alpha value used for blending - not supported on all backends.

Parameters

alpha

[float or None]

`set_edgecolor`(*self*, *colors*)
Set the edgecolor(s) of the collection.

Parameters

c

[color or list of colors or 'face'] The collection edgecolor(s). If a sequence, the patches cycle through it. If 'face', match the facecolor.

`set_facecolor`(*self*, *colors*)
Set the facecolor(s) of the collection. *c* can be a color (all patches have same color), or a sequence of colors; if it is a sequence the patches will cycle through the sequence.

If *c* is 'none', the patch will not be filled.

Parameters

c

[color or list of colors]

`set_sort_zpos`(*self*, *val*)
Set the position to use for z-sorting.

`set_verts`(*self*, *verts*, *closed=True*)
Set 3D vertices.

`set_verts_and_codes`(*self*, *verts*, *codes*)
Set 3D vertices with path codes.

`set_zsort`(*self*, *zsort*)
Set the calculation method for the z-order.

Parameters

zsort

[{'average', 'min', 'max'}] The function applied on the z-coordinates of the vertices in the viewer's coordinate system, to determine the z-order.

Examples using `mpl_toolkits.mplot3d.art3d.Poly3DCollection`

- sphx_glr_gallery_pyplots_whats_new_1_subplot3d.py
- sphx_glr_gallery_frontpage_3D.py
- sphx_glr_gallery_mplot3d_custom_shaded_3d_surface.py
- sphx_glr_gallery_mplot3d_mixed_subplots.py
- sphx_glr_gallery_mplot3d_polys3d.py
- sphx_glr_gallery_mplot3d_subplot3d.py
- sphx_glr_gallery_mplot3d_surface3d.py
- sphx_glr_gallery_mplot3d_surface3d_3.py

mpl_toolkits.mplot3d.art3d.Text3D

class mpl_toolkits.mplot3d.art3d.**Text3D**(*x=0*, *y=0*, *z=0*, *text=''*, *zdir='z'*, ***kwargs*)

Bases: *matplotlib.text.Text*

Text object with 3D position and direction.

Parameters

x, y, z

The position of the text.

text

[str] The text string to display.

zdir

[{'x', 'y', 'z', None, 3-tuple}] The direction of the text. See *get_dir_vector* for a description of the values.

Other Parameters

****kwargs**

All other parameters are passed on to *Text*.

Create a *Text* instance at *x*, *y* with string *text*.

Valid keyword arguments are:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array an

Tab

Property	Description
alpha	float or None
animated	bool
backgroundcolor	color
bbox	dict with properties for *patches.FancyBboxPatch*
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color or c	color
contains	unknown
figure	*Figure*
fontfamily or family	{FONTNAME, 'serif', 'sans-serif', 'cursive', 'fantasy', '
fontproperties or font or font_properties	*font_manager.FontProperties* or `str` or `pathlib.Path`
fontsize or size	float or {'xx-small', 'x-small', 'small', 'medium', 'large',
fontstretch or stretch	{a numeric value in range 0-1000, 'ultra-condensed',
fontstyle or style	{'normal', 'italic', 'oblique'}
fontvariant or variant	{'normal', 'small-caps'}
fontweight or weight	{a numeric value in range 0-1000, 'ultralight', 'light',
gid	str
horizontalalignment or ha	{'center', 'right', 'left'}
in_layout	bool
label	object
linespacing	float (multiple of font size)
multialignment or ma	{'left', 'right', 'center'}
path_effects	*AbstractPathEffect*
picker	None or bool or callable
position	(float, float)
rasterized	bool or None
rotation	float or {'vertical', 'horizontal'}
rotation_mode	{None, 'default', 'anchor'}
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
text	object
transform	*Transform*
url	str
usetex	bool or None
verticalalignment or va	{'center', 'top', 'bottom', 'baseline', 'center_baseline'}
visible	bool
wrap	bool
x	float
y	float
zorder	float

__init__(*self*, *x=0*, *y=0*, *z=0*, *text=''*, *zdir='z'*, ***kwargs*)

Create a *Text* instance at *x*, *y* with string *text*.

Valid keyword arguments are:

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array an
alpha	float or None
animated	bool
backgroundcolor	color
bbox	dict with properties for *patches.FancyBboxPatch*
clip_box	*Bbox*
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color or c	color
contains	unknown
figure	*Figure*
fontfamily or family	{FONTNAME, 'serif', 'sans-serif', 'cursive', 'fantasy', '
fontproperties or font or font_properties	*font_manager.FontProperties* or `str` or `pathlib.Path`
fontsize or size	float or {'xx-small', 'x-small', 'small', 'medium', 'large',
fontstretch or stretch	{a numeric value in range 0-1000, 'ultra-condensed',
fontstyle or style	{'normal', 'italic', 'oblique'}
fontvariant or variant	{'normal', 'small-caps'}
fontweight or weight	{a numeric value in range 0-1000, 'ultralight', 'light',
gid	str
horizontalalignment or ha	{'center', 'right', 'left'}
in_layout	bool
label	object
linespacing	float (multiple of font size)
multialignment or ma	{'left', 'right', 'center'}
path_effects	*AbstractPathEffect*
picker	None or bool or callable
position	(float, float)
rasterized	bool or None
rotation	float or {'vertical', 'horizontal'}
rotation_mode	{None, 'default', 'anchor'}
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
text	object
transform	*Transform*
url	str
usetex	bool or None
verticalalignment or va	{'center', 'top', 'bottom', 'baseline', 'center_baseline'}
visible	bool
wrap	bool
x	float
y	float

Tab

Property	Description
zorder	float

__module__ = 'mpl_toolkits.mplot3d.art3d'

draw(*self*, *renderer*)

Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

Parameters

renderer

[*RendererBase* subclass.]

Notes

This method is overridden in the Artist subclasses.

get_tightbbox(*self*, *renderer*)

Like *Artist.get_window_extent*, but includes any clipping.

Parameters

renderer

[*RendererBase* subclass] renderer that will be used to draw the figures (i.e. `fig.canvas.get_renderer()`)

Returns

Bbox

The enclosing bounding box (in figure pixel coordinates).

set_3d_properties(*self*, *z=0*, *zdir='z'*)

Examples using `mpl_toolkits.mplot3d.art3d.Text3D`

mpl_toolkits.mplot3d.art3d.get_dir_vector

`mpl_toolkits.mplot3d.art3d.get_dir_vector(zdir)`

Return a direction vector.

Parameters

zdir

[{'x', 'y', 'z', None, 3-tuple}] The direction. Possible values are: - 'x': equivalent to (1, 0, 0) - 'y': equivalent to (0, 1, 0) - 'z': equivalent to (0, 0, 1) - *None*: equivalent to (0, 0, 0) - an iterable (x, y, z) is returned unchanged.

Returns

x, y, z

[array-like] The direction vector. This is either a numpy.array or *zdir* itself if *zdir* is already a length-3 iterable.

Examples using `mpl_toolkits.mplot3d.art3d.get_dir_vector`

mpl_toolkits.mplot3d.art3d.juggle_axes

`mpl_toolkits.mplot3d.art3d.juggle_axes(xs, ys, zs, zdir)`

Reorder coordinates so that 2D xs, ys can be plotted in the plane orthogonal to zdir. zdir is normally x, y or z. However, if zdir starts with a '-' it is interpreted as a compensation for rotate_axes.

Examples using `mpl_toolkits.mplot3d.art3d.juggle_axes`

mpl_toolkits.mplot3d.art3d.line_2d_to_3d

`mpl_toolkits.mplot3d.art3d.line_2d_to_3d(line, zs=0, zdir='z')`

Convert a 2D line to 3D.

Examples using `mpl_toolkits.mplot3d.art3d.line_2d_to_3d`

mpl_toolkits.mplot3d.art3d.line_collection_2d_to_3d

`mpl_toolkits.mplot3d.art3d.line_collection_2d_to_3d`(*col*, *zs=0*, *zdir='z'*)

Convert a LineCollection to a Line3DCollection object.

Examples using `mpl_toolkits.mplot3d.art3d.line_collection_2d_to_3d`

mpl_toolkits.mplot3d.art3d.patch_2d_to_3d

`mpl_toolkits.mplot3d.art3d.patch_2d_to_3d`(*patch*, *z=0*, *zdir='z'*)

Convert a Patch to a Patch3D object.

Examples using `mpl_toolkits.mplot3d.art3d.patch_2d_to_3d`

mpl_toolkits.mplot3d.art3d.patch_collection_2d_to_3d

`mpl_toolkits.mplot3d.art3d.patch_collection_2d_to_3d`(*col*, *zs=0*, *zdir='z'*, *depthshade=True*)

Convert a *PatchCollection* into a *Patch3DCollection* object (or a *PathCollection* into a *Path3DCollection* object).

Parameters

za

The location or locations to place the patches in the collection along the *zdir* axis. Default: 0.

zdir

The axis in which to place the patches. Default: "z".

depthshade

Whether to shade the patches to give a sense of depth. Default: *True*.

Examples using `mpl_toolkits.mplot3d.art3d.patch_collection_2d_to_3d`

mpl_toolkits.mplot3d.art3d.pathpatch_2d_to_3d

`mpl_toolkits.mplot3d.art3d.`**`pathpatch_2d_to_3d`**(*pathpatch*, *z=0*, *zdir='z'*)
Convert a PathPatch to a PathPatch3D object.

Examples using `mpl_toolkits.mplot3d.art3d.pathpatch_2d_to_3d`

- sphx_glr_gallery_mplot3d_pathpatch3d.py

mpl_toolkits.mplot3d.art3d.poly_collection_2d_to_3d

`mpl_toolkits.mplot3d.art3d.`**`poly_collection_2d_to_3d`**(*col*, *zs=0*, *zdir='z'*)
Convert a PolyCollection to a Poly3DCollection object.

Examples using `mpl_toolkits.mplot3d.art3d.poly_collection_2d_to_3d`

mpl_toolkits.mplot3d.art3d.rotate_axes

`mpl_toolkits.mplot3d.art3d.`**`rotate_axes`**(*xs*, *ys*, *zs*, *zdir*)
Reorder coordinates so that the axes are rotated with zdir along the original z axis. Prepending the axis with a '-' does the inverse transform, so zdir can be x, -x, y, -y, z or -z

Examples using `mpl_toolkits.mplot3d.art3d.rotate_axes`

mpl_toolkits.mplot3d.art3d.text_2d_to_3d

`mpl_toolkits.mplot3d.art3d.`**`text_2d_to_3d`**(*obj*, *z=0*, *zdir='z'*)
Convert a Text to a Text3D object.

Examples using `mpl_toolkits.mplot3d.art3d.text_2d_to_3d`

19.2.4 proj3d

proj3d.inv_transform(xs, ys, zs, M)	
proj3d.persp_transformation(zfront, zback)	

continues on next page

Table 57 – continued from previous page

proj3d.proj_points(points, M)	
proj3d.proj_trans_points(points, M)	
proj3d.proj_transform(xs, ys, zs, M)	Transform the points by the projection matrix
proj3d.proj_transform_clip(xs, ys, zs, M)	Transform the points by the projection matrix and return the clipping result returns txs, tys, tzs, tis
proj3d.rot_x(V, alpha)	
proj3d.transform(xs, ys, zs, M)	Transform the points by the projection matrix
proj3d.view_transformation(E, R, V)	
proj3d.world_transformation(xmin, xmax, ...)	Produce a matrix that scales homogeneous coords in the specified ranges to [0, 1], or [0, pb_aspect[i]] if the plotbox aspect ratio is specified.

mpl_toolkits.mplot3d.proj3d.inv_transform

mpl_toolkits.mplot3d.proj3d.**inv_transform**(*xs*, *ys*, *zs*, *M*)

Examples using `mpl_toolkits.mplot3d.proj3d.inv_transform`

mpl_toolkits.mplot3d.proj3d.persp_transformation

mpl_toolkits.mplot3d.proj3d.**persp_transformation**(*zfront*, *zback*)

Examples using `mpl_toolkits.mplot3d.proj3d.persp_transformation`

mpl_toolkits.mplot3d.proj3d.proj_points

mpl_toolkits.mplot3d.proj3d.**proj_points**(*points*, *M*)

Examples using `mpl_toolkits.mplot3d.proj3d.proj_points`

mpl_toolkits.mplot3d.proj3d.proj_trans_points

mpl_toolkits.mplot3d.proj3d.**proj_trans_points**(*points*, *M*)

Examples using `mpl_toolkits.mplot3d.proj3d.proj_trans_points`

mpl_toolkits.mplot3d.proj3d.proj_transform

mpl_toolkits.mplot3d.proj3d.**proj_transform**(*xs*, *ys*, *zs*, *M*)
Transform the points by the projection matrix

Examples using `mpl_toolkits.mplot3d.proj3d.proj_transform`

mpl_toolkits.mplot3d.proj3d.proj_transform_clip

mpl_toolkits.mplot3d.proj3d.**proj_transform_clip**(*xs*, *ys*, *zs*, *M*)
Transform the points by the projection matrix and return the clipping result returns txs, tys, tzs, tis

Examples using `mpl_toolkits.mplot3d.proj3d.proj_transform_clip`

mpl_toolkits.mplot3d.proj3d.rot_x

mpl_toolkits.mplot3d.proj3d.**rot_x**(*V*, *alpha*)

Examples using `mpl_toolkits.mplot3d.proj3d.rot_x`

mpl_toolkits.mplot3d.proj3d.transform

mpl_toolkits.mplot3d.proj3d.**transform**(*xs*, *ys*, *zs*, *M*)
Transform the points by the projection matrix

Examples using `mpl_toolkits.mplot3d.proj3d.transform`

mpl_toolkits.mplot3d.proj3d.view_transformation

mpl_toolkits.mplot3d.proj3d.**view_transformation**(*E*, *R*, *V*)

Examples using `mpl_toolkits.mplot3d.proj3d.view_transformation`

mpl_toolkits.mplot3d.proj3d.world_transformation

`mpl_toolkits.mplot3d.proj3d.`**`world_transformation`**(*xmin*, *xmax*, *ymin*, *ymax*, *zmin*, *zmax*, *pb_aspect=None*)

Produce a matrix that scales homogeneous coords in the specified ranges to [0, 1], or [0, pb_aspect[i]] if the plotbox aspect ratio is specified.

Examples using `mpl_toolkits.mplot3d.proj3d.world_transformation`

19.3 Matplotlib axes_grid Toolkit

Note: AxesGrid toolkit has been a part of matplotlib since v 0.99. Originally, the toolkit had a single namespace of *axes_grid*. In more recent version, the toolkit has divided into two separate namespace (*axes_grid1* and *axisartist*). While *axes_grid* namespace is maintained for the backward compatibility, use of *axes_grid1* and *axisartist* is recommended. For the documentation on `axes_grid`, see the previous version of the docs.

Part IV

External Resources

CHAPTER

TWENTY

BOOKS, CHAPTERS AND ARTICLES

- Mastering matplotlib by Duncan M. McGreggor
- Interactive Applications Using Matplotlib by Benjamin Root
- Matplotlib for Python Developers by Sandro Tosi
- Matplotlib chapter by John Hunter and Michael Droettboom in The Architecture of Open Source Applications
- Ten Simple Rules for Better Figures by Nicolas P. Rougier, Michael Droettboom and Philip E. Bourne
- Learning Scientific Programming with Python chapter 7 by Christian Hill

CHAPTER

TWENTYONE

VIDEOS

- Plotting with matplotlib by Mike Müller
- Introduction to NumPy and Matplotlib by Eric Jones
- Anatomy of Matplotlib by Benjamin Root
- Data Visualization Basics with Python (O'Reilly) by Randal S. Olson

CHAPTER

TWENTYTWO

TUTORIALS

- Matplotlib tutorial by Nicolas P. Rougier
- Anatomy of Matplotlib - IPython Notebooks by Benjamin Root

Part V

Third party packages

Several external packages that extend or build on Matplotlib functionality are listed below. They are maintained and distributed separately from Matplotlib and thus need to be installed individually.

Please submit an issue or pull request on GitHub if you have created a package that you would like to have included. We are also happy to host third party packages within the Matplotlib GitHub Organization.

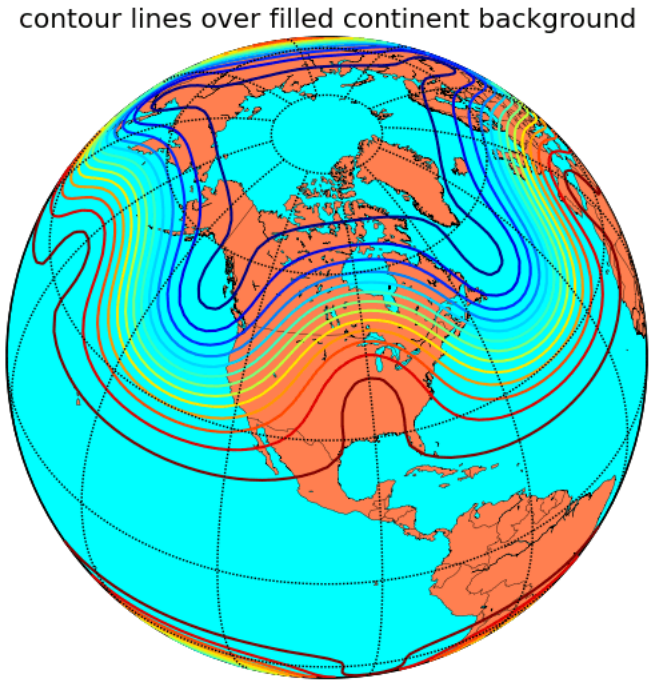
CHAPTER

TWENTYTHREE

MAPPING TOOLKITS

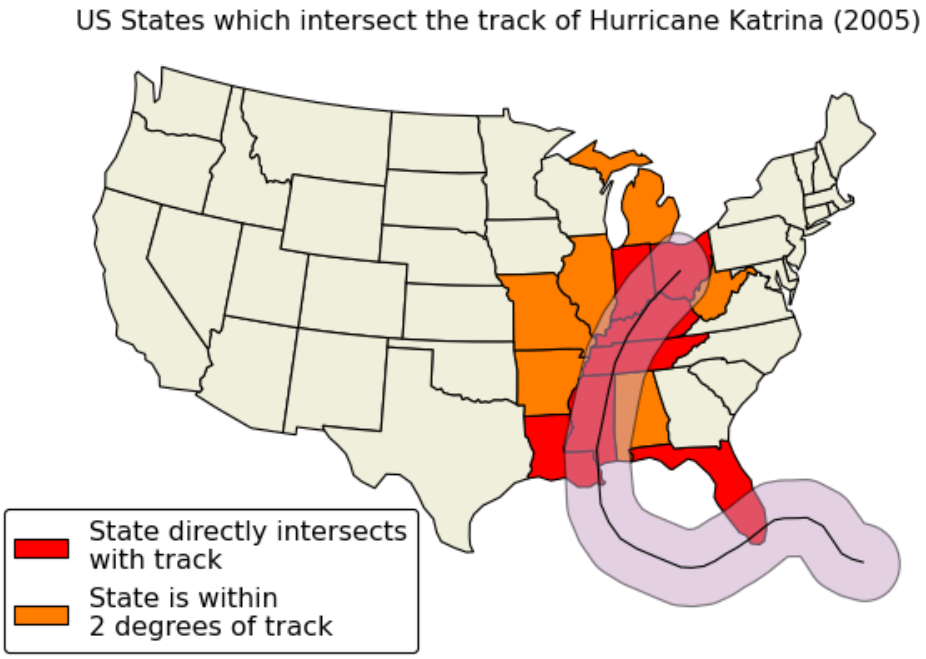
23.1 Basemap

Basemap plots data on map projections, with continental and political boundaries.



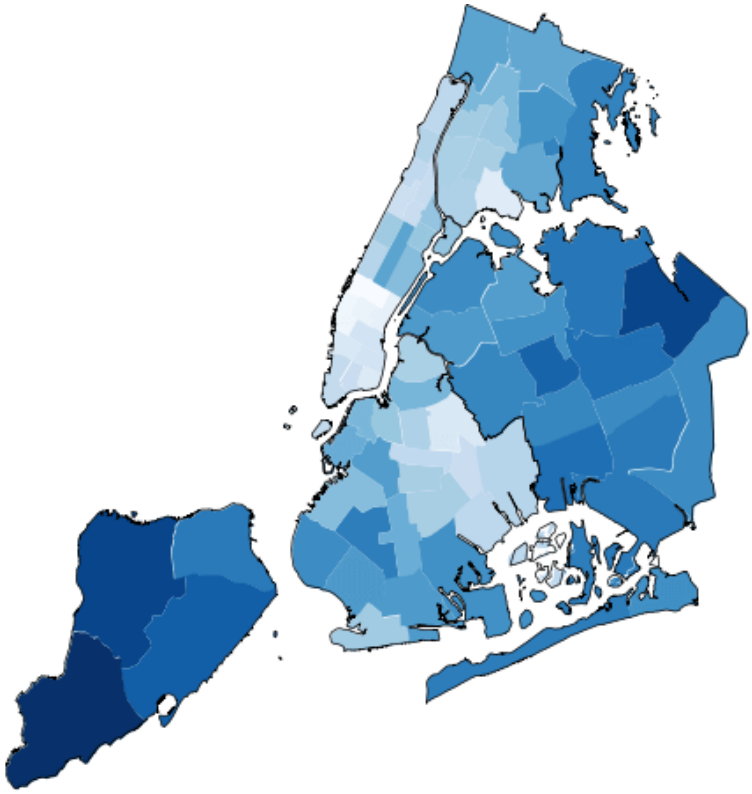
23.2 Cartopy

Cartopy builds on top of Matplotlib to provide object oriented map projection definitions and close integration with Shapely for powerful yet easy-to-use vector data processing tools. An example plot from the Cartopy gallery:



23.3 Geoplot

Geoplot builds on top of Matplotlib and Cartopy to provide a "standard library" of simple, powerful, and customizable plot types. An example plot from the Geoplot gallery:



23.4 Ridge Map

ridge_map uses Matplotlib, SRTM.py, NumPy, and scikit-image to make ridge plots of your favorite ridges.


THE WHITE
MOUNTAINS

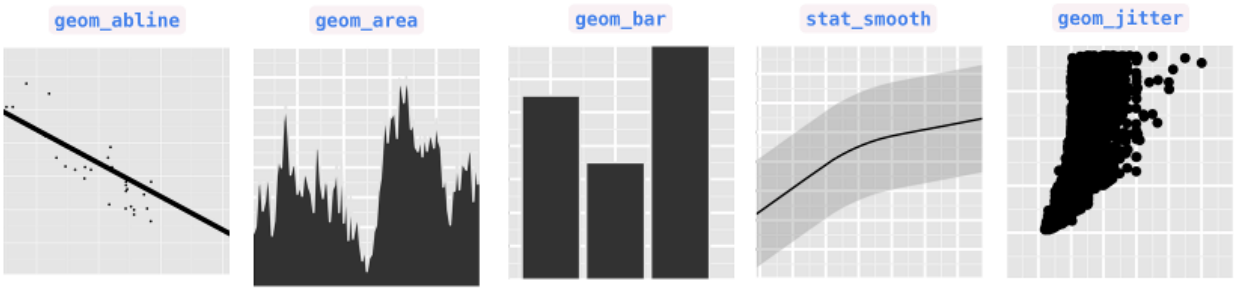
CHAPTER

TWENTYFOUR

DECLARATIVE LIBRARIES

24.1 ggplot

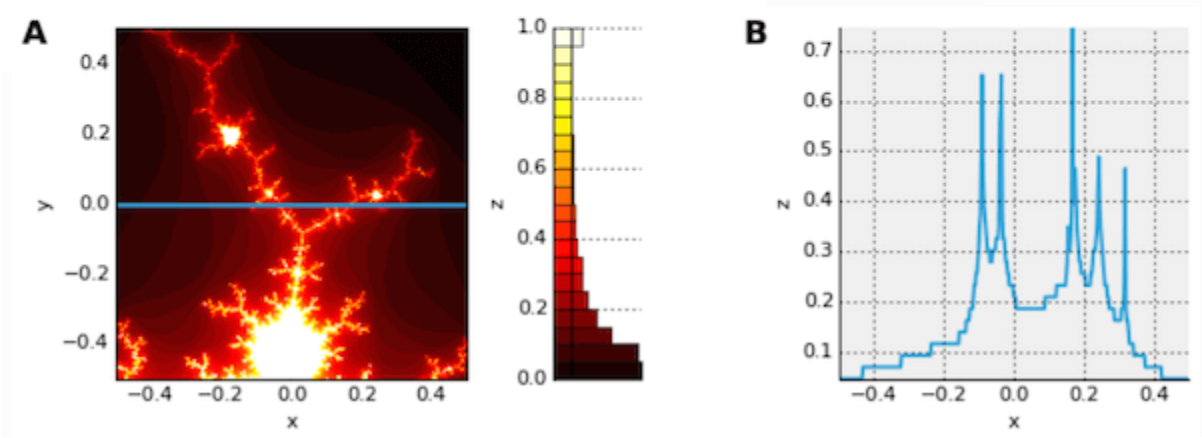
ggplot is a port of the R ggplot2 package to python based on Matplotlib.



24.2 holoviews

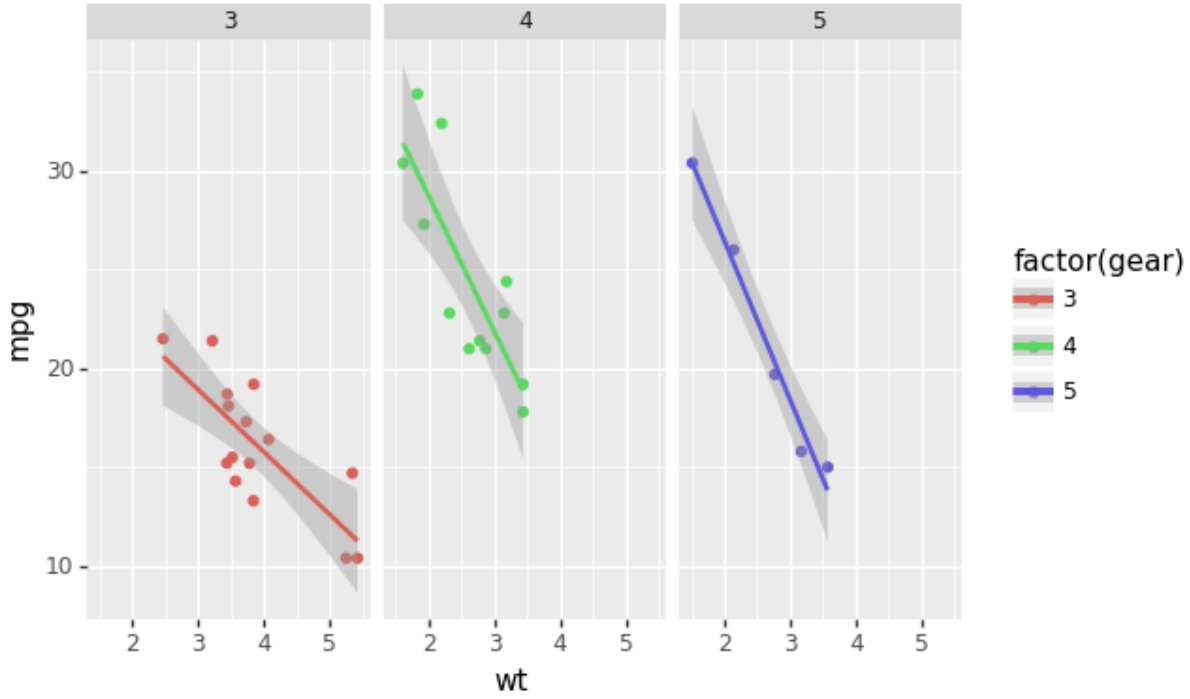
holoviews makes it easier to visualize data interactively, especially in a Jupyter notebook, by providing a set of declarative plotting objects that store your data and associated metadata. Your data is then immediately visualizable alongside or overlaid with other data, either statically or with automatically provided widgets for parameter exploration.

```
import numpy as np
import holoviews as hv
hv.notebook_extension('matplotlib')
fractal = hv.Image(np.load('mandelbrot.npy'))

((fractal * hv.HLine(y=0)).hist() + fractal.sample(y=0))
```



24.3 plotnine

plotnine implements a grammar of graphics, similar to R's ggplot2. The grammar allows users to compose plots by explicitly mapping data to the visual objects that make up the plot.


3
4
5
30
20
10
mpg
2
3
4
5
wt
factor(gear)
3
4
5

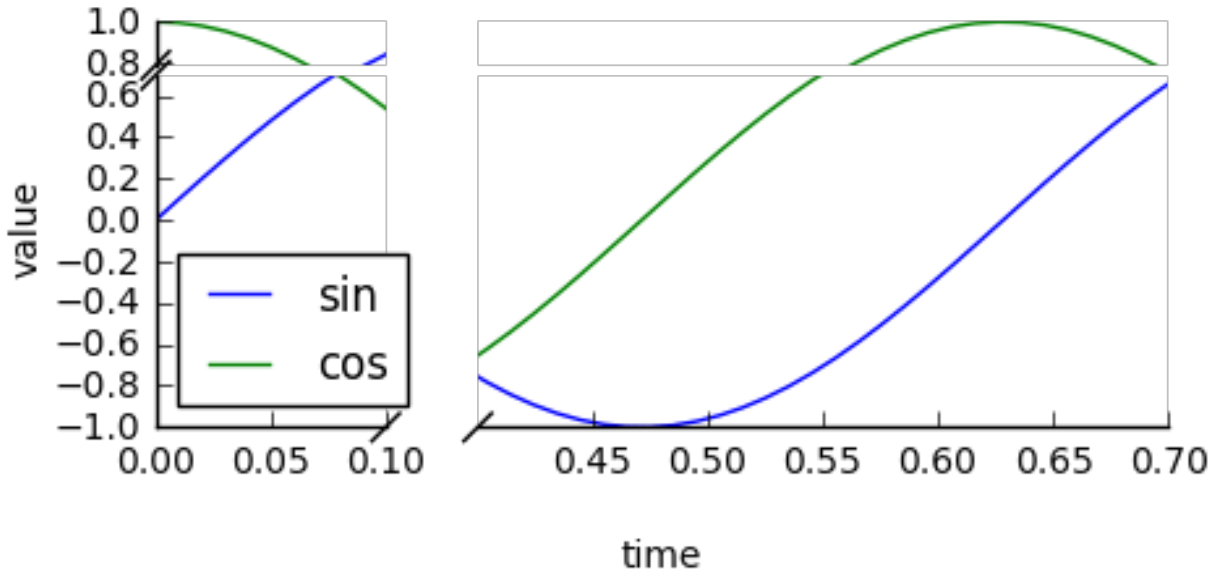
CHAPTER

TWENTYFIVE

SPECIALTY PLOTS

25.1 Broken Axes

brokenaxes supplies an axes class that can have a visual break to indicate a discontinuous range.

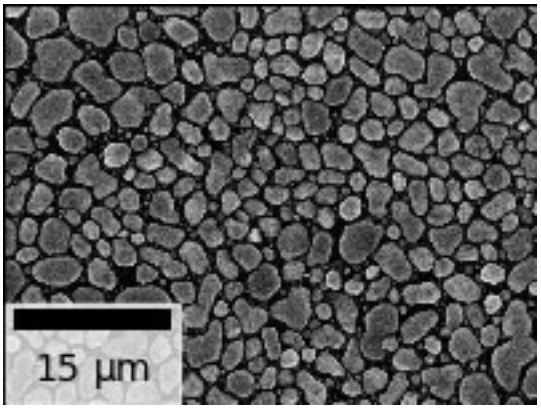


25.2 DeCiDa

DeCiDa is a library of functions and classes for electron device characterization, electronic circuit design and general data visualization and analysis.

25.3 matplotlib-scalebar

matplotlib-scalebar provides a new artist to display a scale bar, aka micron bar. It is particularly useful when displaying calibrated images plotted using `plt.imshow(...)`.

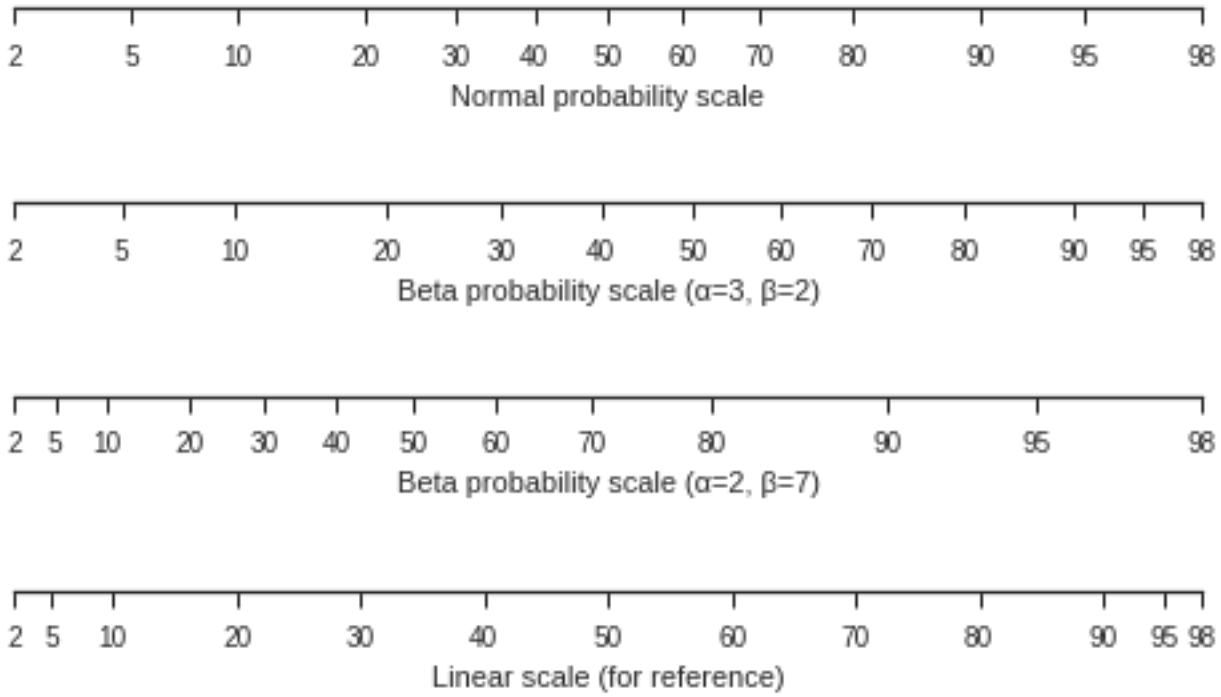


25.4 Matplotlib-Venn

Matplotlib-Venn provides a set of functions for plotting 2- and 3-set area-weighted (or unweighted) Venn diagrams.

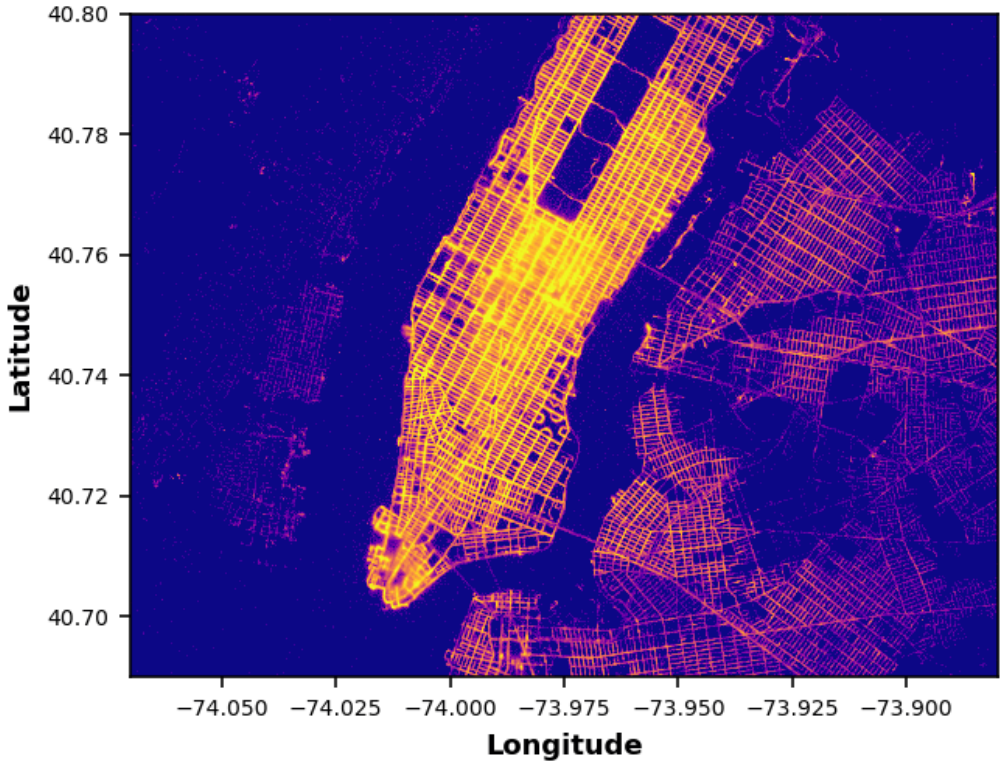
25.5 mpl-probscale

mpl-probscale is a small extension that allows Matplotlib users to specify probability scales. Simply importing the `probscale` module registers the scale with Matplotlib, making it accessible via e.g., `ax.set_xscale('prob')` or `plt.yscale('prob')`.



25.6 mpl-scatter-density

mpl-scatter-density is a small package that makes it easy to make scatter plots of large numbers of points using a density map. The following example contains around 13 million points and the plotting (excluding reading in the data) took less than a second on an average laptop:



When used in interactive mode, the density map is downsampled on-the-fly while panning/zooming in order to provide a smooth interactive experience.

25.7 mplstereonet

mplstereonet provides stereonets for plotting and analyzing orientation data in Matplotlib.

25.8 Natgrid

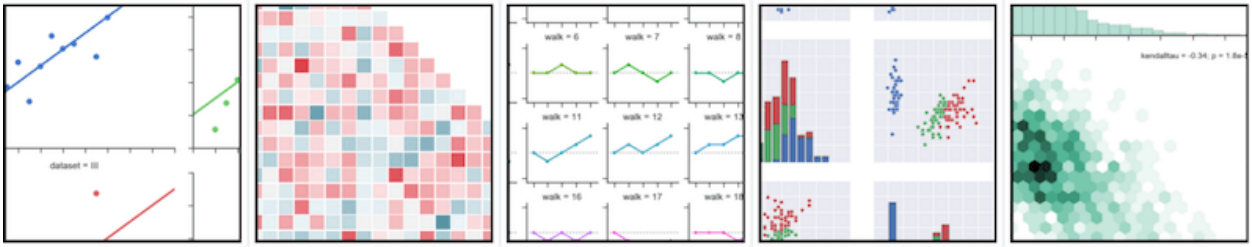
mpl_toolkits.natgrid is an interface to the natgrid C library for gridding irregularly spaced data.

25.9 pyUpSet

pyUpSet is a static Python implementation of the UpSet suite by Lex et al. to explore complex intersections of sets and data frames.

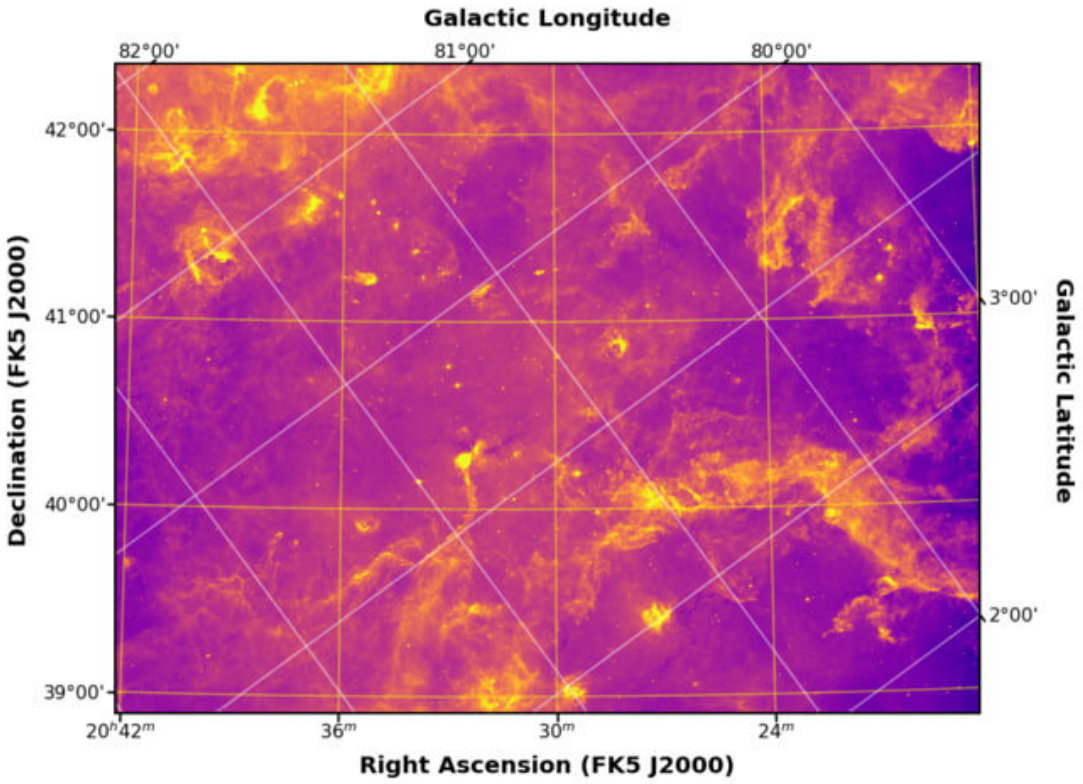
25.10 seaborn

seaborn is a high level interface for drawing statistical graphics with Matplotlib. It aims to make visualization a central part of exploring and understanding complex datasets.



25.11 WCSAxes

The Astropy core package includes a submodule called WCSAxes (available at astropy.visualization.wcsaxes) which adds Matplotlib projections for Astronomical image data. The following is an example of a plot made with WCSAxes which includes the original coordinate system of the image and an overlay of a different coordinate system:

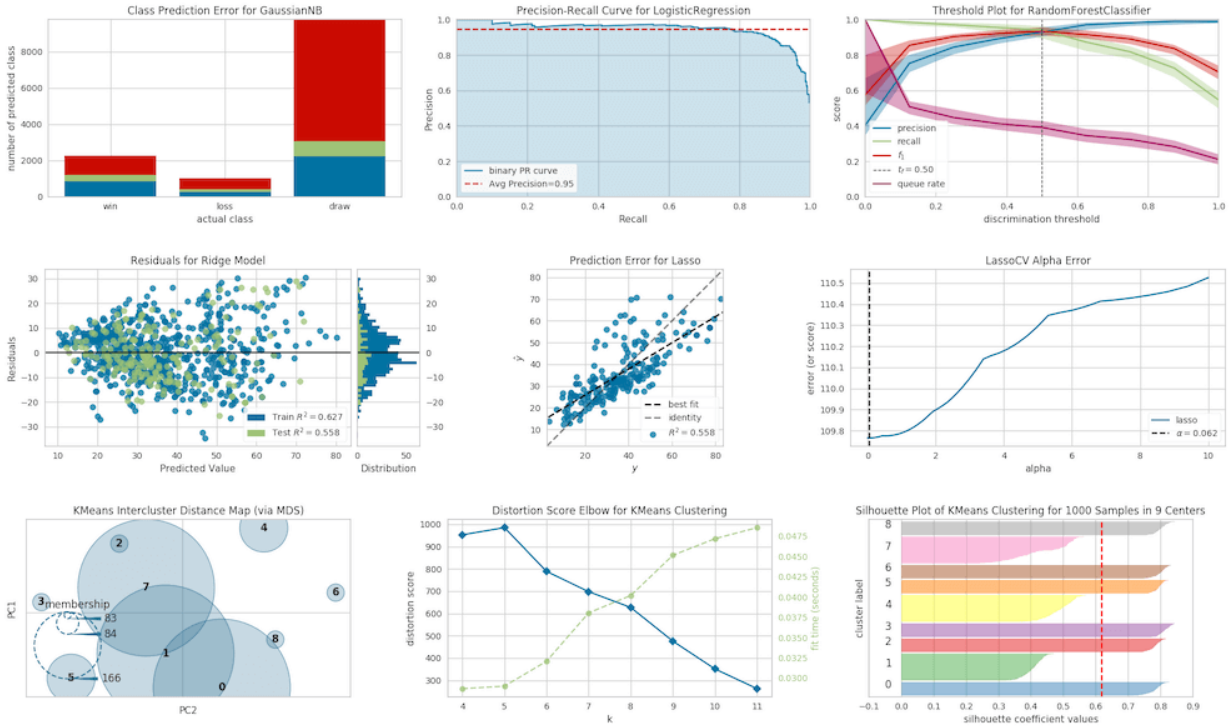


25.12 Windrose

Windrose is a Python Matplotlib, Numpy library to manage wind data, draw windroses (also known as polar rose plots), draw probability density functions and fit Weibull distributions.

25.13 Yellowbrick

Yellowbrick is a suite of visual diagnostic tools for machine learning that enables human steering of the model selection process. Yellowbrick combines scikit-learn with matplotlib using an estimator-based API called the `Visualizer`, which wraps both sklearn models and matplotlib Axes. `Visualizer` objects fit neatly into the machine learning workflow allowing data scientists to integrate visual diagnostic and model interpretation tools into experimentation without extra steps.


Class Prediction Error for GaussianNB
number of predicted class
actual class
win
loss
draw
Precision-Recall Curve for LogisticRegression
Precision
Recall
binary PR curve
Avg Precision=0.95
Threshold Plot for RandomForestClassifier
score
discrimination threshold
precision
recall
queue rate
Residuals for Ridge Model
Residuals
Predicted Value
Distribution
Train $R^2 = 0.627$
Test $R^2 = 0.558$
Prediction Error for Lasso
best fit
identity
$R^2 = 0.558$
LassoCV Alpha Error
error (or score)
alpha
lasso
KMeans Intercluster Distance Map (via MDS)
PC1
PC2
membership
Distortion Score Elbow for KMeans Clustering
distortion score
fit time (seconds)
k
Silhouette Plot of KMeans Clustering for 1000 Samples in 9 Centers
cluster label
silhouette coefficient values

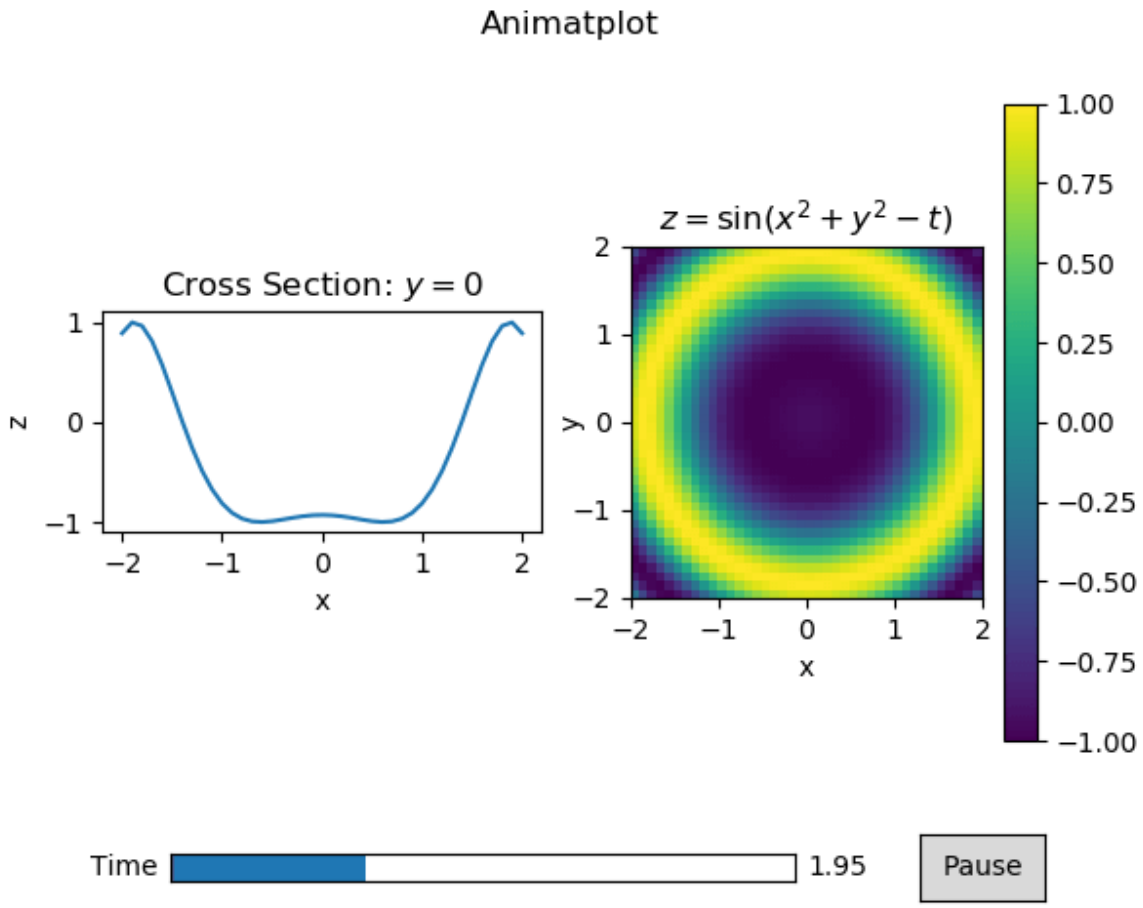
CHAPTER

TWENTYSIX

ANIMATIONS

26.1 animatplot

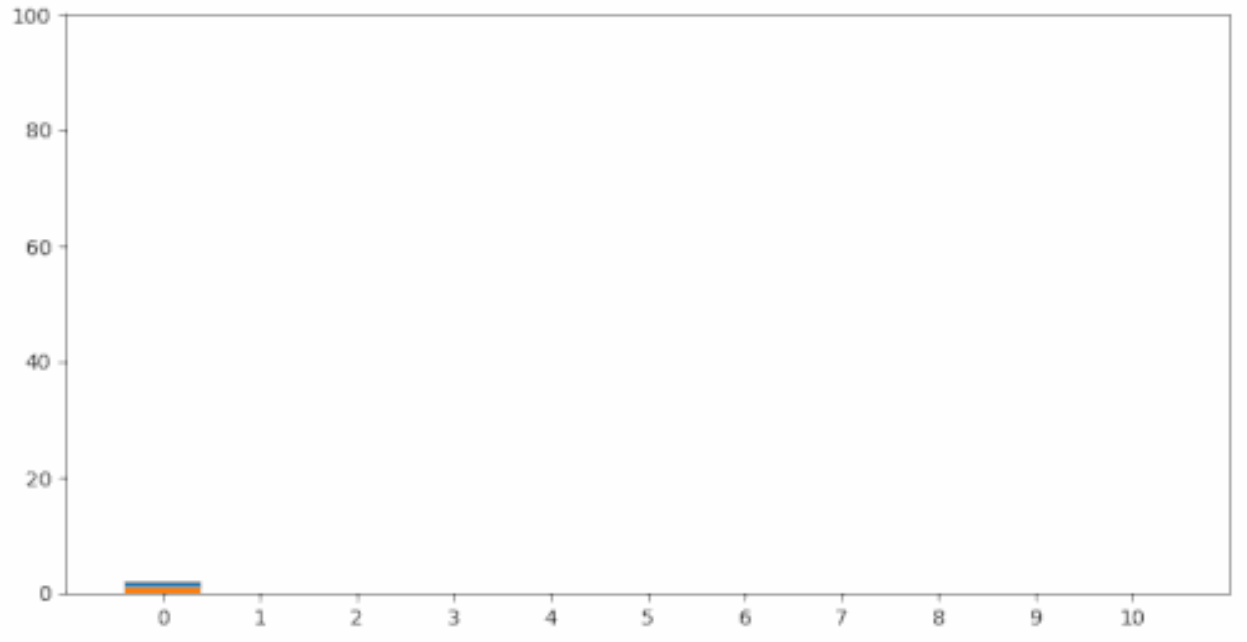
animatplot is a library for producing interactive animated plots with the goal of making production of animated plots almost as easy as static ones.



For an animated version of the above picture and more examples, see the animatplot gallery.

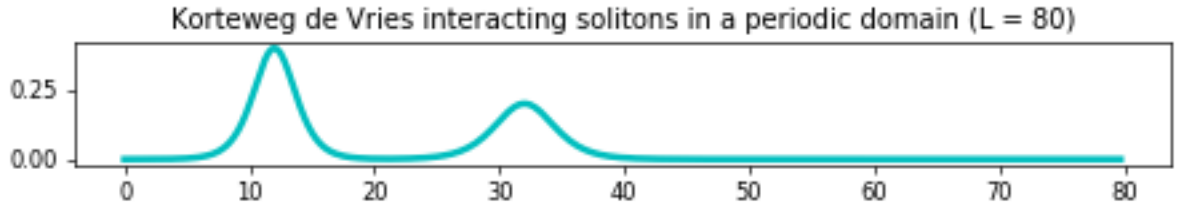
26.2 gif

gif is an ultra lightweight animated gif API.



26.3 numpngw

numpngw provides functions for writing NumPy arrays to PNG and animated PNG files. It also includes the class `AnimatedPNGWriter` that can be used to save a Matplotlib animation as an animated PNG file. See the example on the PyPI page or at the `numpngw` github repository.



CHAPTER

TWENTYSEVEN

INTERACTIVITY

27.1 mplcursors

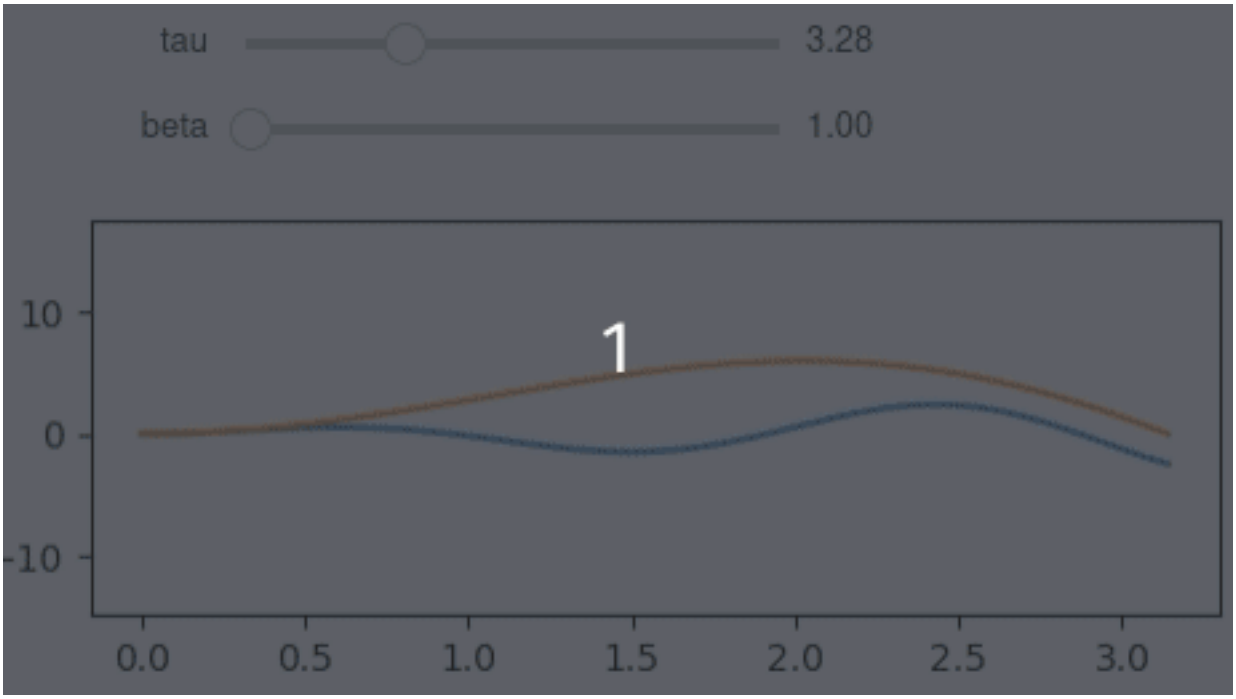
mplcursors provides interactive data cursors for Matplotlib.

27.2 MplDataCursor

MplDataCursor is a toolkit written by Joe Kington to provide interactive "data cursors" (clickable annotation boxes) for Matplotlib.

27.3 mpl_interactions

mpl_interactions makes it easy to create interactive plots controlled by sliders and other widgets. It also provides several handy capabilties such as manual image segmentation, comparing cross-sections of arrays, and using the scroll wheel to zoom.


tau
3.28
beta
1.00
10
0
-10
1
0.0
0.5
1.0
1.5
2.0
2.5
3.0

CHAPTER

TWENTYEIGHT

RENDERING BACKENDS

28.1 mplcairo

mplcairo is a cairo backend for Matplotlib, with faster and more accurate marker drawing, support for a wider selection of font formats and complex text layout, and various other features.

28.2 gr

gr is a framework for cross-platform visualisation applications, which can be used as a high-performance Matplotlib backend.

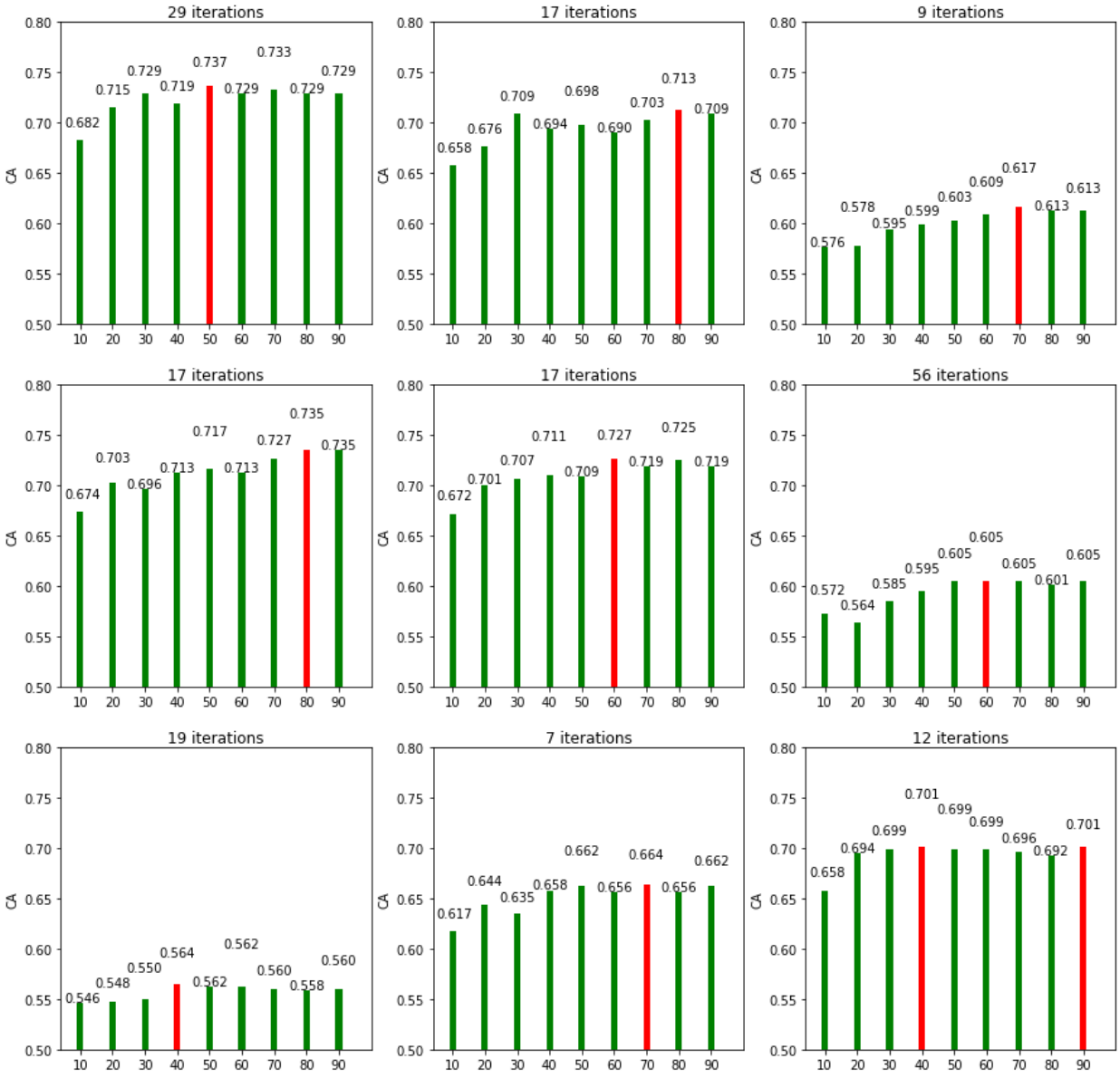
CHAPTER

TWENTYNINE

MISCELLANEOUS

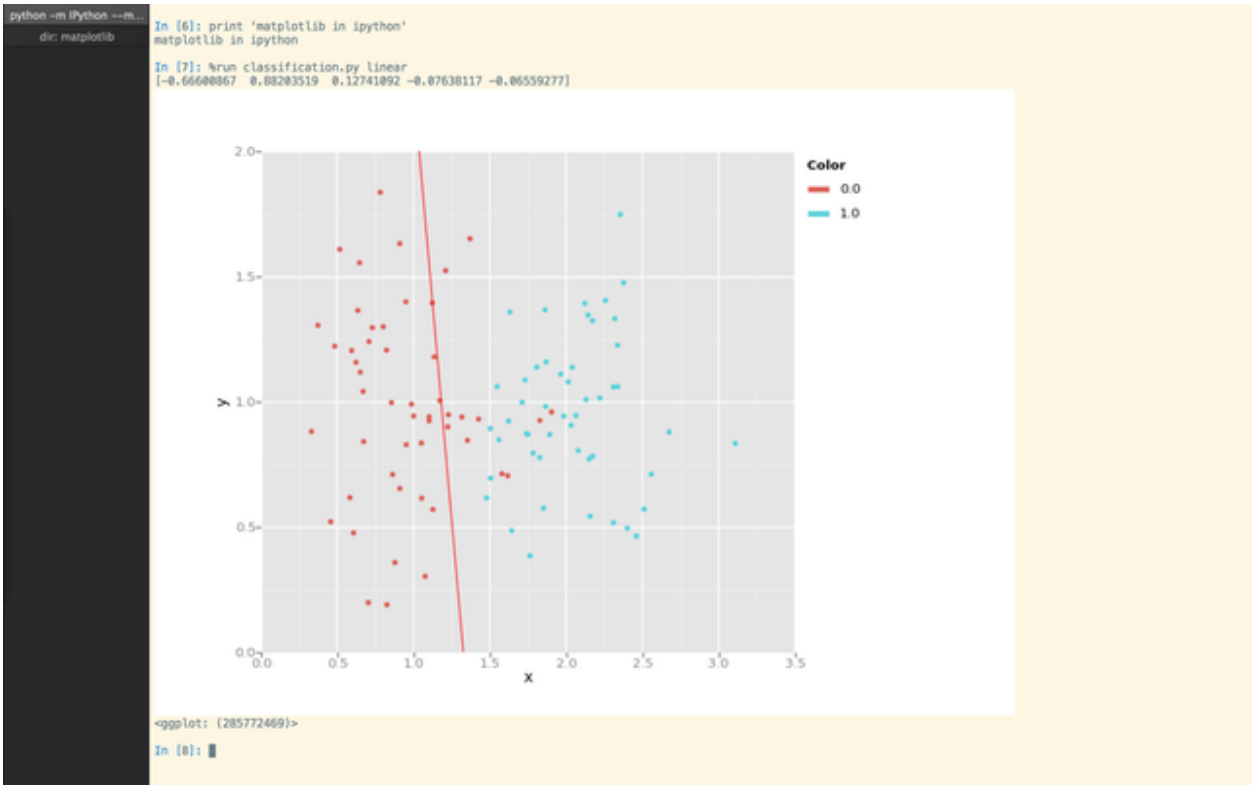
29.1 adjustText

adjustText is a small library for automatically adjusting text position in Matplotlib plots to minimize overlaps between them, specified points and other objects.



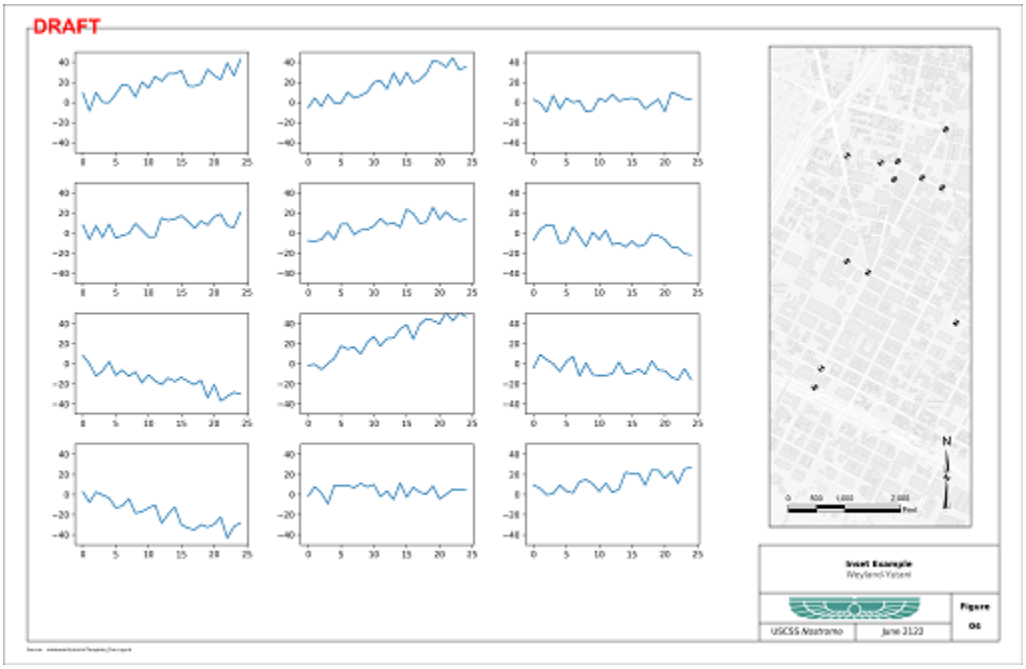
29.2 iTerm2 terminal backend

matplotlib_iterm2 is an external Matplotlib backend using the iTerm2 nightly build inline image display feature.



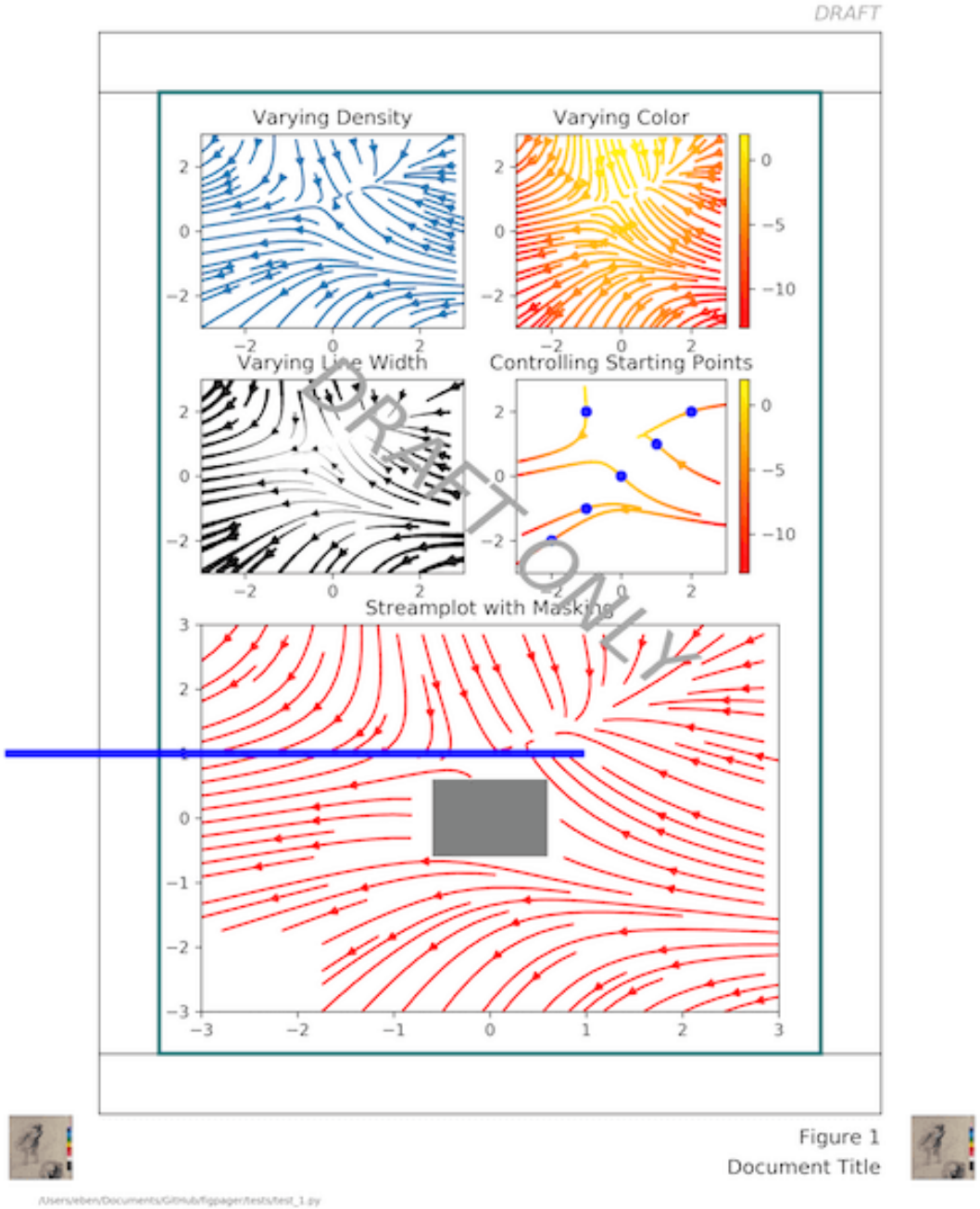
29.3 mpl-template

mpl-template provides a customizable way to add engineering figure elements such as a title block, border, and logo.



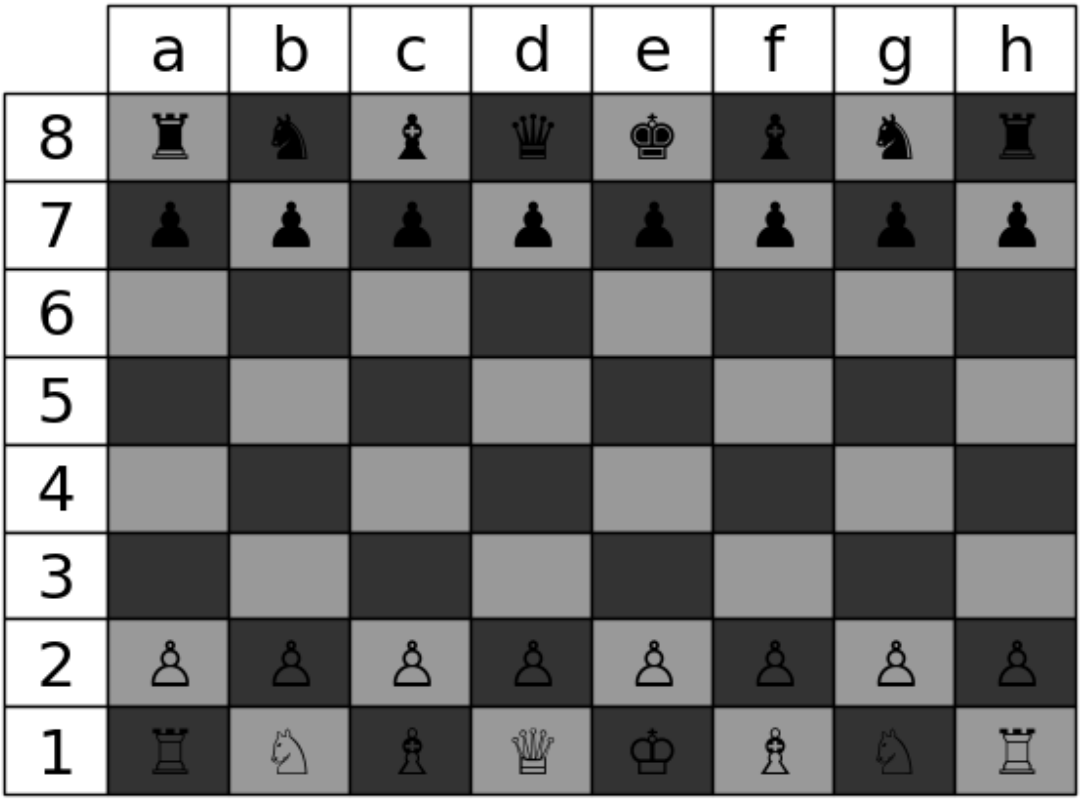
29.4 figpager

figpager provides customizable figure elements such as text, lines and images and subplot layout control for single or multi page output.



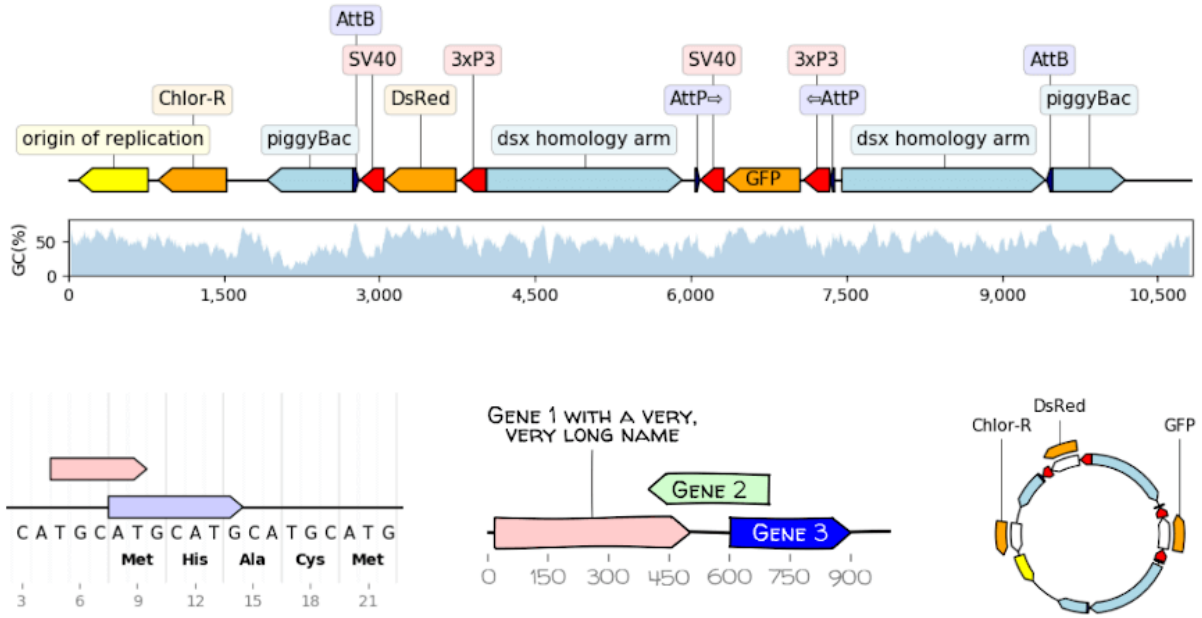
29.5 blume

blume provides a replacement for the Matplotlib `table` module. It fixes a number of issues with the existing table. See the blume github repository for more details.



29.6 DNA Features Viewer

DNA Features Viewer provides methods to plot annotated DNA sequence maps (possibly along other Matplotlib plots) for Bioinformatics and Synthetic Biology applications.



CHAPTER

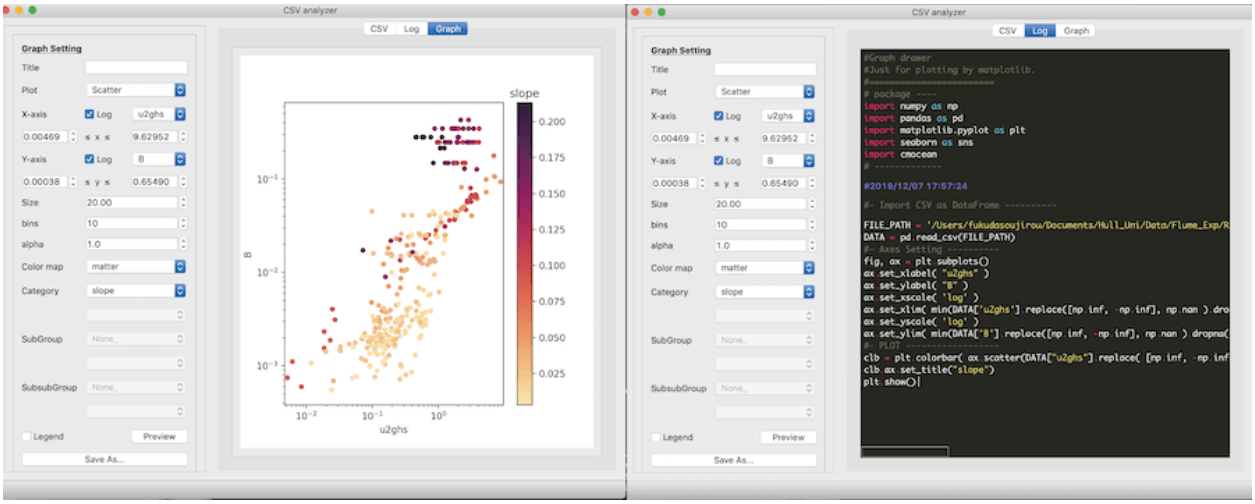
THIRTY

GUI APPLICATIONS

30.1 sviewgui

sviewgui is a PyQt-based GUI for visualisation of data from csv files or `pandas.DataFrame`s. Main features:

- Scatter, line, density, histogram, and box plot types
- Settings for the marker size, line width, number of bins of histogram, color map (from cmocean)
- Save figure as editable PDF
- Code of the plotted graph is available so that it can be reused and modified outside of sviewgui



Part VI

The Matplotlib Developers' Guide

CHAPTER

THIRTYONE

CONTRIBUTING

This project is a community effort, and everyone is welcome to contribute. We follow the Python Software Foundation Code of Conduct in everything we do.

The project is hosted on https://github.com/matplotlib/matplotlib

31.1 Submitting a bug report

If you find a bug in the code or documentation, do not hesitate to submit a ticket to the Bug Tracker. You are also welcome to post feature requests or pull requests.

If you are reporting a bug, please do your best to include the following:

1. A short, top-level summary of the bug. In most cases, this should be 1-2 sentences.
2. A short, self-contained code snippet to reproduce the bug, ideally allowing a simple copy and paste to reproduce. Please do your best to reduce the code snippet to the minimum required.
3. The actual outcome of the code snippet.
4. The expected outcome of the code snippet.
5. The Matplotlib version, Python version and platform that you are using. You can grab the version with the following commands:

```
>>> import matplotlib
>>> matplotlib.__version__
'1.5.3'
>>> import platform
>>> platform.python_version()
'2.7.12'
```

We have preloaded the issue creation page with a Markdown template that you can use to organize this information.

Thank you for your help in keeping bug reports complete, targeted and descriptive.

31.2 Retrieving and installing the latest version of the code

When developing Matplotlib, sources must be downloaded, built, and installed into a local environment on your machine.

We use Git for version control and GitHub for hosting our main repository.

You can check out the latest sources with the command (see *Set up your fork* for more details):

```
git clone https://github.com/matplotlib/matplotlib.git
```

and navigate to the `matplotlib` directory. If you have the proper privileges, you can use `git@` instead of `https://`, which works through the ssh protocol and might be easier to use if you are using 2-factor authentication.

31.2.1 Installing Matplotlib in developer mode

It is strongly recommended to set up a clean virtual environment. Do not use on a preexisting environment!

A new environment can be set up with

```
python3 -mvenv /path/to/devel/env
```

and activated with one of the following:

```
source /path/to/devel/env/bin/activate  # Linux/macOS
/path/to/devel/env/Scripts/activate.bat  # Windows cmd.exe
/path/to/devel/env/Scripts/Activate.ps1  # Windows PowerShell
```

Whenever you plan to work on Matplotlib, remember to activate the development environment in your shell!

To install Matplotlib (and compile the C-extensions) run the following command from the top-level directory

```
python -mpip install -ve .
```

This installs Matplotlib in 'editable/develop mode', i.e., builds everything and places the correct link entries in the install directory so that python will be able to import Matplotlib from the source directory. Thus, any changes to the `*.py` files will be reflected the next time you import the library. If you change the C-extension source (which might happen if you change branches) you will need to run

```
python setup.py build_ext --inplace
```

or re-run `python -mpip install -ve .`.

You can then run the tests to check your work environment is set up properly:

```
python -mpytest
```

Note: **Additional dependencies for testing**: pytest (version 3.6 or later), Ghostscript, Inkscape

See also:

- *Developer's tips for testing*

31.3 Contributing code

31.3.1 How to contribute

The preferred way to contribute to Matplotlib is to fork the main repository on GitHub, then submit a "pull request" (PR).

The best practices for using GitHub to make PRs to Matplotlib are documented in the *Development workflow* section.

A brief overview is:

1. Create an account on GitHub if you do not already have one.
2. Fork the project repository: click on the 'Fork' button near the top of the page. This creates a copy of the code under your account on the GitHub server.
3. Clone this copy to your local disk:

   ```
   $ git clone https://github.com/YourLogin/matplotlib.git
   ```

4. Create a branch to hold your changes:

   ```
   $ git checkout -b my-feature origin/master
   ```

   and start making changes. Never work in the `master` branch!

5. Work on this copy, on your computer, using Git to do the version control. When you're done editing e.g., `lib/matplotlib/collections.py`, do:

   ```
   $ git add lib/matplotlib/collections.py
   $ git commit
   ```

   to record your changes in Git, then push them to GitHub with:

   ```
   $ git push -u origin my-feature
   ```

Finally, go to the web page of your fork of the Matplotlib repo, and click 'Pull request' to send your changes to the maintainers for review. You may want to consider sending an email to the mailing list for more visibility.

See also:

- Git documentation
- Git-Contributing to a Project
- Introduction to GitHub
- *Development workflow*
- *Working with Matplotlib source code*

31.3.2 Contributing pull requests

It is recommended to check that your contribution complies with the following rules before submitting a pull request:

- If your pull request addresses an issue, please use the title to describe the issue and mention the issue number in the pull request description to ensure that a link is created to the original issue.
- All public methods should have informative docstrings with sample usage when appropriate. Use the numpy docstring standard.
- Formatting should follow the recommendations of PEP8. You should consider installing/enabling automatic PEP8 checking in your editor. Part of the test suite is checking PEP8 compliance, things go smoother if the code is mostly PEP8 compliant to begin with.
- Each high-level plotting function should have a simple example in the `Example` section of the docstring. This should be as simple as possible to demonstrate the method. More complex examples should go in the `examples` tree.
- Changes (both new features and bugfixes) should be tested. See *Developer's tips for testing* for more details.
- Import the following modules using the standard scipy conventions:

```
import numpy as np
import numpy.ma as ma
import matplotlib as mpl
import matplotlib.pyplot as plt
import matplotlib.cbook as cbook
import matplotlib.patches as mpatches
```

  In general, Matplotlib modules should **not** import *rcParams* using `from matplotlib import rcParams`, but rather access it as `mpl.rcParams`. This is because some modules are imported very early, before the *rcParams* singleton is constructed.
- If your change is a major new feature, add an entry to the `What's new` section by adding a new file in `doc/users/next_whats_new` (see `doc/users/next_whats_new/README.rst` for more information).

- If you change the API in a backward-incompatible way, please document it in `doc/api/api_changes`, by adding to the relevant file (see `doc/api/api_changes.rst` for more information)
- See below for additional points about *Keyword argument processing*, if applicable for your pull request.

In addition, you can check for common programming errors with the following tools:

- Code with a good unittest coverage (at least 70%, better 100%), check with:

```
python -mpip install coverage
python -mpytest --cov=matplotlib --showlocals -v
```

- No pyflakes warnings, check with:

```
python -mpip install pyflakes
pyflakes path/to/module.py
```

Note: The current state of the Matplotlib code base is not compliant with all of those guidelines, but we expect that enforcing those constraints on all new contributions will move the overall code base quality in the right direction.

See also:

- *Coding guidelines*
- *Developer's tips for testing*
- *Writing documentation*

31.3.3 Issues for New Contributors

New contributors should look for the following tags when looking for issues. We strongly recommend that new contributors tackle issues labeled good first issue as they are easy, well documented issues, that do not require an understanding of the different submodules of Matplotlib. This helps the contributor become familiar with the contribution workflow, and for the core devs to become acquainted with the contributor; besides which, we frequently underestimate how easy an issue is to solve!

31.4 Contributing documentation

Code is not the only way to contribute to Matplotlib. For instance, documentation is also a very important part of the project and often doesn't get as much attention as it deserves. If you find a typo in the documentation, or have made improvements, do not hesitate to send an email to the mailing list or submit a GitHub pull request. To make a pull request, refer to the guidelines outlined in *How to contribute*.

Full documentation can be found under the `doc/`, `tutorials/`, and `examples/` directories.

See also:

- *Writing documentation*

31.5 Other ways to contribute

It also helps us if you spread the word: reference the project from your blog and articles or link to it from your website! If Matplotlib contributes to a project that leads to a scientific publication, please follow the *Citing Matplotlib* guidelines.

31.6 Coding guidelines

31.6.1 API changes

Changes to the public API must follow a standard deprecation procedure to prevent unexpected breaking of code that uses Matplotlib.

- Deprecations must be announced via an entry in the most recent `doc/api/api_changes_X.Y`
- Deprecations are targeted at the next point-release (i.e. 3.x.0).
- The deprecated API should, to the maximum extent possible, remain fully functional during the deprecation period. In cases where this is not possible, the deprecation must never make a given piece of code do something different than it was before; at least an exception should be raised.
- If possible, usage of an deprecated API should emit a *`MatplotlibDeprecationWarning`*. There are a number of helper tools for this:
  - Use `cbook.warn_deprecated()` for general deprecation warnings.
  - Use the decorator `@cbook.deprecated` to deprecate classes, functions, methods, or properties.
  - To warn on changes of the function signature, use the decorators `@cbook._delete_parameter`, `@cbook._rename_parameter`, and `@cbook._make_keyword_only`.

- Deprecated API may be removed two point-releases after they were deprecated.

31.6.2 Adding new API

Every new function, parameter and attribute that is not explicitly marked as private (i.e., starts with an underscore) becomes part of Matplotlib's public API. As discussed above, changing the existing API is cumbersome. Therefore, take particular care when adding new API:

- Mark helper functions and internal attributes as private by prefixing them with an underscore.
- Carefully think about good names for your functions and variables.
- Try to adopt patterns and naming conventions from existing parts of the Matplotlib API.
- Consider making as many arguments keyword-only as possible. See also API Evolution the Right Way -- Add Parameters Compatibly.

31.6.3 New modules and files: installation

- If you have added new files or directories, or reorganized existing ones, make sure the new files are included in the match patterns in `MANIFEST.in`, and/or in *package_data* in `setup.py`.

31.6.4 C/C++ extensions

- Extensions may be written in C or C++.
- Code style should conform to PEP7 (understanding that PEP7 doesn't address C++, but most of its admonitions still apply).
- Python/C interface code should be kept separate from the core C/C++ code. The interface code should be named `FOO_wrap.cpp` or `FOO_wrapper.cpp`.
- Header file documentation (aka docstrings) should be in Numpydoc format. We don't plan on using automated tools for these docstrings, and the Numpydoc format is well understood in the scientific Python community.

31.6.5 Keyword argument processing

Matplotlib makes extensive use of `**kwargs` for pass-through customizations from one function to another. A typical example is in *matplotlib.pyplot.text()*. The definition of the pylab text function is a simple pass-through to *matplotlib.axes.Axes.text()*:

```
# in pylab.py
def text(*args, **kwargs):
    ret =  gca().text(*args, **kwargs)
    draw_if_interactive()
    return ret
```

text() in simplified form looks like this, i.e., it just passes all `args` and `kwargs` on to `matplotlib.text.Text.__init__()`:

```
# in axes/_axes.py
def text(self, x, y, s, fontdict=None, withdash=False, **kwargs):
    t = Text(x=x, y=y, text=s, **kwargs)
```

and `__init__()` (again with liberties for illustration) just passes them on to the *matplotlib.artist.Artist.update()* method:

```
# in text.py
def __init__(self, x=0, y=0, text='', **kwargs):
    Artist.__init__(self)
    self.update(kwargs)
```

`update` does the work looking for methods named like `set_property` if `property` is a keyword argument. i.e., no one looks at the keywords, they just get passed through the API to the artist constructor which looks for suitably named methods and calls them with the value.

As a general rule, the use of `**kwargs` should be reserved for pass-through keyword arguments, as in the example above. If all the keyword args are to be used in the function, and not passed on, use the key/value keyword args in the function definition rather than the `**kwargs` idiom.

In some cases, you may want to consume some keys in the local function, and let others pass through. Instead of popping arguments to use off `**kwargs`, specify them as keyword-only arguments to the local function. This makes it obvious at a glance which arguments will be consumed in the function. For example, in *plot()*, `scalex` and `scaley` are local arguments and the rest are passed on as *Line2D()* keyword arguments:

```
# in axes/_axes.py
def plot(self, *args, scalex=True, scaley=True, **kwargs):
    lines = []
    for line in self._get_lines(*args, **kwargs):
        self.add_line(line)
        lines.append(line)
```

31.6.6 Using logging for debug messages

Matplotlib uses the standard python `logging` library to write verbose warnings, information, and debug messages. Please use it! In all those places you write `print()` statements to do your debugging, try using `log.debug()` instead!

To include `logging` in your module, at the top of the module, you need to `import logging`. Then calls in your code like:

```
_log = logging.getLogger(__name__)  # right after the imports

# code
# more code
_log.info('Here is some information')
_log.debug('Here is some more detailed information')
```

will log to a logger named `matplotlib.yourmodulename`.

If an end-user of Matplotlib sets up `logging` to display at levels more verbose than `logging.WARNING` in their code with the Matplotlib-provided helper:

```
plt.set_loglevel("debug")
```

or manually with

```
import logging
logging.basicConfig(level=logging.DEBUG)
import matplotlib.pyplot as plt
```

Then they will receive messages like:

```
DEBUG:matplotlib.backends:backend MacOSX version unknown
DEBUG:matplotlib.yourmodulename:Here is some information
DEBUG:matplotlib.yourmodulename:Here is some more detailed information
```

Which logging level to use?

There are five levels at which you can emit messages.

- `logging.critical` and `logging.error` are really only there for errors that will end the use of the library but not kill the interpreter.
- `logging.warning` and `cbook._warn_external` are used to warn the user, see below.
- `logging.info` is for information that the user may want to know if the program behaves oddly. They are not displayed by default. For instance, if an object isn't drawn because its position is `NaN`, that can usually be ignored, but a mystified user could call `logging.basicConfig(level=logging.INFO)` and get an error message that says why.

- `logging.debug` is the least likely to be displayed, and hence can be the most verbose. "Expected" code paths (e.g., reporting normal intermediate steps of layouting or rendering) should only log at this level.

By default, `logging` displays all log messages at levels higher than `logging.WARNING` to `sys.stderr`.

The logging tutorial suggests that the difference between `logging.warning` and `cbook._warn_external` (which uses `warnings.warn`) is that `cbook._warn_external` should be used for things the user must change to stop the warning (typically in the source), whereas `logging.warning` can be more persistent. Moreover, note that `cbook._warn_external` will by default only emit a given warning *once* for each line of user code, whereas `logging.warning` will display the message every time it is called.

By default, `warnings.warn` displays the line of code that has the `warn` call. This usually isn't more informative than the warning message itself. Therefore, Matplotlib uses `cbook._warn_external` which uses `warnings.warn`, but goes up the stack and displays the first line of code outside of Matplotlib. For example, for the module:

```
# in my_matplotlib_module.py
import warnings

def set_range(bottom, top):
    if bottom == top:
        warnings.warn('Attempting to set identical bottom==top')
```

running the script:

```
from matplotlib import my_matplotlib_module
my_matplotlib_module.set_range(0, 0)  #set range
```

will display:

```
UserWarning: Attempting to set identical bottom==top
warnings.warn('Attempting to set identical bottom==top')
```

Modifying the module to use `cbook._warn_external`:

```
from matplotlib import cbook

def set_range(bottom, top):
    if bottom == top:
        cbook._warn_external('Attempting to set identical bottom==top')
```

and running the same script will display:

```
UserWarning: Attempting to set identical bottom==top
my_matplotlib_module.set_range(0, 0)  #set range
```

31.6.7 Writing examples

We have hundreds of examples in subdirectories of `matplotlib/examples`, and these are automatically generated when the website is built to show up in the `examples` section of the website.

Any sample data that the example uses should be kept small and distributed with Matplotlib in the `lib/matplotlib/mpl-data/sample_data/` directory. Then in your example code you can load it into a file handle with:

```
import matplotlib.cbook as cbook
fh = cbook.get_sample_data('mydata.dat')
```

CHAPTER

THIRTYTWO

DEVELOPER'S TIPS FOR TESTING

Matplotlib's testing infrastructure depends on pytest. The tests are in `lib/matplotlib/tests`, and customizations to the pytest testing infrastructure are in *matplotlib.testing*.

32.1 Requirements

Install the latest version of Matplotlib as documented in *Retrieving and installing the latest version of the code*.

The following software is required to run the tests:

- pytest (>=3.6)
- Ghostscript (>= 9.0, to render PDF files)
- Inkscape (to render SVG files)

Optionally you can install:

- pytest-cov (>=2.3.1) to collect coverage information
- pytest-flake8 to test coding standards using flake8
- pytest-timeout to limit runtime in case of stuck tests
- pytest-xdist to run tests in parallel

32.2 Running the tests

Running the tests is simple. Make sure you have pytest installed and run:

```
pytest
```

in the root directory of the repository.

pytest can be configured via a lot of command-line parameters. Some particularly useful ones are:

`-v` or `--verbose`	Be more verbose
`-n NUM`	Run tests in parallel over NUM processes (requires pytest-xdist)
`--timeout=SECONDS`	Set timeout for results from each test process (requires pytest-timeout)
`--capture=no` or `-s`	Do not capture stdout
`--flake8`	Check coding standards using flake8 (requires pytest-flake8)

To run a single test from the command line, you can provide a file path, optionally followed by the function separated by two colons, e.g., (tests do not need to be installed, but Matplotlib should be):

```
pytest lib/matplotlib/tests/test_simplification.py::test_clipping
```

or, if tests are installed, a dot-separated path to the module, optionally followed by the function separated by two colons, such as:

```
pytest --pyargs matplotlib.tests.test_simplification::test_clipping
```

If you want to run the full test suite, but want to save wall time try running the tests in parallel:

```
pytest --verbose -n 5
```

An alternative implementation that does not look at command line arguments and works from within Python is to run the tests from the Matplotlib library function `matplotlib.test()`:

```
import matplotlib
matplotlib.test()
```

32.3 Writing a simple test

Many elements of Matplotlib can be tested using standard tests. For example, here is a test from `matplotlib.tests.test_basic`:

```
def test_simple():
    """
    very simple example test
    """
    assert 1 + 1 == 2
```

Pytest determines which functions are tests by searching for files whose names begin with `"test_"` and then within those files for functions beginning with `"test"` or classes beginning with `"Test"`.

Some tests have internal side effects that need to be cleaned up after their execution (such as created figures or modified *rcParams*). The pytest fixture `mpl_test_settings()` will automatically clean these up; there is no need to do anything further.

32.4 Random data in tests

Random data is a very convenient way to generate data for examples, however the randomness is problematic for testing (as the tests must be deterministic!). To work around this set the seed in each test. For numpy use:

```
import numpy as np
np.random.seed(19680801)
```

and Python's random number generator:

```
import random
random.seed(19680801)
```

The seed is John Hunter's birthday.

32.5 Writing an image comparison test

Writing an image-based test is only slightly more difficult than a simple test. The main consideration is that you must specify the "baseline", or expected, images in the *image_comparison* decorator. For example, this test generates a single image and automatically tests it:

```
from matplotlib.testing.decorators import image_comparison
import matplotlib.pyplot as plt

@image_comparison(baseline_images=['line_dashes'], remove_text=True,
                  extensions=['png'])
def test_line_dashes():
    fig, ax = plt.subplots()
    ax.plot(range(10), linestyle=(0, (3, 3)), lw=5)
```

The first time this test is run, there will be no baseline image to compare against, so the test will fail. Copy the output images (in this case `result_images/test_lines/test_line_dashes.png`) to the correct subdirectory of `baseline_images` tree in the source directory (in this case `lib/matplotlib/tests/baseline_images/test_lines`). Put this new file under source code revision control (with `git add`). When rerunning the tests, they should now pass.

Baseline images take a lot of space in the Matplotlib repository. An alternative approach for image comparison tests is to use the *check_figures_equal* decorator, which should be used to decorate a function taking two *Figure* parameters and draws the same images on the figures using two different methods (the tested method and the

baseline method). The decorator will arrange for setting up the figures and then collect the drawn results and compare them.

See the documentation of *image_comparison* and *check_figures_equal* for additional information about their use.

32.6 Known failing tests

If you're writing a test, you may mark it as a known failing test with the `pytest.mark.xfail()` decorator. This allows the test to be added to the test suite and run on the buildbots without causing undue alarm. For example, although the following test will fail, it is an expected failure:

```
import pytest

@pytest.mark.xfail
def test_simple_fail():
    '''very simple example test that should fail'''
    assert 1 + 1 == 3
```

Note that the first argument to the `xfail()` decorator is a fail condition, which can be a value such as True, False, or may be a dynamically evaluated expression. If a condition is supplied, then a reason must also be supplied with the `reason='message'` keyword argument.

32.7 Creating a new module in matplotlib.tests

We try to keep the tests categorized by the primary module they are testing. For example, the tests related to the `mathtext.py` module are in `test_mathtext.py`.

32.8 Using Travis CI

Travis CI is a hosted CI system "in the cloud".

Travis is configured to receive notifications of new commits to GitHub repos (via GitHub "service hooks") and to run builds or tests when it sees these new commits. It looks for a YAML file called `.travis.yml` in the root of the repository to see how to test the project.

Travis CI is already enabled for the main Matplotlib GitHub repository -- for example, see its Travis page.

If you want to enable Travis CI for your personal Matplotlib GitHub repo, simply enable the repo to use Travis CI in either the Travis CI UI or the GitHub UI (Admin | Service Hooks). For details, see the Travis CI Getting Started page. This generally

isn't necessary, since any pull request submitted against the main Matplotlib repository will be tested.

Once this is configured, you can see the Travis CI results at https://travis-ci.org/your_GitHub_user_name/matplotlib -- here's an example.

32.9 Using tox

Tox is a tool for running tests against multiple Python environments, including multiple versions of Python (e.g., 3.6, 3.7) and even different Python implementations altogether (e.g., CPython, PyPy, Jython, etc.), as long as all these versions are available on your system's $PATH (consider using your system package manager, e.g. apt-get, yum, or Homebrew, to install them).

tox makes it easy to determine if your working copy introduced any regressions before submitting a pull request. Here's how to use it:

```
$ pip install tox
$ tox
```

You can also run tox on a subset of environments:

```
$ tox -e py36,py37
```

Tox processes everything serially so it can take a long time to test several environments. To speed it up, you might try using a new, parallelized version of tox called `detox`. Give this a try:

```
$ pip install -U -i http://pypi.testrun.org detox
$ detox
```

Tox is configured using a file called `tox.ini`. You may need to edit this file if you want to add new environments to test (e.g., `py33`) or if you want to tweak the dependencies or the way the tests are run. For more info on the `tox.ini` file, see the Tox Configuration Specification.

32.10 Building old versions of Matplotlib

When running a `git bisect` to see which commit introduced a certain bug, you may (rarely) need to build very old versions of Matplotlib. The following constraints need to be taken into account:

- Matplotlib 1.3 (or earlier) requires numpy 1.8 (or earlier).

CHAPTER

THIRTYTHREE

WRITING DOCUMENTATION

Contents

33.1 Getting started

33.1.1 General file structure

All documentation is built from the `doc/`, `tutorials/`, and `examples/` directories. The `doc/` directory contains configuration files for Sphinx and reStructuredText (ReST; `.rst`) files that are rendered to documentation pages.

The main entry point is `doc/index.rst`, which pulls in the `index.rst` file for the users guide (`doc/users`), developers guide (`doc/devel`), api reference (`doc/api`), and FAQs (`doc/faq`). The documentation suite is built as a single document in order to make the most effective use of cross referencing.

Sphinx also creates `.rst` files that are staged in `doc/api` from the docstrings of the classes in the Matplotlib library. Except for `doc/api/api_changes/`, these `.rst` files are created when the documentation is built.

Similarly, the contents of `doc/gallery` and `doc/tutorials` are generated by the Sphinx Gallery from the sources in `examples/` and `tutorials/`. These sources consist of python scripts that have ReST documentation built into their comments.

Note: Don't directly edit the `.rst` files in `doc/gallery`, `doc/tutorials`, and `doc/api` (excepting `doc/api/api_changes/`). Sphinx regenerates files in these directories when building documentation.

33.1.2 Installing dependencies

The documentation for Matplotlib is generated from reStructuredText (ReST) using the Sphinx documentation generation tool. To build the documentation you will need to (1) set up an appropriate Python environment and (2) separately install LaTeX and Graphviz.

To (1) set up an appropriate Python environment for building the documentation, you should:

- create a clean virtual environment with no existing Matplotlib installation
- install the Python packages required for Matplotlib
- install the additional Python packages required to build the documentation

There are several extra python packages that are needed to build the documentation. They are listed in `doc-requirements.txt`, which is shown below:

```
# Requirements for building docs
#
# You will first need a matching Matplotlib installation
# e.g (from the Matplotlib root directory)
#     pip install -e .
#
# Install the documentation requirements with:
#     pip install -r requirements/doc/doc-requirements.txt
#
sphinx>=1.8.1,!=2.0.0
colorspacious
ipython
ipywidgets
numpydoc>=0.8
sphinxcontrib-svg2pdfconverter>=1.1.0
sphinx-gallery>=0.7
sphinx-copybutton
scipy
```

To (2) set up LaTeX and Graphviz dependencies you should:

- install a minimal working LaTeX distribution
- install the LaTeX packages cm-super and dvipng
- install Graphviz

Note: The documentation will not build without LaTeX and Graphviz. These are not Python packages and must be installed separately.

33.1.3 Building the docs

The documentation sources are found in the `doc/` directory in the trunk. The configuration file for Sphinx is `doc/conf.py`. It controls which directories Sphinx parses, how the docs are built, and how the extensions are used. To build the documentation in html format, cd into `doc/` and run:

```
make html
```

Other useful invocations include

```
# Delete built files.  May help if you get errors about missing paths or
# broken links.
make clean

# Build pdf docs.
make latexpdf
```

The `SPHINXOPTS` variable is set to `-W --keep-going` by default to build the complete docs but exit with exit status 1 if there are warnings. To unset it, use

```
make SPHINXOPTS= html
```

On Windows the arguments must be at the end of the statement:

```
make html SPHINXOPTS=
```

You can use the `O` variable to set additional options:

- `make O=-j4 html` runs a parallel build with 4 processes.
- `make O=-Dplot_formats=png:100 html` saves figures in low resolution.
- `make O=-Dplot_gallery=0 html` skips the gallery build.

Multiple options can be combined using e.g. `make O='-j4 -Dplot_gallery=0' html`.

On Windows, either use the format shown above or set options as environment variables, e.g.:

```
set O=-W --keep-going -j4
make html
```

33.2 Writing ReST pages

Most documentation is either in the docstring of individual classes and methods, in explicit `.rst` files, or in examples and tutorials. All of these use the ReST syntax. Users should look at the ReST documentation for a full description. But some specific hints and conventions Matplotlib uses are useful for creating documentation.

33.2.1 Formatting and style conventions

It is useful to strive for consistency in the Matplotlib documentation. Here are some formatting and style conventions that are used.

Section name formatting

For everything but top-level chapters, use `Upper lower` for section titles, e.g., `Possible hangups` rather than `Possible Hangups`

Function arguments

Function arguments and keywords within docstrings should be referred to using the `*emphasis*` role. This will keep Matplotlib's documentation consistent with Python's documentation:

```
Here is a description of *argument*
```

Do not use the `` `default role` ``:

```
Do not describe `argument` like this.  As per the next section,
this syntax will (unsuccessfully) attempt to resolve the argument as a
link to a class or method in the library.
```

nor the ``` ``literal`` ``` role:

```
Do not describe ``argument`` like this.
```

33.2.2 Referring to other documents and sections

Sphinx allows internal references between documents.

Documents can be linked with the `:doc:` directive:

```
See the :doc:`/faq/installing_faq`

See the tutorial :doc:`/tutorials/introductory/sample_plots`

See the example :doc:`/gallery/lines_bars_and_markers/simple_plot`
```

will render as:

> See the *Installation*
>
> See the tutorial *Sample plots in Matplotlib*
>
> See the example /gallery/lines_bars_and_markers/simple_plot

Sections can also be given reference names. For instance from the *Installation* link:

```
.. _clean-install:

How to completely remove Matplotlib
===================================

Occasionally, problems with Matplotlib can be solved with a clean...
```

and refer to it using the standard reference syntax:

```
See :ref:`clean-install`
```

will give the following link: *How to completely remove Matplotlib*

To maximize internal consistency in section labeling and references, use hyphen separated, descriptive labels for section references. Keep in mind that contents may be reorganized later, so avoid top level names in references like `user` or `devel` or `faq` unless necessary, because for example the FAQ "what is a backend?" could later become part of the users guide, so the label:

```
.. _what-is-a-backend:
```

is better than:

```
.. _faq-backend:
```

In addition, since underscores are widely used by Sphinx itself, use hyphens to separate words.

33.2.3 Referring to other code

To link to other methods, classes, or modules in Matplotlib you can use back ticks, for example:

```
`matplotlib.collections.LineCollection`
```

generates a link like this: *matplotlib.collections.LineCollection*.

Note: We use the sphinx setting `default_role = 'obj'` so that you don't have to use qualifiers like `:class:`, `:func:`, `:meth:` and the likes.

Often, you don't want to show the full package and module name. As long as the target is unanbigous you can simply leave them out:

```
`.LineCollection`
```

and the link still works: *LineCollection*.

If there are multiple code elements with the same name (e.g. `plot()` is a method in multiple classes), you'll have to extend the definition:

```
`.pyplot.plot` or `.Axes.plot`
```

These will show up as *pyplot.plot* or *Axes.plot*. To still show only the last segment you can add a tilde as prefix:

```
`~.pyplot.plot` or `~.Axes.plot`
```

will render as *plot* or *plot*.

Other packages can also be linked via intersphinx:

```
`numpy.mean`
```

will return this link: numpy.mean. This works for Python, Numpy, Scipy, and Pandas (full list is in `doc/conf.py`). If external linking fails, you can check the full list of referenceable objects with the following commands:

```
python -m sphinx.ext.intersphinx 'https://docs.python.org/3/objects.inv'
python -m sphinx.ext.intersphinx 'https://docs.scipy.org/doc/numpy/objects.inv'
python -m sphinx.ext.intersphinx 'https://docs.scipy.org/doc/scipy/reference/objects.inv'
python -m sphinx.ext.intersphinx 'https://pandas.pydata.org/pandas-docs/stable/objects.
↪inv'
```

33.2.4 Including figures and files

Image files can directly included in pages with the `image::` directive. e.g., `thirdpartypackages/index.rst` displays the images for the third-party packages as static images:

```
.. image:: /_static/toolbar.png
```

as rendered on the page: *Third party packages*.

Files can be included verbatim. For instance the `matplotlibrc` file is important for customizing Matplotlib, and is included verbatim in the tutorial in *Customizing Matplotlib with style sheets and rcParams*:

```
.. literalinclude:: ../../_static/matplotlibrc
```

This is rendered at the bottom of *Customizing Matplotlib with style sheets and rcParams*. Note that this is in a tutorial; see *Writing examples and tutorials* below.

The examples directory is also copied to `doc/gallery` by sphinx-gallery, so plots from the examples directory can be included using

```
.. plot:: gallery/lines_bars_and_markers/simple_plot.py
```

Note that the python script that generates the plot is referred to, rather than any plot that is created. Sphinx-gallery will provide the correct reference when the documentation is built.

33.3 Writing docstrings

Most of the API documentation is written in docstrings. These are comment blocks in source code that explain how the code works.

Note: Some parts of the documentation do not yet conform to the current documentation style. If in doubt, follow the rules given here and not what you may see in the source code. Pull requests updating docstrings to the current style are very welcome.

All new or edited docstrings should conform to the numpydoc docstring guide. Much of the ReST syntax discussed above (*Writing ReST pages*) can be used for links and references. These docstrings eventually populate the `doc/api` directory and form the reference documentation for the library.

33.3.1 Example docstring

An example docstring looks like:

```
def hlines(self, y, xmin, xmax, colors='k', linestyles='solid',
           label='', **kwargs):
    """
    Plot horizontal lines at each *y* from *xmin* to *xmax*.

    Parameters
    ----------
    y : float or array-like
        y-indexes where to plot the lines.

    xmin, xmax : float or array-like
        Respective beginning and end of each line. If scalars are
        provided, all lines will have the same length.

    colors : array-like of colors, default: 'k'

    linestyles : {'solid', 'dashed', 'dashdot', 'dotted'}, default: 'solid'

    label : str, default: ''

    Returns
    -------
    lines : `~matplotlib.collections.LineCollection`

    Other Parameters
    ----------------
    **kwargs : `~matplotlib.collections.LineCollection` properties

    See also
    --------
    vlines : vertical lines
    axhline: horizontal line across the axes
    """
```

See the *hlines* documentation for how this renders.

The Sphinx website also contains plenty of documentation concerning ReST markup and working with Sphinx in general.

33.3.2 Formatting conventions

The basic docstring conventions are covered in the numpydoc docstring guide and the Sphinx documentation. Some Matplotlib-specific formatting conventions to keep in mind:

Quote positions

The quotes for single line docstrings are on the same line (pydocstyle D200):

```
def get_linewidth(self):
    """Return the line width in points."""
```

The quotes for multi-line docstrings are on separate lines (pydocstyle D213):

```
def set_linestyle(self, ls):
"""
Set the linestyle of the line.

[...]
"""
```

Function arguments

Function arguments and keywords within docstrings should be referred to using the `*emphasis*` role. This will keep Matplotlib's documentation consistent with Python's documentation:

```
If *linestyles* is *None*, the default is 'solid'.
```

Do not use the `` `default role` `` or the ``` ``literal`` ``` role:

```
Neither `argument` nor ``argument`` should be used.
```

Quotes for strings

Matplotlib does not have a convention whether to use single-quotes or double-quotes. There is a mixture of both in the current code.

Use simple single or double quotes when giving string values, e.g.

```
If 'tight', try to figure out the tight bbox of the figure.

No ``'extra'`` literal quotes.
```

The use of extra literal quotes around the text is discouraged. While they slightly improve the rendered docs, they are cumbersome to type and difficult to read in plain-text docs.

Parameter type descriptions

The main goal for parameter type descriptions is to be readable and understandable by humans. If the possible types are too complex use a simplification for the type description and explain the type more precisely in the text.

Generally, the numpydoc docstring guide conventions apply. The following rules expand on them where the numpydoc conventions are not specific.

Use `float` for a type that can be any number.

Use `(float, float)` to describe a 2D position. The parentheses should be included to make the tuple-ness more obvious.

Use `array-like` for homogeneous numeric sequences, which could typically be a numpy.array. Dimensionality may be specified using `2D`, `3D`, `n-dimensional`. If you need to have variables denoting the sizes of the dimensions, use capital letters in brackets (`array-like (M, N)`). When referring to them in the text they are easier read and no special formatting is needed.

`float` is the implicit default dtype for array-likes. For other dtypes use `array-like of int`.

Some possible uses:

```
2D array-like
array-like (N)
array-like (M, N)
array-like (M, N, 3)
array-like of int
```

Non-numeric homogeneous sequences are described as lists, e.g.:

```
list of str
list of `.Artist`
```

Referencing types

Generally, the rules from *referring-to-other-code* apply. More specifically:

Use full references `` `~matplotlib.colors.Normalize` `` with an abbreviation tilde in parameter types. While the full name helps the reader of plain text docstrings, the HTML does not need to show the full name as it links to it. Hence, the ~-shortening keeps it more readable.

Use abbreviated links `` `.Normalize` `` in the text.

```
norm : `~matplotlib.colors.Normalize`, optional
     A `.Normalize` instance is used to scale luminance data to 0, 1.
```

Default values

As opposed to the numpydoc guide, parameters need not be marked as *optional* if they have a simple default:

- use `{name} : {type}, default: {val}` when possible.
- use `{name} : {type}, optional` and describe the default in the text if it cannot be explained sufficiently in the recommended manner.

The default value should provide semantic information targeted at a human reader. In simple cases, it restates the value in the function signature. If applicable, units should be added.

```
Prefer:
    interval : int, default: 1000ms
over:
    interval : int, default: 1000
```

If *None* is only used as a sentinel value for "parameter not specified", do not document it as the default. Depending on the context, give the actual default, or mark the parameter as optional if not specifying has no particular effect.

```
Prefer:
    dpi : float, default: :rc:`figure.dpi`
over:
    dpi : float, default: None

Prefer:
    textprops : dict, optional
        Dictionary of keyword parameters to be passed to the
        `~matplotlib.text.Text` instance contained inside TextArea.
over:
    textprops : dict, default: None
        Dictionary of keyword parameters to be passed to the
        `~matplotlib.text.Text` instance contained inside TextArea.
```

`See also` sections

Sphinx automatically links code elements in the definition blocks of `See also` sections. No need to use backticks there:

```
See also
--------
vlines : vertical lines
axhline: horizontal line across the axes
```

Wrapping parameter lists

Long parameter lists should be wrapped using a \ for continuation and starting on the new line without any indent (no indent because pydoc will parse the docstring and strip the line continuation so that indent would result in a lot of whitespace within the line):

```
def add_axes(self, *args, **kwargs):
    """
    ...

    Parameters
    ----------
    projection : {'aitoff', 'hammer', 'lambert', 'mollweide', 'polar', \
'rectilinear'}, optional
        The projection type of the axes.

    ...
    """
```

Alternatively, you can describe the valid parameter values in a dedicated section of the docstring.

rcParams

rcParams can be referenced with the custom `:rc:` role: `:rc:`foo`` yields `rcParams["foo"] = 'default'`, which is a link to the `matplotlibrc` file description.

33.3.3 Setters and getters

Artist properties are implemented using setter and getter methods (because Matplotlib predates the introductions of the `property` decorator in Python). By convention, these setters and getters are named `set_PROPERTYNAME` and `get_PROPERTYNAME`; the list of properties thusly defined on an artist and their values can be listed by the *setp* and *getp* functions.

The Parameters block of property setter methods is parsed to document the accepted values, e.g. the docstring of *Line2D.set_linestyle* starts with

```
def set_linestyle(self, ls):
    """
    Set the linestyle of the line.

    Parameters
    ----------
    ls : {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
        etc.
    """
```

which results in the following line in the output of `plt.setp(line)` or `plt.setp(line, "linestyle")`:

```
linestyle or ls: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
```

In some rare cases (mostly, setters which accept both a single tuple and an unpacked tuple), the accepted values cannot be documented in such a fashion; in that case, they can be documented as an `.. ACCEPTS:` block, e.g. for *axes.Axes.set_xlim*:

```
def set_xlim(self, ...):
    """
    Set the x-axis view limits.

    Parameters
    ----------
    left : float, optional
        The left xlim in data coordinates. Passing *None* leaves the
        limit unchanged.

        The left and right xlims may also be passed as the tuple
        (*left*, *right*) as the first positional argument (or as
        the *left* keyword argument).

        .. ACCEPTS: (bottom: float, top: float)

    right : float, optional
        etc.
    """
```

Note that the leading `..` makes the `.. ACCEPTS:` block a reST comment, hiding it from the rendered docs.

33.3.4 Keyword arguments

Note: The information in this section is being actively discussed by the development team, so use the docstring interpolation only if necessary. This section has been left in place for now because this interpolation is part of the existing documentation.

Since Matplotlib uses a lot of pass-through `kwargs`, e.g., in every function that creates a line (*plot*, *semilogx*, *semilogy*, etc...), it can be difficult for the new user to know which `kwargs` are supported. Matplotlib uses a docstring interpolation scheme to support documentation of every function that takes a `**kwargs`. The requirements are:

1. single point of configuration so changes to the properties don't require multiple docstring edits.
2. as automated as possible so that as properties change, the docs are updated automatically.

The function *matplotlib.artist.kwdoc* and the decorator `matplotlib.docstring.dedent_interpd` facilitate this. They combine Python string interpolation in the docstring with the Matplotlib artist introspection facility that underlies `setp` and `getp`. The `kwdoc` function gives the list of properties as a docstring. In order to use this in another docstring, first update the `matplotlib.docstring.interpd` object, as seen in this example from *matplotlib.lines*:

```
# in lines.py
docstring.interpd.update(Line2D=artist.kwdoc(Line2D))
```

Then in any function accepting *Line2D* pass-through `kwargs`, e.g., *matplotlib.axes.Axes.plot*:

```
# in axes.py
@docstring.dedent_interpd
def plot(self, *args, **kwargs):
    """
    Some stuff omitted

    The kwargs are Line2D properties:
    %(_Line2D_docstr)s

    kwargs scalex and scaley, if defined, are passed on
    to autoscale_view to determine whether the x and y axes are
    autoscaled; default True.  See Axes.autoscale_view for more
    information
    """
```

Note there is a problem for *Artist* `__init__` methods, e.g., *matplotlib.patches.Patch.__init__*, which supports `Patch` `kwargs`, since the artist inspector cannot work until the class is fully defined and we can't modify the `Patch.__init__.__doc__` docstring outside the class definition. There are some some manual hacks in this case, violating the "single entry point" requirement above -- see the `docstring.interpd.update` calls in *matplotlib.patches*.

33.3.5 Inheriting docstrings

If a subclass overrides a method but does not change the semantics, we can reuse the parent docstring for the method of the child class. Python does this automatically, if the subclass method does not have a docstring.

Use a plain comment `# docstring inherited` to denote the intention to reuse the parent docstring. That way we do not accidentally create a docstring in the future:

```
class A:
    def foo():
        """The parent docstring."""
        pass
```

(continues on next page)

(continued from previous page)

```
class B(A):
    def foo():
        # docstring inherited
        pass
```

33.3.6 Adding figures

As above (see *Including figures and files*), figures in the examples gallery can be referenced with a `:plot:` directive pointing to the python script that created the figure. For instance the *legend* docstring references the file `examples/text_labels_and_annotations/legend.py`:

```
"""
...

Examples
--------

.. plot:: gallery/text_labels_and_annotations/legend.py
"""
```

Note that `examples/text_labels_and_annotations/legend.py` has been mapped to `gallery/text_labels_and_annotations/legend.py`, a redirection that may be fixed in future re-organization of the docs.

Plots can also be directly placed inside docstrings. Details are in *matplotlib.sphinxext.plot_directive*. A short example is:

```
"""
...

Examples
--------

.. plot::
   import matplotlib.image as mpimg
   img = mpimg.imread('_static/stinkbug.png')
   imgplot = plt.imshow(img)
"""
```

An advantage of this style over referencing an example script is that the code will also appear in interactive docstrings.

33.4 Writing examples and tutorials

Examples and tutorials are python scripts that are run by Sphinx Gallery to create a gallery of images in the `/doc/gallery` and `/doc/tutorials` directories respectively. To exclude an example from having an plot generated insert "sgskip" somewhere in the filename.

The format of these files is relatively straightforward. Properly formatted comment blocks are treated as ReST text, the code is displayed, and figures are put into the built page.

For instance the example /gallery/lines_bars_and_markers/simple_plot example is generated from `/examples/lines_bars_and_markers/simple_plot.py`, which looks like:

```
"""
===========
Simple Plot
===========

Create a simple plot.
"""
import matplotlib.pyplot as plt
import numpy as np

# Data for plotting
t = np.arange(0.0, 2.0, 0.01)
s = 1 + np.sin(2 * np.pi * t)

# Note that using plt.subplots below is equivalent to using
# fig = plt.figure and then ax = fig.add_subplot(111)
fig, ax = plt.subplots()
ax.plot(t, s)

ax.set(xlabel='time (s)', ylabel='voltage (mV)',
       title='About as simple as it gets, folks')
ax.grid()
plt.show()
```

The first comment block is treated as ReST text. The other comment blocks render as comments in /gallery/lines_bars_and_markers/simple_plot.

Tutorials are made with the exact same mechanism, except they are longer, and typically have more than one comment block (i.e. *Usage Guide*). The first comment block can be the same as the example above. Subsequent blocks of ReST text are delimited by a line of `###` characters:

```
"""
===========
Simple Plot
===========

```

(continues on next page)

(continued from previous page)

```
Create a simple plot.
"""
...
ax.grid()
plt.show()

########################################################################
# Second plot
# ===========
#
# This is a second plot that is very nice

fig, ax = plt.subplots()
ax.plot(np.sin(range(50)))
```

In this way text, code, and figures are output in a "notebook" style.

33.4.1 Order of examples in the gallery

The order of the sections of the *Tutorials* and the gallery, as well as the order of the examples within each section are determined in a two step process from within the `/doc/sphinxext/gallery_order.py`:

- *Explicit order*: This file contains a list of folders for the section order and a list of examples for the subsection order. The order of the items shown in the doc pages is the order those items appear in those lists.
- *Implicit order*: If a folder or example is not in those lists, it will be appended after the explicitly ordered items and all of those additional items will be ordered by pathname (for the sections) or by filename (for the subsections).

As a consequence, if you want to let your example appear in a certain position in the gallery, extend those lists with your example. In case no explicit order is desired or necessary, still make sure to name your example consistently, i.e. use the main function or subject of the example as first word in the filename; e.g. an image example should ideally be named similar to `imshow_mynewexample.py`.

33.5 Miscellaneous

33.5.1 Adding animations

Animations are scraped automatically by Sphinx-gallery. If this is not desired, there is also a Matplotlib Google/Gmail account with username `mplgithub` which was used to setup the github account but can be used for other purposes, like hosting Google docs or Youtube videos. You can embed a Matplotlib animation in the docs by first saving the animation as a movie using *matplotlib.animation.Animation.save()*, and

then uploading to Matplotlib's Youtube channel and inserting the embedding string youtube provides like:

```
.. raw:: html

   <iframe width="420" height="315"
     src="http://www.youtube.com/embed/32cjc6V0OZY"
     frameborder="0" allowfullscreen>
   </iframe>
```

An example save command to generate a movie looks like this

```
ani = animation.FuncAnimation(fig, animate, np.arange(1, len(y)),
    interval=25, blit=True, init_func=init)

ani.save('double_pendulum.mp4', fps=15)
```

Contact Michael Droettboom for the login password to upload youtube videos of google docs to the mplgithub account.

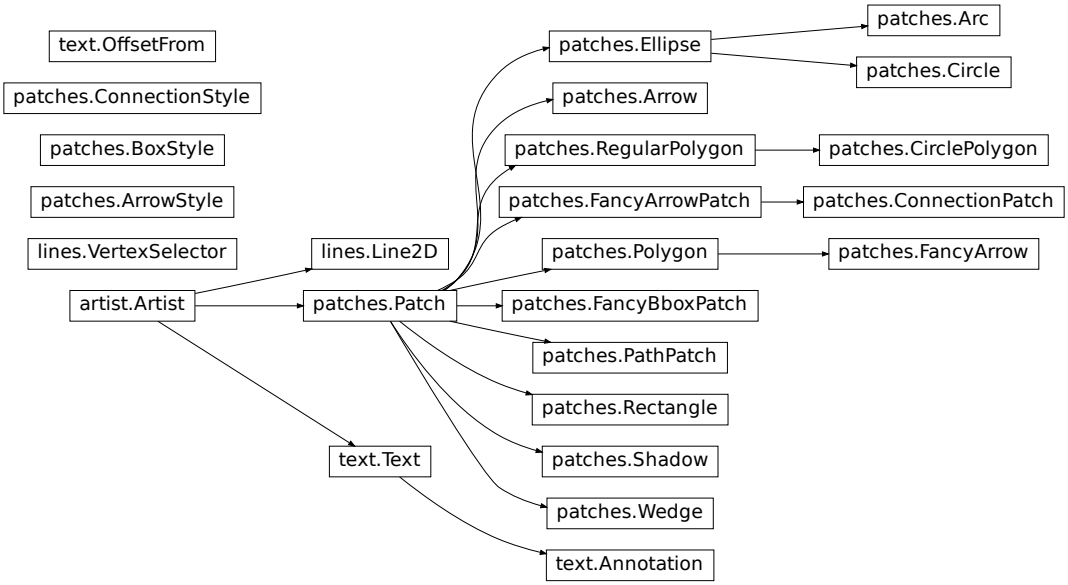
33.5.2 Generating inheritance diagrams

Class inheritance diagrams can be generated with the `inheritance-diagram` directive. To use it, provide the directive with a number of class or module names (separated by whitespace). If a module name is provided, all classes in that module will be used. All of the ancestors of these classes will be included in the inheritance diagram.

A single option is available: *parts* controls how many of parts in the path to the class are shown. For example, if *parts* == 1, the class `matplotlib.patches.Patch` is shown as `Patch`. If *parts* == 2, it is shown as `patches.Patch`. If *parts* == 0, the full path is shown.

Example:

```
.. inheritance-diagram:: matplotlib.patches matplotlib.lines matplotlib.text
   :parts: 2
```



33.5.3 Emacs helpers

There is an emacs mode rst.el which automates many important ReST tasks like building and updating table-of-contents, and promoting or demoting section headings. Here is the basic `.emacs` configuration:

```
(require 'rst)
(setq auto-mode-alist
      (append '(("\\.txt$" . rst-mode)
                ("\\.rst$" . rst-mode)
                ("\\.rest$" . rst-mode)) auto-mode-alist))
```

Some helpful functions:

```
C-c TAB - rst-toc-insert

  Insert table of contents at point

C-c C-u - rst-toc-update

    Update the table of contents at point

C-c C-l rst-shift-region-left

    Shift region to the left
```

(continues on next page)

(continued from previous page)

```
C-c C-r rst-shift-region-right

    Shift region to the right
```

CHAPTER

THIRTYFOUR

DEVELOPER'S GUIDE FOR CREATING SCALES AND TRANSFORMATIONS

Matplotlib supports the addition of custom procedures that transform the data before it is displayed.

There is an important distinction between two kinds of transformations. Separable transformations, working on a single dimension, are called "scales", and non-separable transformations, that handle data in two or more dimensions at a time, are called "projections".

From the user's perspective, the scale of a plot can be set with *Axes.set_xscale* and *Axes.set_yscale*. Projections can be chosen using the *projection* keyword argument of functions that create Axes, such as *pyplot.subplot* or *pyplot.axes*, e.g.

```
plt.subplot(projection="custom")
```

This document is intended for developers and advanced users who need to create new scales and projections for Matplotlib. The necessary code for scales and projections can be included anywhere: directly within a plot script, in third-party code, or in the Matplotlib source tree itself.

34.1 Creating a new scale

Adding a new scale consists of defining a subclass of *matplotlib.scale.ScaleBase*, that includes the following elements:

- A transformation from data coordinates into display coordinates.
- An inverse of that transformation. This is used, for example, to convert mouse positions from screen space back into data space.
- A function to limit the range of the axis to acceptable values (`limit_range_for_scale()`). A log scale, for instance, would prevent the range from including values less than or equal to zero.
- Locators (major and minor) that determine where to place ticks in the plot, and optionally, how to adjust the limits of the plot to some "good" values. Unlike

`limit_range_for_scale()`, which is always enforced, the range setting here is only used when automatically setting the range of the plot.

- Formatters (major and minor) that specify how the tick labels should be drawn.

Once the class is defined, it must be registered with Matplotlib so that the user can select it.

A full-fledged and heavily annotated example is in /gallery/scales/custom_scale. There are also some classes in *matplotlib.scale* that may be used as starting points.

34.2 Creating a new projection

Adding a new projection consists of defining a projection axes which subclasses *matplotlib.axes.Axes* and includes the following elements:

- A transformation from data coordinates into display coordinates.
- An inverse of that transformation. This is used, for example, to convert mouse positions from screen space back into data space.
- Transformations for the gridlines, ticks and ticklabels. Custom projections will often need to place these elements in special locations, and Matplotlib has a facility to help with doing so.
- Setting up default values (overriding *cla()*), since the defaults for a rectilinear axes may not be appropriate.
- Defining the shape of the axes, for example, an elliptical axes, that will be used to draw the background of the plot and for clipping any data elements.
- Defining custom locators and formatters for the projection. For example, in a geographic projection, it may be more convenient to display the grid in degrees, even if the data is in radians.
- Set up interactive panning and zooming. This is left as an "advanced" feature left to the reader, but there is an example of this for polar plots in *matplotlib.projections.polar*.
- Any additional methods for additional convenience or features.

Once the projection axes is defined, it can be used in one of two ways:

- By defining the class attribute `name`, the projection axes can be registered with *matplotlib.projections.register_projection()* and subsequently simply invoked by name:

```
plt.axes(projection='my_proj_name')
```

- For more complex, parameterisable projections, a generic "projection" object may be defined which includes the method `_as_mpl_axes`. `_as_mpl_axes` should take no arguments and return the projection's axes subclass and a dictionary of

additional arguments to pass to the subclass' `__init__` method. Subsequently a parameterised projection can be initialised with:

```
plt.axes(projection=MyProjection(param1=param1_value))
```

where MyProjection is an object which implements a `_as_mpl_axes` method.

A full-fledged and heavily annotated example is in /gallery/misc/custom_projection. The polar plot functionality in *matplotlib.projections.polar* may also be of interest.

34.3 API documentation

- *matplotlib.scale*
- *matplotlib.projections*
- *matplotlib.projections.polar*

CHAPTER

THIRTYFIVE

WORKING WITH *MATPLOTLIB* SOURCE CODE

Contents:

35.1 Introduction

These pages describe a git and github workflow for the Matplotlib project.

There are several different workflows here, for different ways of working with *Matplotlib*.

This is not a comprehensive git reference, it's just a workflow for our own project. It's tailored to the github hosting service. You may well find better or quicker ways of getting stuff done with git, but these should get you started.

For general resources for learning git, see *git resources*.

35.2 Install git

35.2.1 Overview

Debian / Ubuntu	`sudo apt-get install git`
Fedora	`sudo yum install git`
Windows	Download and install msysGit
OS X	Use the git-osx-installer

35.2.2 In detail

See the git page for the most recent information.

Have a look at the github install help pages available from github help

There are good instructions here: https://git-scm.com/book/en/v2/Getting-Started-Installing-Git

35.3 Following the latest source

These are the instructions if you just want to follow the latest *Matplotlib* source, but you don't need to do any development for now.

The steps are:

- *Install git*
- get local copy of the Matplotlib github git repository
- update local copy from time to time

35.3.1 Get the local copy of the code

From the command line:

```
git clone git://github.com/matplotlib/matplotlib.git
```

You now have a copy of the code tree in the new `matplotlib` directory.

35.3.2 Updating the code

From time to time you may want to pull down the latest code. Do this with:

```
cd matplotlib
git pull
```

The tree in `matplotlib` will now have the latest changes from the initial repository.

35.4 Making a patch

You've discovered a bug or something else you want to change in Matplotlib .. — excellent!

You've worked out a way to fix it — even better!

You want to tell us about it — best of all!

The easiest way is to make a *patch* or set of patches. Here we explain how. Making a patch is the simplest and quickest, but if you're going to be doing anything more than simple quick things, please consider following the *Git for development* model instead.

35.4.1 Making patches

Overview

```
# tell git who you are
git config --global user.email you@yourdomain.example.com
git config --global user.name "Your Name Comes Here"
# get the repository if you don't have it
git clone git://github.com/matplotlib/matplotlib.git
# make a branch for your patching
cd matplotlib
git branch the-fix-im-thinking-of
git checkout the-fix-im-thinking-of
# hack, hack, hack
# Tell git about any new files you've made
git add somewhere/tests/test_my_bug.py
# commit work in progress as you go
git commit -am 'BF - added tests for Funny bug'
# hack hack, hack
git commit -am 'BF - added fix for Funny bug'
# make the patch files
git format-patch -M -C master
```

Then, send the generated patch files to the Matplotlib mailing list — where we will thank you warmly.

In detail

1. Tell git who you are so it can label the commits you've made:

```
git config --global user.email you@yourdomain.example.com
git config --global user.name "Your Name Comes Here"
```

2. If you don't already have one, clone a copy of the Matplotlib repository:

```
git clone git://github.com/matplotlib/matplotlib.git
cd matplotlib
```

3. Make a 'feature branch'. This will be where you work on your bug fix. It's nice and safe and leaves you with access to an unmodified copy of the code in the main branch:

```
git branch the-fix-im-thinking-of
git checkout the-fix-im-thinking-of
```

4. Do some edits, and commit them as you go:

```
# hack, hack, hack
# Tell git about any new files you've made
git add somewhere/tests/test_my_bug.py
# commit work in progress as you go
git commit -am 'BF - added tests for Funny bug'
# hack hack, hack
git commit -am 'BF - added fix for Funny bug'
```

   Note the `-am` options to `commit`. The `m` flag just signals that you're going to type a message on the command line. The `a` flag — you can just take on faith — or see why the -a flag?.

5. When you have finished, check you have committed all your changes:

```
git status
```

6. Finally, make your commits into patches. You want all the commits since you branched from the `master` branch:

```
git format-patch -M -C master
```

   You will now have several files named for the commits:

```
0001-BF-added-tests-for-Funny-bug.patch
0002-BF-added-fix-for-Funny-bug.patch
```

   Send these files to the Matplotlib mailing list.

When you are done, to switch back to the main copy of the code, just return to the `master` branch:

```
git checkout master
```

35.4.2 Moving from patching to development

If you find you have done some patches, and you have one or more feature branches, you will probably want to switch to development mode. You can do this with the repository you have.

Fork the Matplotlib repository on github — *Making your own copy (fork) of Matplotlib*. Then:

```
# checkout and refresh master branch from main repo
git checkout master
git pull origin master
# rename pointer to main repository to 'upstream'
git remote rename origin upstream
# point your repo to default read / write to your fork on github
git remote add origin git@github.com:your-user-name/matplotlib.git
# push up any branches you've made and want to keep
git push origin the-fix-im-thinking-of
```

Then you can, if you want, follow the *Development workflow*.

35.5 Git for development

Contents:

35.5.1 Making your own copy (fork) of Matplotlib

You need to do this only once. The instructions here are very similar to the instructions at https://help.github.com/forking/ — please see that page for more detail. We're repeating some of it here just to give the specifics for the Matplotlib project, and to suggest some default names.

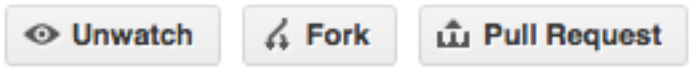
Set up and configure a github account

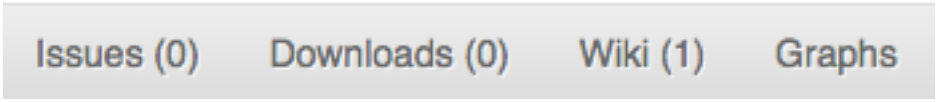
If you don't have a github account, go to the github page, and make one.

You then need to configure your account to allow write access — see the `Generating SSH keys` help on github help.

Create your own forked copy of Matplotlib

1. Log into your github account.
2. Go to the Matplotlib github home at Matplotlib github.
3. Click on the *fork* button:





Now, after a short pause, you should find yourself at the home page for your own forked copy of Matplotlib.

35.5.2 Set up your fork

First you follow the instructions for *Making your own copy (fork) of Matplotlib*.

Overview

```
git clone https://github.com/your-user-name/matplotlib.git
cd matplotlib
git remote add upstream git://github.com/matplotlib/matplotlib.git
```

In detail

Clone your fork

1. Clone your fork to the local computer with `git clone https://github.com/your-user-name/matplotlib.git`
2. Investigate. Change directory to your new repo: `cd matplotlib`. Then `git branch -a` to show you all branches. You'll get something like:

```
* master
remotes/origin/master
```

This tells you that you are currently on the `master` branch, and that you also have a `remote` connection to `origin/master`. What remote repository is `remote/origin`? Try `git remote -v` to see the URLs for the remote. They will point to your github fork.

Now you want to connect to the upstream Matplotlib github repository, so you can merge in changes from trunk.

Linking your repository to the upstream repo

```
cd matplotlib
git remote add upstream git://github.com/matplotlib/matplotlib.git
```

`upstream` here is just the arbitrary name we're using to refer to the main Matplotlib repository at Matplotlib github.

Note that we've used `git://` for the URL rather than `https://` or `git@`. The `git://` URL is read only. This means that we can't accidentally (or deliberately) write to the upstream repo, and we are only going to use it to merge into our own code.

Just for your own satisfaction, show yourself that you now have a new 'remote', with `git remote -v show`, giving you something like:

```
upstream     git://github.com/matplotlib/matplotlib.git (fetch)
upstream     git://github.com/matplotlib/matplotlib.git (push)
origin       https://github.com/your-user-name/matplotlib.git (fetch)
origin       https://github.com/your-user-name/matplotlib.git (push)
```

35.5.3 Configure git

Overview

Your personal git configurations are saved in the `.gitconfig` file in your home directory.

Here is an example `.gitconfig` file:

```
[user]
        name = Your Name
        email = you@yourdomain.example.com

[alias]
        ci = commit -a
        co = checkout
        st = status
        stat = status
        br = branch
        wdiff = diff --color-words

[core]
        editor = vim

[merge]
        summary = true
```

You can check what is already in your config file using the `git config --list` command. You can edit the `.gitconfig` file directly or you can use the `git config --global` command.:

```
git config --global user.name "Your Name"
git config --global user.email you@yourdomain.example.com
git config --global alias.ci "commit -a"
git config --global alias.co checkout
git config --global alias.st "status -a"
git config --global alias.stat "status -a"
git config --global alias.br branch
git config --global alias.wdiff "diff --color-words"
git config --global core.editor vim
git config --global merge.summary true
```

To set up on another computer, you can copy your `~/.gitconfig` file, or run the commands above.

In detail

user.name and user.email

It is good practice to tell git who you are, for labeling any changes you make to the code. The simplest way to do this is from the command line:

```
git config --global user.name "Your Name"
git config --global user.email you@yourdomain.example.com
```

This will write the settings into your git configuration file, which should now contain a user section with your name and email:

```
[user]
      name = Your Name
      email = you@yourdomain.example.com
```

You'll need to replace `Your Name` and `you@yourdomain.example.com` with your actual name and email address.

Aliases

You might well benefit from some aliases to common commands.

For example, you might well want to be able to shorten `git checkout` to `git co`. Or you may want to alias `git diff --color-words` (which gives a nicely formatted output of the diff) to `git wdiff`

The following `git config --global` commands:

```
git config --global alias.ci "commit -a"
git config --global alias.co checkout
git config --global alias.st "status -a"
git config --global alias.stat "status -a"
git config --global alias.br branch
git config --global alias.wdiff "diff --color-words"
```

will create an `alias` section in your `.gitconfig` file with contents like this:

```
[alias]
        ci = commit -a
        co = checkout
        st = status -a
        stat = status -a
        br = branch
        wdiff = diff --color-words
```

Editor

You may also want to make sure that your editor of choice is used

```
git config --global core.editor vim
```

Merging

To enforce summaries when doing merges (`~/.gitconfig` file again):

```
[merge]
   log = true
```

Or from the command line:

```
git config --global merge.log true
```

Fancy log output

This is a very nice alias to get a fancy log output; it should go in the `alias` section of your `.gitconfig` file:

```
lg = log --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr)
↪%C(bold blue)[%an]%Creset' --abbrev-commit --date=relative
```

You use the alias with:

```
git lg
```

and it gives graph / text output something like this (but with color!):

```
* 6d8e1ee - (HEAD, origin/my-fancy-feature, my-fancy-feature) NF - a fancy file (45␣
↪minutes ago) [Matthew Brett]
*   d304a73 - (origin/placeholder, placeholder) Merge pull request #48 from hhuuggoo/
↪master (2 weeks ago) [Jonathan Terhorst]
|\
| * 4aff2a8 - fixed bug 35, and added a test in test_bugfixes (2 weeks ago) [Hugo]
|/
* a7ff2e5 - Added notes on discussion/proposal made during Data Array Summit. (2 weeks␣
↪ago) [Corran Webster]
* 68f6752 - Initial implementation of AxisIndexer - uses 'index_by' which needs to be␣
↪changed to a call on an Axes object - this is all very sketchy right now. (2 weeks␣
↪ago) [Corr
*   376adbd - Merge pull request #46 from terhorst/master (2 weeks ago) [Jonathan␣
↪Terhorst]
|\
| * b605216 - updated joshu example to current api (3 weeks ago) [Jonathan Terhorst]
| * 2e991e8 - add testing for outer ufunc (3 weeks ago) [Jonathan Terhorst]
| * 7beda5a - prevent axis from throwing an exception if testing equality with non-axis␣
↪object (3 weeks ago) [Jonathan Terhorst]
| * 65af65e - convert unit testing code to assertions (3 weeks ago) [Jonathan Terhorst]
| *   956fbab - Merge remote-tracking branch 'upstream/master' (3 weeks ago) [Jonathan␣
↪Terhorst]
| |\
| |/
```

Thanks to Yury V. Zaytsev for posting it.

35.5.4 Development workflow

You already have your own forked copy of the Matplotlib repository, by following *Making your own copy (fork) of Matplotlib*. You have *Set up your fork*. You have configured git by following *Configure git*. Now you are ready for some real work.

Workflow summary

In what follows we'll refer to the upstream Matplotlib `master` branch, as "trunk".

- Don't use your `master` branch for anything. Consider deleting it.
- When you are starting a new set of changes, fetch any changes from trunk, and start a new *feature branch* from that.
- Make a new branch for each separable set of changes — "one task, one branch" (ipython git workflow).
- Name your branch for the purpose of the changes - e.g. `bugfix-for-issue-14` or `refactor-database-code`.

- If you can possibly avoid it, avoid merging trunk or any other branches into your feature branch while you are working.
- If you do find yourself merging from trunk, consider *Rebasing on trunk*
- Ask on the Matplotlib mailing list if you get stuck.
- Ask for code review!

This way of working helps to keep work well organized, with readable history. This in turn makes it easier for project maintainers (that might be you) to see what you've done, and why you did it.

See linux git workflow and ipython git workflow for some explanation.

Consider deleting your master branch

It may sound strange, but deleting your own `master` branch can help reduce confusion about which branch you are on. See deleting master on github for details.

Update the mirror of trunk

First make sure you have done *Linking your repository to the upstream repo*.

From time to time you should fetch the upstream (trunk) changes from github:

```
git fetch upstream
```

This will pull down any commits you don't have, and set the remote branches to point to the right commit. For example, 'trunk' is the branch referred to by (remote/branchname) `upstream/master` - and if there have been commits since you last checked, `upstream/master` will change after you do the fetch.

Make a new feature branch

When you are ready to make some changes to the code, you should start a new branch. Branches that are for a collection of related edits are often called 'feature branches'.

Making an new branch for each set of related changes will make it easier for someone reviewing your branch to see what you are doing.

Choose an informative name for the branch to remind yourself and the rest of us what the changes in the branch are for. For example `add-ability-to-fly`, or `buxfix-for-issue-42`.

```
# Update the mirror of trunk
git fetch upstream
# Make new feature branch starting at current trunk
git branch my-new-feature upstream/master
git checkout my-new-feature
```

Generally, you will want to keep your feature branches on your public github fork of Matplotlib. To do this, you git push this new branch up to your github repo. Generally (if you followed the instructions in these pages, and by default), git will have a link to your github repo, called `origin`. You push up to your own repo on github with:

```
git push origin my-new-feature
```

In git >= 1.7 you can ensure that the link is correctly set by using the `--set-upstream` option:

```
git push --set-upstream origin my-new-feature
```

From now on git will know that `my-new-feature` is related to the `my-new-feature` branch in the github repo.

The editing workflow

Overview

```
# hack hack
git add my_new_file
git commit -am 'NF - some message'
git push
```

In more detail

1. Make some changes
2. See which files have changed with `git status` (see git status). You'll see a listing like this one:

   ```
   # On branch ny-new-feature
   # Changed but not updated:
   #   (use "git add <file>..." to update what will be committed)
   #   (use "git checkout -- <file>..." to discard changes in working directory)
   #
   #  modified:   README
   #
   # Untracked files:
   #   (use "git add <file>..." to include in what will be committed)
   #
   #  INSTALL
   no changes added to commit (use "git add" and/or "git commit -a")
   ```

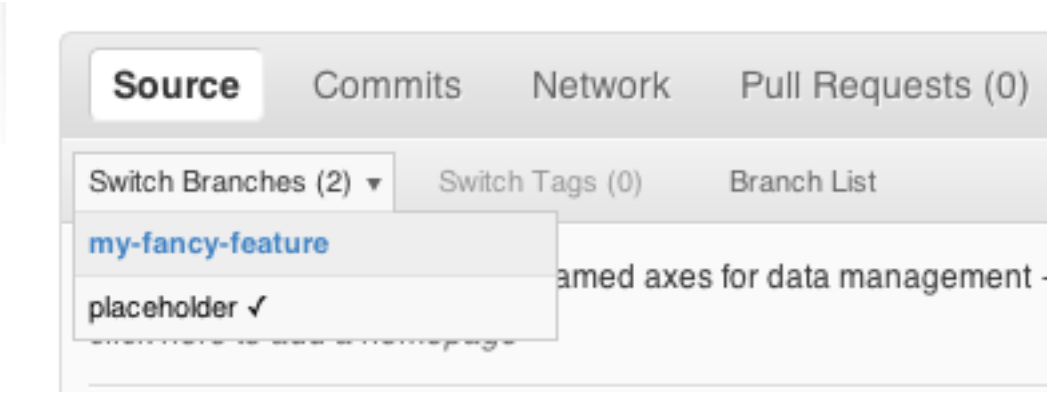
3. Check what the actual changes are with `git diff` (git diff).
4. Add any new files to version control `git add new_file_name` (see git add).

5. To commit all modified files into the local copy of your repo,, do `git commit -am 'A commit message'`. Note the `-am` options to `commit`. The `m` flag just signals that you're going to type a message on the command line. The `a` flag — you can just take on faith — or see why the -a flag? — and the helpful use-case description in the tangled working copy problem. The git commit manual page might also be useful.
6. To push the changes up to your forked repo on github, do a `git push` (see git push).

Ask for your changes to be reviewed or merged

When you are ready to ask for someone to review your code and consider a merge:

1. Go to the URL of your forked repo, say `https://github.com/your-user-name/matplotlib`.
2. Use the 'Switch Branches' dropdown menu near the top left of the page to select the branch with your changes:



3. Click on the 'Pull request' button:

Enter a title for the set of changes, and some explanation of what you've done. Say if there is anything you'd like particular attention for - like a complicated change or some code you are not happy with.

If you don't think your request is ready to be merged, just say so in your pull request message. This is still a good way of getting some preliminary code review.

Some other things you might want to do

Delete a branch on github

```
git checkout master
# delete branch locally
git branch -D my-unwanted-branch
# delete branch on github
git push origin :my-unwanted-branch
```

Note the colon : before `my-unwanted-branch`. See also: https://help.github.com/articles/pushing-to-a-remote/#deleting-a-remote-branch-or-tag

Several people sharing a single repository

If you want to work on some stuff with other people, where you are all committing into the same repository, or even the same branch, then just share it via github.

First fork Matplotlib into your account, as from *Making your own copy (fork) of Matplotlib*.

Then, go to your forked repository github page, say `https://github.com/your-user-name/matplotlib`

Click on the 'Admin' button, and add anyone else to the repo as a collaborator:

Now all those people can do:

```
git clone https://github.com/your-user-name/matplotlib.git
```

Remember that links starting with `https` or `git@` are read-write, and that `git@` uses the ssh protocol; links starting with `git://` are read-only.

Your collaborators can then commit directly into that repo with the usual:

```
git commit -am 'ENH - much better code'
git push origin master # pushes directly into your repo
```

Explore your repository

To see a graphical representation of the repository branches and commits:

```
gitk --all
```

To see a linear list of commits for this branch:

```
git log
```

You can also look at the network graph visualizer for your github repo.

Finally the *Fancy log output* `lg` alias will give you a reasonable text-based graph of the repository.

Rebasing on trunk

Let's say you thought of some work you'd like to do. You *Update the mirror of trunk* and *Make a new feature branch* called `cool-feature`. At this stage trunk is at some commit, let's call it E. Now you make some new commits on your `cool-feature` branch, let's call them A, B, C. Maybe your changes take a while, or you come back to them after a while. In the meantime, trunk has progressed from commit E to commit (say) G:

```
      A---B---C cool-feature
     /
D---E---F---G trunk
```

At this stage you consider merging trunk into your feature branch, and you remember that this here page sternly advises you not to do that, because the history will get messy. Most of the time you can just ask for a review, and not worry that trunk has got a little ahead. But sometimes, the changes in trunk might affect your changes, and you need to harmonize them. In this situation you may prefer to do a rebase.

rebase takes your changes (A, B, C) and replays them as if they had been made to the current state of `trunk`. In other words, in this case, it takes the changes represented by A, B, C and replays them on top of G. After the rebase, your history will look like this:

```
              A'--B'--C' cool-feature
             /
D---E---F---G trunk
```

See rebase without tears for more detail.

To do a rebase on trunk:

```
# Update the mirror of trunk
git fetch upstream
```

(continues on next page)

(continued from previous page)

```
# go to the feature branch
git checkout cool-feature
# make a backup in case you mess up
git branch tmp cool-feature
# rebase cool-feature onto trunk
git rebase --onto upstream/master upstream/master cool-feature
```

In this situation, where you are already on branch `cool-feature`, the last command can be written more succinctly as:

```
git rebase upstream/master
```

When all looks good you can delete your backup branch:

```
git branch -D tmp
```

If it doesn't look good you may need to have a look at *Recovering from mess-ups*.

If you have made changes to files that have also changed in trunk, this may generate merge conflicts that you need to resolve - see the git rebase man page for some instructions at the end of the "Description" section. There is some related help on merging in the git user manual - see resolving a merge.

Recovering from mess-ups

Sometimes, you mess up merges or rebases. Luckily, in git it is relatively straightforward to recover from such mistakes.

If you mess up during a rebase:

```
git rebase --abort
```

If you notice you messed up after the rebase:

```
# reset branch back to the saved point
git reset --hard tmp
```

If you forgot to make a backup branch:

```
# look at the reflog of the branch
git reflog show cool-feature

8630830 cool-feature@{0}: commit: BUG: io: close file handles immediately
278dd2a cool-feature@{1}: rebase finished: refs/heads/my-feature-branch onto␣
↪11ee694744f2552d
26aa21a cool-feature@{2}: commit: BUG: lib: make seek_gzip_factory not leak gzip obj
...

# reset the branch to where it was before the botched rebase
git reset --hard cool-feature@{2}
```

Rewriting commit history

Note: Do this only for your own feature branches.

There's an embarrassing typo in a commit you made? Or perhaps the you made several false starts you would like the posterity not to see.

This can be done via *interactive rebasing*.

Suppose that the commit history looks like this:

```
git log --oneline
eadc391 Fix some remaining bugs
a815645 Modify it so that it works
2dec1ac Fix a few bugs + disable
13d7934 First implementation
6ad92e5 * masked is now an instance of a new object, MaskedConstant
29001ed Add pre-nep for a copule of structured_array_extensions.
...
```

and `6ad92e5` is the last commit in the `cool-feature` branch. Suppose we want to make the following changes:

- Rewrite the commit message for `13d7934` to something more sensible.
- Combine the commits `2dec1ac`, `a815645`, `eadc391` into a single one.

We do as follows:

```
# make a backup of the current state
git branch tmp HEAD
# interactive rebase
git rebase -i 6ad92e5
```

This will open an editor with the following text in it:

```
pick 13d7934 First implementation
pick 2dec1ac Fix a few bugs + disable
pick a815645 Modify it so that it works
pick eadc391 Fix some remaining bugs

# Rebase 6ad92e5..eadc391 onto 6ad92e5
#
# Commands:
#  p, pick = use commit
#  r, reword = use commit, but edit the commit message
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#  f, fixup = like "squash", but discard this commit's log message
#
# If you remove a line here THAT COMMIT WILL BE LOST.
```

(continues on next page)

(continued from previous page)

```
# However, if you remove everything, the rebase will be aborted.
#
```

To achieve what we want, we will make the following changes to it:

```
r 13d7934 First implementation
pick 2dec1ac Fix a few bugs + disable
f a815645 Modify it so that it works
f eadc391 Fix some remaining bugs
```

This means that (i) we want to edit the commit message for `13d7934`, and (ii) collapse the last three commits into one. Now we save and quit the editor.

Git will then immediately bring up an editor for editing the commit message. After revising it, we get the output:

```
[detached HEAD 721fc64] FOO: First implementation
 2 files changed, 199 insertions(+), 66 deletions(-)
[detached HEAD 0f22701] Fix a few bugs + disable
 1 files changed, 79 insertions(+), 61 deletions(-)
Successfully rebased and updated refs/heads/my-feature-branch.
```

and the history looks now like this:

```
0f22701 Fix a few bugs + disable
721fc64 ENH: Sophisticated feature
6ad92e5 * masked is now an instance of a new object, MaskedConstant
```

If it went wrong, recovery is again possible as explained *above*.

35.5.5 Maintainer workflow

This page is for maintainers — those of us who merge our own or other peoples' changes into the upstream repository.

Being as how you're a maintainer, you are completely on top of the basic stuff in *Development workflow*.

The instructions in *Linking your repository to the upstream repo* add a remote that has read-only access to the upstream repo. Being a maintainer, you've got read-write access.

It's good to have your upstream remote have a scary name, to remind you that it's a read-write remote:

```
git remote add upstream-rw git@github.com:matplotlib/matplotlib.git
git fetch upstream-rw
```

Integrating changes

Let's say you have some changes that need to go into trunk (`upstream-rw/master`).

The changes are in some branch that you are currently on. For example, you are looking at someone's changes like this:

```
git remote add someone git://github.com/someone/matplotlib.git
git fetch someone
git branch cool-feature --track someone/cool-feature
git checkout cool-feature
```

So now you are on the branch with the changes to be incorporated upstream. The rest of this section assumes you are on this branch.

A few commits

If there are only a few commits, consider rebasing to upstream:

```
# Fetch upstream changes
git fetch upstream-rw
# rebase
git rebase upstream-rw/master
```

Remember that, if you do a rebase, and push that, you'll have to close any github pull requests manually, because github will not be able to detect the changes have already been merged.

A long series of commits

If there are a longer series of related commits, consider a merge instead:

```
git fetch upstream-rw
git merge --no-ff upstream-rw/master
```

The merge will be detected by github, and should close any related pull requests automatically.

Note the `--no-ff` above. This forces git to make a merge commit, rather than doing a fast-forward, so that these set of commits branch off trunk then rejoin the main history with a merge, rather than appearing to have been made directly on top of trunk.

Check the history

Now, in either case, you should check that the history is sensible and you have the right commits:

```
git log --oneline --graph
git log -p upstream-rw/master..
```

The first line above just shows the history in a compact way, with a text representation of the history graph. The second line shows the log of commits excluding those that can be reached from trunk (`upstream-rw/master`), and including those that can be reached from current HEAD (implied with the `..` at the end). So, it shows the commits unique to this branch compared to trunk. The `-p` option shows the diff for these commits in patch form.

Push to trunk

```
git push upstream-rw my-new-feature:master
```

This pushes the `my-new-feature` branch in this repository to the `master` branch in the `upstream-rw` repository.

35.6 git resources

35.6.1 Tutorials and summaries

- github help has an excellent series of how-to guides.
- The pro git book is a good in-depth book on git.
- A git cheat sheet is a page giving summaries of common commands.
- The git user manual
- The git tutorial
- The git community book
- git ready — a nice series of tutorials
- git magic — extended introduction with intermediate detail
- The git parable is an easy read explaining the concepts behind git.
- git foundation expands on the git parable.
- Fernando Perez' git page — Fernando's git page — many links and tips
- A good but technical page on git concepts

- git svn crash course: git for those of us used to subversion

35.6.2 Advanced git workflow

There are many ways of working with git; here are some posts on the rules of thumb that other projects have come up with:

- Linus Torvalds on git management
- Linus Torvalds on linux git workflow . Summary; use the git tools to make the history of your edits as clean as possible; merge from upstream edits as little as possible in branches where you are doing active development.

35.6.3 Manual pages online

You can get these on your own machine with (e.g) `git help push` or (same thing) `git push --help`, but, for convenience, here are the online manual pages for some common commands:

- git add
- git branch
- git checkout
- git clone
- git commit
- git config
- git diff
- git log
- git pull
- git push
- git remote
- git status

35.7 Two and three dots in difference specs

Thanks to Yarik Halchenko for this explanation.

Imagine a series of commits A, B, C, D... Imagine that there are two branches, *topic* and *master*. You branched *topic* off *master* when *master* was at commit 'E'. The graph of the commits looks like this:

```
     A---B---C topic
     /
D---E---F---G master
```

Then:

```
git diff master..topic
```

will output the difference from G to C (i.e. with effects of F and G), while:

```
git diff master...topic
```

would output just differences in the topic branch (i.e. only A, B, and C).

CHAPTER

THIRTYSIX

PULL REQUEST GUIDELINES

Pull requests (PRs) are the mechanism for contributing to Matplotlibs code and documentation.

36.1 Summary for PR authors

Note:

- We value contributions from people with all levels of experience. In particular if this is your first PR not everything has to be perfect. We'll guide you through the PR process.
- Nevertheless, try to follow the guidelines below as well as you can to help make the PR process quick and smooth.
- Be patient with reviewers. We try our best to respond quickly, but we have limited bandwidth. If there is no feedback within a couple of days, please ping us by posting a comment to your PR.

When making a PR, pay attention to:

- *Target the master branch*.
- Adhere to the *Coding guidelines*.
- Update the *documentation* if necessary.
- Aim at making the PR as "ready-to-go" as you can. This helps to speed up the review process.
- It is ok to open incomplete or work-in-progress PRs if you need help or feedback from the developers. You may mark these as draft pull requests on GitHub.
- When updating your PR, instead of adding new commits to fix something, please consider amending your initial commit(s) to keep the history clean. You can achieve this using:

```
git commit --amend --no-edit
git push [your-remote-repo] [your-branch] --force-with-lease
```

See also *Contributing* for how to make a PR.

36.2 Summary for PR reviewers

Note:

- If you have commit rights, then you are trusted to use them. **Please help review and merge PRs!**
- Be patient and kind with contributors.

Content topics:

- Is the feature / bugfix reasonable?
- Does the PR conform with the *Coding guidelines*?
- Is the *documentation* (docstrings, examples, what's new, API changes) updated?

Organizational topics:

- Make sure all *automated tests* pass.
- The PR should *target the master branch*.
- Tag with descriptive *labels*.
- Set the *milestone*.
- Keep an eye on the *number of commits*.
- Approve if all of the above topics handled.
- *Merge* if a sufficient number of approvals is reached.

36.3 Detailed Guidelines

36.3.1 Documentation

- Every new feature should be documented. If it's a new module, don't forget to add a new rst file to the API docs.
- Each high-level plotting function should have a small example in the `Examples` section of the docstring. This should be as simple as possible to demonstrate the method. More complex examples should go into a dedicated example file

in the `examples` directory, which will be rendered to the examples gallery in the documentation.

- Build the docs and make sure all formatting warnings are addressed.
- See *Writing documentation* for our documentation style guide.
- If your change is a major new feature, add an entry to `doc/users/whats_new.rst`.
- If you change the API in a backward-incompatible way, please document it in the relevant file in most recent `doc/api/api_changes_X.Y`.

36.3.2 Labels

- If you have the rights to set labels, tag the PR with descriptive labels. See the list of labels.

36.3.3 Milestones

- Set the milestone according to these rules:
  - *New features and API changes* are milestoned for the next minor release `v3.X.0`.
  - *Bugfixes and docstring changes* are milestoned for the next patch release `v3.X.Y`
  - *Documentation changes* (all .rst files and examples) are milestoned `v3.X-doc`

  If multiple rules apply, choose the first matching from the above list.

  Setting a milestone does not imply or guarantee that a PR will be merged for that release, but if it were to be merged what release it would be in.

  All of these PRs should target the master branch. The milestone tag triggers an *automatic backport* for milestones which have a corresponding branch.

36.3.4 Merging

- Documentation and examples may be merged by the first reviewer. Use the threshold "is this better than it was?" as the review criteria.
- For code changes (anything in `src` or `lib`) at least two core developers (those with commit rights) should review all pull requests. If you are the first to review a PR and approve of the changes use the GitHub 'approve review' tool to mark it as such. If you are a subsequent reviewer please approve the review and if you think no more review is needed, merge the PR.

  Ensure that all API changes are documented in the relevant file in the most recent `doc/api/api_changes_X.Y` and significant new features have an entry in `doc/user/whats_new`.

  - If a PR already has a positive review, a core developer (e.g. the first reviewer, but not necessarily) may champion that PR for merging. In order to do so, they should ping all core devs both on GitHub and on the dev mailing list, and label the PR with the "Merge with single review?" label. Other core devs can then either review the PR and merge or reject it, or simply request that it gets a second review before being merged. If no one asks for such a second review within a week, the PR can then be merged on the basis of that single review.

    A core dev should only champion one PR at a time and we should try to keep the flow of championed PRs reasonable.

- Do not self merge, except for 'small' patches to un-break the CI or when another reviewer explicitly allows it (ex, "Approve modulo CI passing, may self merge when green").

36.3.5 Automated tests

Whenever a pull request is created or updated, various automated test tools will run on all supported platforms and versions of Python.

- Make sure the Linting, Travis, AppVeyor, CircleCI, and Azure pipelines are passing before merging (All checks are listed at the bottom of the GitHub page of your pull request). Here are some tips for finding the cause of the test failure:
  - If *Linting* fails, you have a code style issue, which will be listed as annotations on the pull request's diff.
  - If a Travis or AppVeyor run fails, search the log for `FAILURES`. The subsequent section will contain information on the failed tests.
  - If CircleCI fails, likely you have some reStructuredText style issue in the docs. Search the CircleCI log for `WARNING`.
  - If Azure pipelines fail with an image comparison error, you can find the images as *artifacts* of the Azure job:
    * Click *Details* on the check on the GitHub PR page.
    * Click *View more details on Azure Pipelines* to go to Azure.
    * On the overview page *artifacts* are listed in the section *Related*.
- Codecov and LGTM are currently for information only. Their failure is not necessarily a blocker.
- tox is not used in the automated testing. It is supported for testing locally.

36.3.6 Number of commits and squashing

- Squashing is case-by-case. The balance is between burden on the contributor, keeping a relatively clean history, and keeping a history usable for bisecting. The only time we are really strict about it is to eliminate binary files (ex multiple test image re-generations) and to remove upstream merges.
- Do not let perfect be the enemy of the good, particularly for documentation or example PRs. If you find yourself making many small suggestions, either open a PR against the original branch, push changes to the contributor branch, or merge the PR and then open a new PR against upstream.
- If you push to a contributor branch leave a comment explaining what you did, ex "I took the liberty of pushing a small clean-up PR to your branch, thanks for your work.". If you are going to make substantial changes to the code or intent of the PR please check with the contributor first.

36.4 Branches and Backports

36.4.1 Current branches

The current active branches are

master

The current development version. Future minor releases (*v3.N.0*) will be branched from this. Supports Python 3.6+.

v3.N.x

Maintenance branch for Matplotlib 3.N. Future patch releases will be branched from this. Supports Python 3.6+.

v3.N.M-doc

Documentation for the current release. On a patch release, this will be replaced by a properly named branch for the new release.

v2.2.x

Maintenance branch for Matplotlib 2.2 LTS. Supports Python 2.7, 3.4+.

v2.2.N-doc

Documentation for the current release. On a patch release, this will be replaced by a properly named branch for the new release.

36.4.2 Branch selection for pull requests

Generally, all pull requests should target the master branch.

Other branches are fed through *automatic* or *manual*. Directly targeting other branches is only rarely necessary for special maintenance work.

36.4.3 Backport strategy

We will always backport to the patch release branch (*v3.N.x*):

- critical bug fixes (segfault, failure to import, things that the user can not work around)
- fixes for regressions against the last two releases.

Everything else (regressions against older releases, bugs/api inconsistencies the user can work around in their code) are on a case-by-case basis, should be low-risk, and need someone to advocate for and shepherd through the backport.

The only changes to be backported to the documentation branch (*v3.N.M-doc*) are changes to `doc`, `examples`, or `tutorials`. Any changes to `lib` or `src` including docstring-only changes should not be backported to this branch.

36.4.4 Automated backports

We use meeseeksdev bot to automatically backport merges to the correct maintenance branch base on the milestone. To work properly the milestone must be set before merging. If you have commit rights, the bot can also be manually triggered after a merge by leaving a message `@meeseeksdev backport to BRANCH` on the PR. If there are conflicts meeseekdevs will inform you that the backport needs to be done manually.

The target branch is configured by putting `on-merge: backport to TARGETBRANCH` in the milestone description on it's own line.

If the bot is not working as expected, please report issues to Meeseeksdev.

36.4.5 Manual backports

When doing backports please copy the form used by meeseekdev, `Backport PR #XXXX: TITLE OF PR`. If you need to manually resolve conflicts make note of them and how you resolved them in the commit message.

We do a backport from master to v2.2.x assuming:

- `matplotlib` is a read-only remote branch of the matplotlib/matplotlib repo

The TARGET_SHA is the hash of the merge commit you would like to backport. This can be read off of the GitHub PR page (in the UI with the merge notification) or through the git CLI tools.

Assuming that you already have a local branch v2.2.x (if not, then git checkout -b v2.2.x), and that your remote pointing to https://github.com/matplotlib/matplotlib is called upstream:

```
git fetch upstream
git checkout v2.2.x  # or include -b if you don't already have this.
git reset --hard upstream/v2.2.x
git cherry-pick -m 1 TARGET_SHA
# resolve conflicts and commit if required
```

Files with conflicts can be listed by git status, and will have to be fixed by hand (search on >>>>>). Once the conflict is resolved, you will have to re-add the file(s) to the branch and then continue the cherry pick:

```
git add lib/matplotlib/conflicted_file.py
git add lib/matplotlib/conflicted_file2.py
git cherry-pick --continue
```

Use your discretion to push directly to upstream or to open a PR; be sure to push or PR against the v2.2.x upstream branch, not master!

CHAPTER

THIRTYSEVEN

RELEASE GUIDE

This document is only relevant for Matplotlib release managers.

A guide for developers who are doing a Matplotlib release.

Note: This assumes that a read-only remote for the canonical repository is `remote` and a read/write remote is `DANGER`

37.1 All Releases

37.1.1 Testing

We use Travis CI for continuous integration. When preparing for a release, the final tagged commit should be tested locally before it is uploaded:

```
pytest -n 8 .
```

In addition the following test should be run and manually inspected:

```
python tools/memleak.py agg 1000 agg.pdf
```

In addition the following should be run and manually inspected, but is currently broken:

```
pushd examples/tests/
python backend_driver_sgskip.py
popd
```

37.1.2 GitHub Stats

We automatically extract GitHub issue, PRs, and authors from GitHub via the API. Copy the current `doc/users/github_stats.rst` to `doc/users/prev_whats_new/github_stats_X.Y.Z.rst`, changing the link target at the top of the file, and removing the "Previous GitHub Stats" section at the end.

For example, when updating from v3.2.0 to v3.2.1:

```
cp doc/users/github_stats.rst doc/users/prev_whats_new/github_stats_3.2.0.rst
$EDITOR doc/users/prev_whats_new/github_stats_3.2.0.rst
# Change contents as noted above.
git add doc/users/prev_whats_new/github_stats_3.2.0.rst
```

Then re-generate the updated stats:

```
python tools/github_stats.py --since-tag v3.2.0 --milestone=v3.2.1 --project 'matplotlib/
↪matplotlib' --links > doc/users/github_stats.rst
```

Review and commit changes. Some issue/PR titles may not be valid reST (the most common issue is * which is interpreted as unclosed markup).

Note: Make sure you authenticate against the GitHub API. If you do not you will get blocked by GitHub for going over the API rate limits. You can authenticate in one of two ways:

- using the `keyring` package; `pip install keyring` and then when running the stats script, you will be prompted for user name and password, that will be stored in your system keyring, or,
- using a personal access token; generate a new token on this GitHub page with the `repo:public_repo` scope and place the token in `~/.ghoauth`.

37.1.3 Update and Validate the Docs

Merge `*-doc` branch

Merge the most recent 'doc' branch (e.g., `v3.2.0-doc`) into the branch you are going to tag on and delete the doc branch on GitHub.

Update "What's New" and "API changes"

Before tagging major and minor releases, the "what's new" and "API changes" listings should be updated. This is not needed for micro releases.

For the "what's new",

1. copy the current content to a file in `doc/users/prev_whats_new`
2. merge all of the files in `doc/users/next_whats_new/` into `doc/users/whats_new.rst` and delete the individual files
3. comment out the next what's new glob at the top

Similarly for the "API changes",

1. copy the current api changes to a file is `doc/api/prev_api_changes`
2. merge all of the files in the most recent `doc/api/api_changes_X.Y` into `doc/api/api_changes.rst`
3. comment out the most recent API changes at the top.

In both cases step 3 will have to be un-done right after the release.

Verify that docs build

Finally, make sure that the docs build cleanly

```
make -Cdoc O=-j$(nproc) html latexpdf
```

After the docs are built, check that all of the links, internal and external, are still valid. We use `linkchecker` for this, which has not been ported to Python3 yet. You will need to create a Python2 environment with `requests==2.9.0` and linkchecker

```
conda create -p /tmp/lnkchk python=2 requests==2.9.0
source activate /tmp/lnkchk
pip install linkchecker
pushd doc/build/html
linkchecker index.html --check-extern
popd
```

Address any issues which may arise. The internal links are checked on Circle CI, this should only flag failed external links.

37.1.4 Create release commit and tag

To create the tag, first create an empty commit with a very terse set of the release notes in the commit message

```
git commit --allow-empty
```

and then create a signed, annotated tag with the same text in the body message

```
git tag -a -s v2.0.0
```

which will prompt you for your GPG key password and an annotation. For pre releases it is important to follow **PEP 440** so that the build artifacts will sort correctly in PyPI.

To prevent issues with any down-stream builders which download the tarball from GitHub it is important to move all branches away from the commit with the tag[1]:

```
git commit --allow-empty
```

Finally, push the tag to GitHub:

```
git push DANGER master v2.0.0
```

Congratulations, the scariest part is done!

If this is a final release, also create a 'doc' branch (this is not done for pre-releases):

```
git branch v2.0.0-doc
git push DANGER v2.0.0-doc
```

and if this is a major or minor release, also create a bug-fix branch (a micro release will be cut from this branch):

```
git branch v2.0.x
```

On this branch un-comment the globs from *Update and Validate the Docs*. And then

```
git push DANGER v2.0.x
```

[1] The tarball that is provided by GitHub is produced using git archive. We use versioneer which uses a format string in `lib/matplotlib/_version.py` to have `git` insert a list of references to exported commit (see `.gitattributes` for the configuration). This string is then used by `versioneer` to produce the correct version, based on the git tag, when users install from the tarball. However, if there is a branch pointed at the tagged commit, then the branch name will also be included in the tarball. When the branch eventually moves, anyone how checked the hash of the tarball before the branch moved will have an incorrect hash.

To generate the file that GitHub does use

```
git archive v2.0.0 -o matplotlib-2.0.0.tar.gz --prefix=matplotlib-2.0.0/
```

37.1.5 Release Management / DOI

Via the GitHub UI, turn the newly pushed tag into a release. If this is a pre-release remember to mark it as such.

For final releases, also get the DOI from zenodo (which will automatically produce one once the tag is pushed). Add the doi post-fix and version to the dictionary in `tools/cache_zenodo_svg.py` and run the script.

This will download the new svg to the `_static` directory in the docs and edit `doc/citing.rst`. Commit the new svg, the change to `tools/cache_zenodo_svg.py`, and the changes to `doc/citing.rst` to the VER-doc branch and push to GitHub.

```
git checkout v2.0.0-doc
$EDITOR tools/cache_zenodo_svg.py
python tools/cache_zenodo_svg.py
$EDITOR doc/citing.html
git commit -a
git push DANGER v2.0.0-doc:v2.0.0-doc
```

37.1.6 Building binaries

We distribute macOS, Windows, and many Linux wheels as well as a source tarball via PyPI. Most builders should trigger automatically once the tag is pushed to GitHub:

- Mac and manylinux wheels are built on Travis CI. Builds are triggered by the GitHub Action defined in `.github/workflows/wheels.yml`, and wheels will be available at the site defined in the matplotlib-wheels repo.
- Windows wheels are built by Christoph Gohlke automatically and will be available at his site once built.
- The auto-tick bot should open a pull request into the conda-forge feedstock. Review and merge (if you have the power to).

Warning: Because this is automated, it is extremely important to bump all branches away from the tag as discussed in *Create release commit and tag*.

If this is a final release the following downstream packagers should be contacted:

- Debian
- Fedora
- Arch
- Gentoo
- Macports
- Homebrew

- Continuum
- Enthought

This can be done ahead of collecting all of the binaries and uploading to pypi.

37.1.7 Make distribution and upload to PyPI

Once you have collected all of the wheels (expect this to take about a day), generate the tarball

```
git checkout v2.0.0
git clean -xfd
python setup.py sdist
```

and copy all of the wheels into `dist` directory. First, check that the dist files are OK

```
twine check dist/*
```

and then use `twine` to upload all of the files to pypi

```
twine upload -s dist/matplotlib*tar.gz
twine upload dist/*whl
```

Congratulations, you have now done the second scariest part!

37.1.8 Build and Deploy Documentation

To build the documentation you must have the tagged version installed, but build the docs from the `ver-doc` branch. An easy way to arrange this is:

```
pip install matplotlib
pip install -r requirements/doc/doc-requirements.txt
git checkout v2.0.0-doc
git clean -xfd
make -Cdoc O=-j$(nproc) html latexpdf LATEXMKOPTS="-silent -f"
```

which will build both the html and pdf version of the documentation.

The built documentation exists in the matplotlib.github.com repository. Pushing changes to master automatically updates the website.

The documentation is organized by version. At the root of the tree is always the documentation for the latest stable release. Under that, there are directories containing the documentation for older versions. The documentation for current master is built on Circle CI and pushed to the devdocs repository. These are available at matplotlib.org/devdocs.

Assuming you have this repository checked out in the same directory as matplotlib

```
cd ../matplotlib.github.com
mkdir 2.0.0
rsync -a ../matplotlib/doc/build/html/* 2.0.0
cp ../matplotlib/doc/build/latex/Matplotlib.pdf 2.0.0
```

which will copy the built docs over. If this is a final release, also replace the top-level docs

```
rsync -a 2.0.0/* ./
```

You will need to manually edit `versions.html` to show the last 3 tagged versions. Now commit and push everything to GitHub

```
git add *
git commit -a -m 'Updating docs for v2.0.0'
git push DANGER master
```

Congratulations you have now done the third scariest part!

If you have access, clear the Cloudflare caches.

It typically takes about 5-10 minutes for GitHub to process the push and update the live web page (remember to clear your browser cache).

37.1.9 Announcing

The final step is to announce the release to the world. A short version of the release notes along with acknowledgments should be sent to

- matplotlib-users@python.org
- matplotlib-devel@python.org
- matplotlib-announce@python.org

For final releases announcements should also be sent to the numpy/scipy/scikit-image mailing lists.

In addition, announcements should be made on social networks (twitter via the `@matplotlib` account, any other via personal accounts). NumFOCUS should be contacted for inclusion in their newsletter.

CHAPTER

THIRTYEIGHT

MINIMUM VERSION OF DEPENDENCIES POLICY

For the purpose of this document, 'minor version' is in the sense of SemVer (major, minor, patch) and includes both major and minor releases. For projects that use date-based versioning, every release is a 'minor version'.

Matplotlib follows NEP 29.

38.1 Python and NumPy

Matplotlib supports:

- All minor versions of Python released 42 months prior to the project, and at minimum the two latest minor versions.
- All minor versions of `numpy` released in the 24 months prior to the project, and at minimum the last three minor versions.

In `setup.py`, the `python_requires` variable should be set to the minimum supported version of Python. All supported minor versions of Python should be in the test matrix and have binary artifacts built for the release.

Minimum Python and NumPy version support should be adjusted upward on every major and minor release, but never on a patch release.

See also the *List of dependency versions*.

38.2 Python Dependencies

For Python dependencies we should support at least:

with compiled extensions

minor versions initially released in the 24 months prior to our planned release date or the oldest that support our minimum Python + NumPy

without complied extensions

minor versions initially released in the 12 months prior to our planned release date or the oldest that supports our minimum Python.

We will only bump these dependencies as we need new features or the old versions no longer support our minimum NumPy or Python.

38.3 Test and Documentation Dependencies

As these packages are only needed for testing or building the docs and not needed by end-users, we can be more aggressive about dropping support for old versions. However, we need to be careful to not over-run what down-stream packagers support (as most of the run the tests and build the documentation as part of the packaging process).

We will support at least minor versions of the development dependencies released in the 12 months prior to our planned release.

We will only bump these as needed or versions no longer support our minimum Python and numpy.

38.4 System and C-dependencies

For system or C-dependencies (FreeType, GUI frameworks, LaTeX, Ghostscript, FFmpeg) support as old as practical. These can be difficult to install for end-users and we want to be usable on as many systems as possible. We will bump these on a case-by-case basis.

38.5 List of dependency versions

The following list shows the minimal versions of Python and NumPy dependencies for different versions of Matplotlib. Follow the links for the full specification of the dependencies.

Matplotlib	Python	NumPy
3.3	3.6	1.15.0
3.2	3.6	1.11.0
3.1	3.6	1.11.0
3.0	3.5	1.10.0
2.2	2.7, 3.4	1.7.1
2.1	2.7, 3.4	1.7.1
2.0	2.7, 3.4	1.7.1
1.5	2.7, 3.4	1.6
1.4	2.6, 3.3	1.6
1.3	2.6, 3.3	1.5
1.2	2.6, 3.1	1.4
1.1	2.4	1.1
1.0	2.4	1.1

CHAPTER

THIRTYNINE

MATPLOTLIB ENHANCEMENT PROPOSALS

Matplotlib Enhancement Proposals (MEP), inspired by cpython's PEP's but less formal, are design documents for large or controversial changes to Matplotilb. These documents should provide a discussion of both why and how the changes should be made.

To create a new MEP open a pull request (PR) adding a file based on *the template* to this the MEP directory. For the initial PR only a rough description is required and it should be merged quickly. Further detailed discussion can happen in follow on PRs.

39.1 MEP Template

This MEP template is a guideline of the sections that a MEP should contain. Extra sections may be added if appropriate, and unnecessary sections may be noted as such.

39.1.1 Status

MEPs go through a number of phases in their lifetime:

- **Discussion**: The MEP is being actively discussed on the mailing list and it is being improved by its author. The mailing list discussion of the MEP should include the MEP number (MEPxxx) in the subject line so they can be easily related to the MEP.
- **Progress**: Consensus was reached and implementation work has begun.
- **Completed**: The implementation has been merged into master.
- **Superseded**: This MEP has been abandoned in favor of another approach.
- **Rejected**: There are currently no plans to implement the proposal.

39.1.2 Branches and Pull requests

All development branches containing work on this MEP should be linked to from here.

All pull requests submitted relating to this MEP should be linked to from here. (A MEP does not need to be implemented in a single pull request if it makes sense to implement it in discrete phases).

39.1.3 Abstract

The abstract should be a short description of what the MEP will achieve.

39.1.4 Detailed description

This section describes the need for the MEP. It should describe the existing problem that it is trying to solve and why this MEP makes the situation better. It should include examples of how the new functionality would be used and perhaps some use cases.

39.1.5 Implementation

This section lists the major steps required to implement the MEP. Where possible, it should be noted where one step is dependent on another, and which steps may be optionally omitted. Where it makes sense, each step should include a link related pull requests as the implementation progresses.

39.1.6 Backward compatibility

This section describes the ways in which the MEP breaks backward incompatibility.

39.1.7 Alternatives

If there were any alternative solutions to solving the same problem, they should be discussed here, along with a justification for the chosen approach.

39.2 MEP8: PEP8

39.2.1 Status

Completed

We are currently enforcing a sub-set of pep8 on new code contributions.

39.2.2 Branches and Pull requests

None so far.

39.2.3 Abstract

The matplotlib codebase predates PEP8, and therefore is less than consistent style-wise in some areas. Bringing the codebase into compliance with PEP8 would go a long way to improving its legibility.

39.2.4 Detailed description

Some files use four space indentation, some use three. Some use different levels in the same file.

For the most part, class/function/variable naming follows PEP8, but it wouldn't hurt to fix where necessary.

39.2.5 Implementation

The implementation should be fairly mechanical: running the pep8 tool over the code and fixing where appropriate.

This should be merged in after the 2.0 release, since the changes will likely make merging any pending pull requests more difficult.

Additionally, and optionally, PEP8 compliance could be tracked by an automated build system.

39.2.6 Backward compatibility

Public names of classes and functions that require change (there shouldn't be many of these) should first be deprecated and then removed in the next release cycle.

39.2.7 Alternatives

PEP8 is a popular standard for Python code style, blessed by the Python core developers, making any alternatives less desirable.

39.3 MEP9: Global interaction manager

Add a global manager for all user interactivity with artists; make any artist resizeable, moveable, highlightable, and selectable as desired by the user.

39.3.1 Status

Discussion

39.3.2 Branches and Pull requests

https://github.com/dhyams/matplotlib/tree/MEP9

39.3.3 Abstract

The goal is to be able to interact with matplotlib artists in a very similar way as drawing programs do. When appropriate, the user should be able to move, resize, or select an artist that is already on the canvas. Of course, the script writer is ultimately in control of whether an artist is able to be interacted with, or whether it is static.

This code to do this has already been privately implemented and tested, and would need to be migrated from its current "mixin" implementation, to a bona-fide part of matplotlib.

The end result would be to have four new keywords available to matplotlib.artist.Artist: _moveable_, _resizeable_, _selectable_, and _highlightable_. Setting any one of these keywords to True would activate interactivity for that artist.

In effect, this MEP is a logical extension of event handling in matplotlib; matplotlib already supports "low level" interactions like left mouse presses, a key press, or similar. The MEP extends the support to the logical level, where callbacks are performed on the artists when certain interactive gestures from the user are detected.

39.3.4 Detailed description

This new functionality would be used to allow the end-user to better interact with the graph. Many times, a graph is almost what the user wants, but a small repositioning and/or resizing of components is necessary. Rather than force the user to go back to the script to trial-and-error the location, and simple drag and drop would be appropriate.

Also, this would better support applications that use matplotlib; here, the end-user has no reasonable access or desire to edit the underlying source in order to fine-tune a plot. Here, if matplotlib offered the capability, one could move or resize artists on the canvas to suit their needs. Also, the user should be able to highlight (with a mouse over) an artist, and select it with a double-click, if the application supports that sort of thing. In this MEP, we also want to support the highlighting and selection natively; it is up to application to handle what happens when the artist is selected. A typical handling would be to display a dialog to edit the properties of the artist.

In the future, as well (this is not part of this MEP), matplotlib could offer backend-specific property dialogs for each artist, which are raised on artist selection. This MEP would be a necessary stepping stone for that sort of capability.

There are currently a few interactive capabilities in matplotlib (e.g. legend.draggable()), but they tend to be scattered and are not available for all artists. This MEP seeks to unify the interactive interface and make it work for all artists.

The current MEP also includes grab handles for resizing artists, and appropriate boxes drawn when artists are moved or resized.

39.3.5 Implementation

- Add appropriate methods to the "tree" of artists so that the interactivity manager has a consistent interface for the interactivity manager to deal with. The proposed methods to add to the artists, if they are to support interactivity, are:
  - get_pixel_position_ll(self): get the pixel position of the lower left corner of the artist's bounding box
  - get_pixel_size(self): get the size of the artist's bounding box, in pixels
  - set_pixel_position_and_size(self,x,y,dx,dy): set the new size of the artist, such that it fits within the specified bounding box.
- add capability to the backends to 1) provide cursors, since these are needed for visual indication of moving/resizing, and 2) provide a function that gets the current mouse position
- Implement the manager. This has already been done privately (by dhyams) as a mixin, and has been tested quite a bit. The goal would be to move the functionality of the manager into the artists so that it is in matplotlib properly, and not as a "monkey patch" as I currently have it coded.

39.3.6 Current summary of the mixin

(Note that this mixin is for now just private code, but can be added to a branch obviously)

InteractiveArtistMixin:

Mixin class to make any generic object that is drawn on a matplotlib canvas moveable and possibly resizeable. The Powerpoint model is followed as closely as possible; not because I'm enamoured with Powerpoint, but because that's what most people understand. An artist can also be selectable, which means that the artist will receive the on_activated() callback when double clicked. Finally, an artist can be highlightable, which means that a highlight is drawn on the artist whenever the mouse passes over. Typically, highlightable artists will also be selectable, but that is left up to the user. So, basically there are four attributes that can be set by the user on a per-artist basis:

- highlightable
- selectable
- moveable
- resizeable

To be moveable (draggable) or resizeable, the object that is the target of the mixin must support the following protocols:

- get_pixel_position_ll(self)
- get_pixel_size(self)
- set_pixel_position_and_size(self,x,y,sx,sy)

Note that nonresizeable objects are free to ignore the sx and sy parameters. To be highlightable, the object that is the target of the mixin must also support the following protocol:

- get_highlight(self)

Which returns a list of artists that will be used to draw the highlight.

If the object that is the target of the mixin is not an matplotlib artist, the following protocols must also be implemented. Doing so is usually fairly trivial, as there has to be an artist *somewhere* that is being drawn. Typically your object would just route these calls to that artist.

- get_figure(self)
- get_axes(self)
- contains(self,event)
- set_animated(self,flag)
- draw(self,renderer)
- get_visible(self)

The following notifications are called on the artist, and the artist can optionally implement these.

- on_select_begin(self)
- on_select_end(self)
- on_drag_begin(self)
- on_drag_end(self)
- on_activated(self)
- on_highlight(self)
- on_right_click(self,event)
- on_left_click(self,event)
- on_middle_click(self,event)
- on_context_click(self,event)
- on_key_up(self,event)
- on_key_down(self,event)

The following notifications are called on the canvas, if no interactive artist handles the event:

- on_press(self,event)
- on_left_click(self,event)
- on_middle_click(self,event)
- on_right_click(self,event)
- on_context_click(self,event)
- on_key_up(self,event)
- on_key_down(self,event)

The following functions, if present, can be used to modify the behavior of the interactive object:

- press_filter(self,event) # determines if the object wants to have the press event routed to it
- handle_unpicked_cursor() # can be used by the object to set a cursor as the cursor passes over the object when it is unpicked.

Supports multiple canvases, maintaining a drag lock, motion notifier, and a global "enabled" flag per canvas. Supports fixed aspect ratio resizings by holding the shift key during the resize.

Known problems:

- Zorder is not obeyed during the selection/drag operations. Because of the blit technique used, I do not believe this can be fixed. The only way I can think of is to search for all artists that have a zorder greater then me, set them all to animated, and then redraw them all on top during each drag refresh. This might be very slow; need to try.
- the mixin only works for wx backends because of two things: 1) the cursors are hardcoded, and 2) there is a call to wx.GetMousePosition() Both of these shortcomings are reasonably fixed by having each backend supply these things.

39.3.7 Backward compatibility

No problems with backward compatibility, although once this is in place, it would be appropriate to obsolete some of the existing interactive functions (like legend.draggable())

39.3.8 Alternatives

None that I know of.

39.4 MEP10: Docstring consistency

39.4.1 Status

Progress

This is still an on-going effort

39.4.2 Branches and Pull requests

39.4.3 Abstract

matplotlib has a great deal of inconsistency between docstrings. This not only makes the docs harder to read, but it is harder on contributors, because they don't know which specifications to follow. There should be a clear docstring convention that is followed consistently.

The organization of the API documentation is difficult to follow. Some pages, such as pyplot and axes, are enormous and hard to browse. There should instead be short summary tables that link to detailed documentation. In addition, some of the docstrings themselves are quite long and contain redundant information.

Building the documentation takes a long time and uses a `make.py` script rather than a Makefile.

39.4.4 Detailed description

There are number of new tools and conventions available since matplotlib started using Sphinx that make life easier. The following is a list of proposed changes to docstrings, most of which involve these new features.

Numpy docstring format

Numpy docstring format: This format divides the docstring into clear sections, each having different parsing rules that make the docstring easy to read both as raw text and as HTML. We could consider alternatives, or invent our own, but this is a strong choice, as it's well used and understood in the Numpy/Scipy community.

Cross references

Most of the docstrings in matplotlib use explicit "roles" when linking to other items, for example: `:func:`myfunction``. As of Sphinx 0.4, there is a "default_role" that can be set to "obj", which will polymorphically link to a Python object of any type. This allows one to write `` `myfunction` `` instead. This makes docstrings much easier to read and edit as raw text. Additionally, Sphinx allows for setting a current module, so links like `` `~matplotlib.axes.Axes.set_xlim` `` could be written as `` `~axes.Axes.set_xlim` ``.

Overriding signatures

Many methods in matplotlib use the `*args` and `**kwargs` syntax to dynamically handle the keyword arguments that are accepted by the function, or to delegate on to another function. This, however, is often not useful as a signature in the documentation. For this reason, many matplotlib methods include something like:

```
def annotate(self, *args, **kwargs):
    """
    Create an annotation: a piece of text referring to a data
    point.

    Call signature::

      annotate(s, xy, xytext=None, xycoords='data',
               textcoords='data', arrowprops=None, **kwargs)
    """
```

This can't be parsed by Sphinx, and is rather verbose in raw text. As of Sphinx 1.1, if the `autodoc_docstring_signature` config value is set to True, Sphinx will extract a replacement signature from the first line of the docstring, allowing this:

```
def annotate(self, *args, **kwargs):
    """
    annotate(s, xy, xytext=None, xycoords='data',
               textcoords='data', arrowprops=None, **kwargs)

    Create an annotation: a piece of text referring to a data
    point.
    """
```

The explicit signature will replace the actual Python one in the generated documentation.

Linking rather than duplicating

Many of the docstrings include long lists of accepted keywords by interpolating things into the docstring at load time. This makes the docstrings very long. Also, since these tables are the same across many docstrings, it inserts a lot of redundant information in the docs -- particularly a problem in the printed version.

These tables should be moved to docstrings on functions whose only purpose is for help. The docstrings that refer to these tables should link to them, rather than including them verbatim.

autosummary extension

The Sphinx autosummary extension should be used to generate summary tables, that link to separate pages of documentation. Some classes that have many methods (e.g. *Axes*) should be documented with one method per page, whereas smaller classes should have all of their methods together.

Examples linking to relevant documentation

The examples, while helpful at illustrating how to use a feature, do not link back to the relevant docstrings. This could be addressed by adding module-level docstrings to the examples, and then including that docstring in the parsed content on the example page. These docstrings could easily include references to any other part of the documentation.

Documentation using help() vs. a browser

Using Sphinx markup in the source allows for good-looking docs in your browser, but the markup also makes the raw text returned using help() look terrible. One of the aims of improving the docstrings should be to make both methods of accessing the docs look good.

39.4.5 Implementation

1. The numpydoc extensions should be turned on for matplotlib. There is an important question as to whether these should be included in the matplotlib source tree, or used as a dependency. Installing Numpy is not sufficient to get the numpydoc extensions -- it's a separate install procedure. In any case, to the extent that they require customization for our needs, we should endeavor to submit those changes upstream and not fork them.
2. Manually go through all of the docstrings and update them to the new format and conventions. Updating the cross references (from `` `:func:`myfunc` `` to `` `func` ``) may be able to be semi-automated. This is a lot of busy work, and perhaps this labor should be divided on a per-module basis so no single developer is over-burdened by it.
3. Reorganize the API docs using autosummary and `sphinx-autogen`. This should hopefully have minimal impact on the narrative documentation.
4. Modify the example page generator (`gen_rst.py`) so that it extracts the module docstring from the example and includes it in a non-literal part of the example page.
5. Use `sphinx-quickstart` to generate a new-style Sphinx Makefile. The following features in the current `make.py` will have to be addressed in some other way:

- Copying of some static content
- Specifying a "small" build (only low-resolution PNG files for examples)

Steps 1, 2, and 3 are interdependent. 4 and 5 may be done independently, though 5 has some dependency on 3.

39.4.6 Backward compatibility

As this mainly involves docstrings, there should be minimal impact on backward compatibility.

39.4.7 Alternatives

None yet discussed.

39.5 MEP11: Third-party dependencies

This MEP attempts to improve the way in which third-party dependencies in matplotlib are handled.

39.5.1 Status

Completed -- needs to be merged

39.5.2 Branches and Pull requests

#1157: Use automatic dependency resolution

#1290: Debundle pyparsing

#1261: Update six to 1.2

39.5.3 Abstract

One of the goals of matplotlib has been to keep it as easy to install as possible. To that end, some third-party dependencies are included in the source tree and, under certain circumstances, installed alongside matplotlib. This MEP aims to resolve some problems with that approach, bring some consistency, while continuing to make installation convenient.

At the time that was initially done, setuptools, easy_install and PyPI were not mature enough to be relied on. However, at present, we should be able to safely leverage the "modern" versions of those tools, distribute and pip.

While matplotlib has dependencies on both Python libraries and C/C++ libraries, this MEP addresses only the Python libraries so as to not confuse the issue. C libraries represent a larger and mostly orthogonal set of problems.

39.5.4 Detailed description

matplotlib depends on the following third-party Python libraries:

- Numpy
- dateutil (pure Python)
- pytz (pure Python)
- six -- required by dateutil (pure Python)
- pyparsing (pure Python)
- PIL (optional)
- GUI frameworks: pygtk, gobject, tkinter, PySide, PyQt4, wx (all optional, but one is required for an interactive GUI)

Current behavior

When installing from source, a `git` checkout or pip:

- `setup.py` attempts to `import numpy`. If this fails, the installation fails.
- For each of dateutil, pytz and six, `setup.py` attempts to import them (from the top-level namespace). If that fails, matplotlib installs its local copy of the library into the top-level namespace.
- pyparsing is always installed inside of the matplotlib namespace.

This behavior is most surprising when used with pip, because no pip dependency resolution is performed, even though it is likely to work for all of these packages.

The fact that pyparsing is installed in the matplotlib namespace has reportedly (#1290) confused some users into thinking it is a matplotlib-related module and import it from there rather than the top-level.

When installing using the Windows installer, dateutil, pytz and six are installed at the top-level *always*, potentially overwriting already installed copies of those libraries.

TODO: Describe behavior with the OS-X installer.

When installing using a package manager (Debian, RedHat, MacPorts etc.), this behavior actually does the right thing, and there are no special patches in the matplotlib packages to deal with the fact that we handle dateutil, pytz and six in this way. However, care should be taken that whatever approach we move to continues to work in that context.

Maintaining these packages in the matplotlib tree and making sure they are up-to-date is a maintenance burden. Advanced new features that may require a third-party pure Python library have a higher barrier to inclusion because of this burden.

Desired behavior

Third-party dependencies are downloaded and installed from their canonical locations by leveraging pip, distribute and PyPI.

dateutil, pytz, and pyparsing should be made into optional dependencies -- though obviously some features would fail if they aren't installed. This will allow the user to decide whether they want to bother installing a particular feature.

39.5.5 Implementation

For installing from source, and assuming the user has all of the C-level compilers and dependencies, this can be accomplished fairly easily using distribute and following the instructions here. The only anticipated change to the matplotlib library code will be to import pyparsing from the top-level namespace rather than from within matplotlib. Note that distribute will also allow us to remove the direct dependency on six, since it is, strictly speaking, only a direct dependency of dateutil.

For binary installations, there are a number of alternatives (here ordered from best/hardest to worst/easiest):

1. The distutils wininst installer allows a post-install script to run. It might be possible to get this script to run pip to install the other dependencies. (See this thread for someone who has trod that ground before).
2. Continue to ship dateutil, pytz, six and pyparsing in our installer, but use the post-install-script to install them *only* if they can not already be found.
3. Move all of these packages inside a (new) `matplotlib.extern` namespace so it is clear for outside users that these are external packages. Add some conditional imports in the core matplotlib codebase so dateutil (at the top-level) is tried first, and failing that `matplotlib.extern.dateutil` is used.

2 and 3 are undesirable as they still require maintaining copies of these packages in our tree -- and this is exacerbated by the fact that they are used less -- only in the binary installers. None of these 3 approaches address Numpy, which will still have to be manually installed using an installer.

TODO: How does this relate to the Mac OS-X installer?

39.5.6 Backward compatibility

At present, matplotlib can be installed from source on a machine without the third party dependencies and without an internet connection. After this change, an internet connection (and a working PyPI) will be required to install matplotlib for the first time. (Subsequent matplotlib updates or development work will run without accessing the network).

39.5.7 Alternatives

Distributing binary eggs doesn't feel like a usable solution. That requires getting easy_install installed first, and Windows users generally prefer the well known `.exe` or `.msi` installer that works out of the box.

39.6 MEP12: Improve Gallery and Examples

39.6.1 Status

Progress

Initial changes added in 1.3. Conversion of the gallery is on-going. 29 September 2015 - The last `pylab_examples` where *`pylab`* is imported has been converted over to use *`matplotlib.pyplot`* and `numpy`.

39.6.2 Branches and Pull requests

#1623, #1924, #2181

PR #2474 demonstrates a single example being cleaned up and moved to the appropriate section.

39.6.3 Abstract

Reorganizing the matplotlib plot gallery would greatly simplify navigation of the gallery. In addition, examples should be cleaned-up and simplified for clarity.

39.6.4 Detailed description

The matplotlib gallery was recently set up to split examples up into sections. As discussed in that PR[1], the current example sections (`api`, `pylab_examples`) aren't terribly useful to users: New sections in the gallery would help users find relevant examples.

These sections would also guide a cleanup of the examples: Initially, all the current examples would remain and be listed under their current directories. Over time, these examples could be cleaned up and moved into one of the new sections.

This process allows users to easily identify examples that need to be cleaned up; i.e. anything in the `api` and `pylab_examples` directories.

39.6.5 Implementation

1. Create new gallery sections. [Done]
2. Clean up examples and move them to the new gallery sections (over the course of many PRs and with the help of many users/developers). [In progress]

Gallery sections

The naming of sections is critical and will guide the clean-up effort. The current sections are:

- Lines, bars, and markers (more-or-less 1D data)
- Shapes and collections
- Statistical plots
- Images, contours, and fields
- Pie and polar charts: Round things
- Color
- Text, labels, and annotations
- Ticks and spines
- Subplots, axes, and figures
- Specialty plots (e.g., sankey, radar, tornado)
- Showcase (plots with tweaks to make them publication-quality)
- separate sections for toolboxes (already exists: 'mplot3d', 'axes_grid', 'units', 'widgets')

These names are certainly up for debate. As these sections grow, we should reevaluate them and split them up as necessary.

[1] https://github.com/matplotlib/matplotlib/pull/714

Clean up guidelines

The current examples in the `api` and `pylab_examples` sections of the gallery would remain in those directories until they are cleaned up. After clean-up, they would be moved to one of the new gallery sections described above. "Clean-up" should involve:

- sphinx-gallery docstrings: a title and a description of the example formatted as follows, at the top of the example:

```
"""
==============================
Colormaps alter your perception
==============================

Here I plot the function

.. math:: f(x, y) = \sin(x) + \cos(y)

with different colormaps. Look at how colormaps alter your perception!
"""
```

- PEP8 clean-ups (running flake8, or a similar checker, is highly recommended)
- Commented-out code should be removed.
- Replace uses of *pylab* interface with *pyplot* (+ `numpy`, etc.). See c25ef1e
- Remove shebang line, e.g.:

  #!/usr/bin/env python

- Use consistent imports. In particular:

  import numpy as np

  import matplotlib.pyplot as plt

  Avoid importing specific functions from these modules (e.g. `from numpy import sin`)

- Each example should focus on a specific feature (excluding `showcase` examples, which will show more "polished" plots). Tweaking unrelated to that feature should be removed. See f7b2217, e57b5fc, and 1458aa8

Use of *pylab* should be demonstrated/discussed on a dedicated help page instead of the gallery examples.

Note: When moving an existing example, you should search for references to that example. For example, the API documentation for `axes.py` and `pyplot.py` may use these examples to generate plots. Use your favorite search tool (e.g., grep, ack, grin, pss) to search the matplotlib package. See 2dc9a46 and aa6b410

Additional suggestions

- Provide links (both ways) between examples and API docs for the methods/objects used. (issue #2222)
- Use `plt.subplots` (note trailing "s") in preference over `plt.subplot`.
- Rename the example to clarify it's purpose. For example, the most basic demo of `imshow` might be `imshow_demo.py`, and one demonstrating different interpolation settings would be `imshow_demo_interpolation.py` (*not* `imshow_demo2.py`).
- Split up examples that try to do too much. See 5099675 and fc2ab07
- Delete examples that don't show anything new.
- Some examples exercise esoteric features for unit testing. These tweaks should be moved out of the gallery to an example in the `unit` directory located in the root directory of the package.
- Add plot titles to clarify intent of the example. See bd2b13c

39.6.6 Backward compatibility

The website for each Matplotlib version is readily accessible, so users who want to refer to old examples can still do so.

39.6.7 Alternatives

Tags

Tagging examples will also help users search the example gallery. Although tags would be a big win for users with specific goals, the plot gallery will remain the entry point to these examples, and sections could really help users navigate the gallery. Thus, tags are complementary to this reorganization.

39.7 MEP13: Use properties for Artists

39.7.1 Status

- **Discussion**

39.7.2 Branches and Pull requests

None

39.7.3 Abstract

Wrap all of the matplotlib getter and setter methods with python properties, allowing them to be read and written like class attributes.

39.7.4 Detailed description

Currently matplotlib uses getter and setter functions (usually prefixed with get_ and set_, respectively) for reading and writing data related to classes. However, since 2.6 python supports properties, which allow such setter and getter functions to be accessed as though they were attributes. This proposal would implement all existing setter and getter methods as properties.

39.7.5 Implementation

1. All existing getter and setter methods will need to have two aliases, one with the get_ or set_ prefix and one without. Getter methods that currently lack prefixes should be recording in a text file.
2. Classes should be reorganized so setter and getter methods are sequential in the code, with getter methods first.

3. Getter and setter methods the provide additional optional optional arguments should have those arguments accessible in another manner, either as additional getter or setter methods or attributes of other classes. If those classes are not accessible, getters for them should be added.
4. Property decorators will be added to the setter and getter methods without the prefix. Those with the prefix will be marked as deprecated.
5. Docstrings will need to be rewritten so the getter with the prefix has the current docstring and the getter without the prefix has a generic docstring appropriate for an attribute.
6. Automatic alias generation will need to be modified so it will also create aliases for the properties.
7. All instances of getter and setter method calls will need to be changed to attribute access.
8. All setter and getter aliases with prefixes will be removed

The following steps can be done simultaneously: 1, 2, and 3; 4 and 5; 6 and 7.

Only the following steps must be done in the same release: 4, 5, and 6. All other changes can be done in separate releases. 8 should be done several major releases after everything else.

39.7.6 Backward compatibility

All existing getter methods that do not have a prefix (such as get_) will need to be changed from function calls to attribute access. In most cases this will only require removing the parenthesis.

setter and getter methods that have additional optional arguments will need to have those arguments implemented in another way, either as a separate property in the same class or as attributes or properties of another class.

Cases where the setter returns a value will need to be changed to using the setter followed by the getter.

Cases where there are set_ATTR_on() and set_ATTR_off() methods will be changed to ATTR_on properties.

39.7.7 Examples

axes.Axes.set_axis_off/set_axis_on

Current implementation:

```
axes.Axes.set_axis_off()
axes.Axes.set_axis_on()
```

New implementation:

```
True = axes.Axes.axis_on
False = axes.Axes.axis_on
axes.Axes.axis_on = True
axes.Axes.axis_on = False
```

axes.Axes.get_xlim/set_xlim and get_autoscalex_on/set_autoscalex_on

Current implementation:

```
[left, right] = axes.Axes.get_xlim()
auto = axes.Axes.get_autoscalex_on()

[left, right] = axes.Axes.set_xlim(left=left, right=right, emit=emit, auto=auto)
[left, right] = axes.Axes.set_xlim(left=left, right=None, emit=emit, auto=auto)
[left, right] = axes.Axes.set_xlim(left=None, right=right, emit=emit, auto=auto)
[left, right] = axes.Axes.set_xlim(left=left, emit=emit, auto=auto)
[left, right] = axes.Axes.set_xlim(right=right, emit=emit, auto=auto)

axes.Axes.set_autoscalex_on(auto)
```

New implementation:

```
[left, right] = axes.Axes.axes_xlim
auto = axes.Axes.autoscalex_on

axes.Axes.axes_xlim = [left, right]
axes.Axes.axes_xlim = [left, None]
axes.Axes.axes_xlim = [None, right]
axes.Axes.axes_xlim[0] = left
axes.Axes.axes_xlim[1] = right

axes.Axes.autoscalex_on = auto

axes.Axes.emit_xlim = emit
```

axes.Axes.get_title/set_title

Current implementation:

```
string = axes.Axes.get_title()
axes.Axes.set_title(string, fontdict=fontdict, **kwargs)
```

New implementation:

```
string = axes.Axes.title
string = axes.Axes.title_text.text

```

(continues on next page)

(continued from previous page)

```
text.Text = axes.Axes.title_text
text.Text.<attribute> = attribute
text.Text.fontdict = fontdict

axes.Axes.title = string
axes.Axes.title = text.Text
axes.Axes.title_text = string
axes.Axes.title_text = text.Text
```

axes.Axes.get_xticklabels/set_xticklabels

Current implementation:

```
[text.Text] = axes.Axes.get_xticklabels()
[text.Text] = axes.Axes.get_xticklabels(minor=False)
[text.Text] = axes.Axes.get_xticklabels(minor=True)
[text.Text] = axes.Axes.([string], fontdict=None, **kwargs)
[text.Text] = axes.Axes.([string], fontdict=None, minor=False, **kwargs)
[text.Text] = axes.Axes.([string], fontdict=None, minor=True, **kwargs)
```

New implementation:

```
[text.Text] = axes.Axes.xticklabels
[text.Text] = axes.Axes.xminorticklabels
axes.Axes.xticklabels = [string]
axes.Axes.xminorticklabels = [string]
axes.Axes.xticklabels = [text.Text]
axes.Axes.xminorticklabels = [text.Text]
```

39.7.8 Alternatives

Instead of using decorators, it is also possible to use the property function. This would change the procedure so that all getter methods that lack a prefix will need to be renamed or removed. This makes handling docstrings more difficult and harder to read.

It is not necessary to deprecate the setter and getter methods, but leaving them in will complicate the code.

This could also serve as an opportunity to rewrite or even remove automatic alias generation.

Another alternate proposal:

Convert `set_xlim`, `set_xlabel`, `set_title`, etc. to `xlim`, `xlabel`, `title`,... to make the transition from `plt` functions to `axes` methods significantly simpler. These would still be methods, not properties, but it's still a great usability enhancement while retaining the interface.

39.8 MEP14: Text handling

39.8.1 Status

- **Discussion**

39.8.2 Branches and Pull requests

Issue #253 demonstrates a bug where using the bounding box rather than the advance width of text results in misaligned text. This is a minor point in the grand scheme of things, but it should be addressed as part of this MEP.

39.8.3 Abstract

By reorganizing how text is handled, this MEP aims to:

- improve support for Unicode and non-ltr languages
- improve text layout (especially multi-line text)
- allow support for more fonts, especially non-Apple-format TrueType fonts and OpenType fonts.
- make the font configuration easier and more transparent

39.8.4 Detailed description

Text layout

At present, matplotlib has two different ways to render text: "built-in" (based on FreeType and our own Python code), and "usetex" (based on calling out to a TeX installation). Adjunct to the "built-in" renderer there is also the Python-based "mathtext" system for rendering mathematical equations using a subset of the TeX language without having a TeX installation available. Support for these two engines in strewn about many source files, including every backend, where one finds clauses like

```
if rcParams['text.usetex']: # do one thing else: # do another
```

Adding a third text rendering approach (more on that later) would require editing all of these places as well, and therefore doesn't scale.

Instead, this MEP proposes adding a concept of "text engines", where the user could select one of many different approaches for rendering text. The implementations of each of these would be localized to their own set of modules, and not have little pieces around the whole source tree.

Why add more text rendering engines? The "built-in" text rendering has a number of shortcomings.

- It only handles right-to-left languages, and doesn't handle many special features of Unicode, such as combining diacriticals.
- The multiline support is imperfect and only supports manual line-breaking -- it can not break up a paragraph into lines of a certain length.
- It also does not handle inline formatting changes in order to support something like Markdown, reStructuredText or HTML. (Though rich-text formatting is contemplated in this MEP, since we want to make sure this design allows it, the specifics of a rich-text formatting implementation is outside of the scope of this MEP.)

Supporting these things is difficult, and is the "full-time job" of a number of other projects:

- pango/harfbuzz
- QtTextLayout
- Microsoft DirectWrite
- Apple Core Text

Of the above options, it should be noted that harfbuzz is designed from the start as a cross platform option with minimal dependencies, so therefore is a good candidate for a single option to support.

Additionally, for supporting rich text, we could consider using WebKit, and possibly whether than represents a good single cross-platform option. Again, however, rich text formatting is outside of the scope of this project.

Rather than trying to reinvent the wheel and add these features to matplotlib's "built-in" text renderer, we should provide a way to leverage these projects to get more powerful text layout. The "built-in" renderer will still need to exist for reasons of ease of installation, but its feature set will be more limited compared to the others. [TODO: This MEP should clearly decide what those limited features are, and fix any bugs to bring the implementation into a state of working correctly in all cases that we want it to work. I know @leejjoon has some thoughts on this.]

Font selection

Going from an abstract description of a font to a file on disk is the task of the font selection algorithm -- it turns out to be much more complicated than it seems at first.

The "built-in" and "usetex" renderers have very different ways of handling font selection, given their different technologies. TeX requires the installation of TeX-specific font packages, for example, and can not use TrueType fonts directly. Unfortunately, despite the different semantics for font selection, the same set of font properties are used for each. This is true of both the *FontProperties* class and the font-related *rcParams* (which basically share the same code underneath). Instead, we should define a core set of font selection parameters that will work across all text engines, and have engine-specific configuration to allow the user to do engine-specific things when required. For example, it is possible to directly select a font by name in the "built-in" using `rcParams["font.family"]` (default: `['sans-serif']`), but the same is not possible with "usetex". It may be possible to make it easier to use TrueType fonts by using XeTeX, but users will still want to use the traditional metafonts through TeX font packages. So the issue still stands that different text engines will need engine-specific configuration, and it should be more obvious to the user which configuration will work across text engines and which are engine-specific.

Note that even excluding "usetex", there are different ways to find fonts. The default is to use the font list cache in *font_manager* which matches fonts using our own algorithm based on the CSS font matching algorithm. It doesn't always do the same thing as the native font selection algorithms on Linux (fontconfig), Mac and Windows, and it doesn't always find all of the fonts on the system that the OS would normally pick up. However, it is cross-platform, and always finds the fonts that ship with matplotlib. The Cairo and MacOSX backends (and presumably a future HTML5-based backend) currently bypass this mechanism and use the OS-native ones. The same is true when not embedding fonts in SVG, PS or PDF files and opening them in a third-party viewer. A downside there is that (at least with Cairo, need to confirm with MacOSX) they don't always find the fonts we ship with matplotlib. (It may be possible to add the fonts to their search path, though, or we may need to find a way to install our fonts to a location the OS expects to find them).

There are also special modes in the PS and PDF to only use the core fonts that are always available to those formats. There, the font lookup mechanism must only match against those fonts. It is unclear whether the OS-native font lookup systems can handle this case.

There is also experimental support for using fontconfig for font selection in matplotlib, turned off by default. fontconfig is the native font selection algorithm on Linux, but is also cross platform and works well on the other platforms (though obviously is an

additional dependency there).

Many of the text layout libraries proposed above (pango, QtTextLayout, DirectWrite and CoreText etc.) insist on using the font selection library from their own ecosystem.

All of the above seems to suggest that we should move away from our self-written font selection algorithm and use the native APIs where possible. That's what Cairo and MacOSX backends already want to use, and it will be a requirement of any complex text layout library. On Linux, we already have the bones of a fontconfig implementation (which could also be accessed through pango). On Windows and Mac we may need to write custom wrappers. The nice thing is that the API for font lookup is relatively small, and essentially consist of "given a dictionary of font properties, give me a matching font file".

Font subsetting

Font subsetting is currently handled using ttconv. ttconv was a standalone commandline utility for converting TrueType fonts to subsetted Type 3 fonts (among other features) written in 1995, which matplotlib (well, I) forked in order to make it work as a library. It only handles Apple-style TrueType fonts, not ones with the Microsoft (or other vendor) encodings. It doesn't handle OpenType fonts at all. This means that even though the STIX fonts come as .otf files, we have to convert them to .ttf files to ship them with matplotlib. The Linux packagers hate this -- they'd rather just depend on the upstream STIX fonts. ttconv has also been shown to have a few bugs that have been difficult to fix over time.

Instead, we should be able to use FreeType to get the font outlines and write our own code (probably in Python) to output subsetted fonts (Type 3 on PS and PDF and paths on SVG). Freetype, as a popular and well-maintained project, handles a wide variety of fonts in the wild. This would remove a lot of custom C code, and remove some code duplication between backends.

Note that subsetting fonts this way, while the easiest route, does lose the hinting in the font, so we will need to continue, as we do now, provide a way to embed the entire font in the file where possible.

Alternative font subsetting options include using the subsetting built-in to Cairo (not clear if it can be used without the rest of Cairo), or using fontforge (which is a heavy and not terribly cross-platform dependency).

Freetype wrappers

Our FreeType wrapper could really use a reworking. It defines its own image buffer class (when a Numpy array would be easier). While FreeType can handle a huge diversity of font files, there are limitations to our wrapper that make it much harder to support non-Apple-vendor TrueType files, and certain features of OpenType files. (See #2088 for a terrible result of this, just to support the fonts that ship with Windows 7 and 8). I think a fresh rewrite of this wrapper would go a long way.

Text anchoring and alignment and rotation

The handling of baselines was changed in 1.3.0 such that the backends are now given the location of the baseline of the text, not the bottom of the text. This is probably

the correct behavior, and the MEP refactoring should also follow this convention.

In order to support alignment on multi-line text, it should be the responsibility of the (proposed) text engine to handle text alignment. For a given chunk of text, each engine calculates a bounding box for that text and the offset of the anchor point within that box. Therefore, if the va of a block was "top", the anchor point would be at the top of the box.

Rotating of text should always be around the anchor point. I'm not sure that lines up with current behavior in matplotlib, but it seems like the sanest/least surprising choice. [This could be revisited once we have something working]. Rotation of text should not be handled by the text engine -- that should be handled by a layer between the text engine and the rendering backend so it can be handled in a uniform way. [I don't see any advantage to rotation being handled by the text engines individually...]

There are other problems with text alignment and anchoring that should be resolved as part of this work. [TODO: enumerate these].

Other minor problems to fix

The mathtext code has backend-specific code -- it should instead provide its output as just another text engine. However, it's still desirable to have mathtext layout inserted as part of a larger layout performed by another text engine, so it should be possible to do this. It's an open question whether embedding the text layout of an arbitrary text engine in another should be possible.

The text mode is currently set by a global rcParam ("text.usetex") so it's either all on or all off. We should continue to have a global rcParam to choose the text engine ("text.layout_engine"), but it should under the hood be an overridable property on the *Text* object, so the same figure can combine the results of multiple text layout engines if necessary.

39.8.5 Implementation

A concept of a "text engine" will be introduced. Each text engine will implement a number of abstract classes. The `TextFont` interface will represent text for a given set of font properties. It isn't necessarily limited to a single font file -- if the layout engine supports rich text, it may handle a number of font files in a family. Given a `TextFont` instance, the user can get a `TextLayout` instance, which represents the layout for a given string of text in a given font. From a `TextLayout`, an iterator over `TextSpans` is returned so the engine can output raw editable text using as few spans as possible. If the engine would rather get individual characters, they can be obtained from the `TextSpan` instance:

```
class TextFont(TextFontBase):
    def __init__(self, font_properties):
        """
        Create a new object for rendering text using the given font properties.
        """
        pass
```

(continues on next page)

(continued from previous page)

```
    def get_layout(self, s, ha, va):
        """
        Get the TextLayout for the given string in the given font and
        the horizontal (left, center, right) and verticalalignment (top,
        center, baseline, bottom)
        """
        pass

class TextLayout(TextLayoutBase):
    def get_metrics(self):
        """
        Return the bounding box of the layout, anchored at (0, 0).
        """
        pass

    def get_spans(self):
        """
        Returns an iterator over the spans of different in the layout.
        This is useful for backends that want to editable raw text as
        individual lines.  For rich text where the font may change,
        each span of different font type will have its own span.
        """
        pass

    def get_image(self):
        """
        Returns a rasterized image of the text.  Useful for raster backends,
        like Agg.

        In all likelihood, this will be overridden in the backend, as it can
        be created from get_layout(), but certain backends may want to
        override it if their library provides it (as freetype does).
        """
        pass

    def get_rectangles(self):
        """
        Returns an iterator over the filled black rectangles in the layout.
        Used by TeX and mathtext for drawing, for example, fraction lines.
        """
        pass

    def get_path(self):
        """
        Returns a single Path object of the entire laid out text.

        [Not strictly necessary, but might be useful for textpath
        functionality]
        """
        pass
```

(continues on next page)

(continued from previous page)

```
class TextSpan(TextSpanBase):
    x, y      # Position of the span -- relative to the text layout as a whole
              # where (0, 0) is the anchor.  y is the baseline of the span.
    fontfile  # The font file to use for the span
    text      # The text content of the span

    def get_path(self):
        pass  # See TextLayout.get_path

    def get_chars(self):
        """
        Returns an iterator over the characters in the span.
        """
        pass

class TextChar(TextCharBase):
    x, y      # Position of the character -- relative to the text layout as
              # a whole, where (0, 0) is the anchor.  y is in the baseline
              # of the character.
    codepoint # The unicode code point of the character -- only for informational
              # purposes, since the mapping of codepoint to glyph_id may have been
              # handled in a complex way by the layout engine.  This is an int
              # to avoid problems on narrow Unicode builds.
    glyph_id  # The index of the glyph within the font
    fontfile  # The font file to use for the char

    def get_path(self):
        """
        Get the path for the character.
        """
pass
```

Graphic backends that want to output subset of fonts would likely build up a file-global dictionary of characters where the keys are (fontname, glyph_id) and the values are the paths so that only one copy of the path for each character will be stored in the file.

Special casing: The "usetex" functionality currently is able to get Postscript directly from TeX to insert directly in a Postscript file, but for other backends, parses a DVI file and generates something more abstract. For a case like this, `TextLayout` would implement `get_spans` for most backends, but add `get_ps` for the Postscript backend, which would look for the presence of this method and use it if available, or fall back to `get_spans`. This kind of special casing may also be necessary, for example, when the graphics backend and text engine belong to the same ecosystem, e.g. Cairo and Pango, or MacOSX and CoreText.

There are three main pieces to the implementation:

1) Rewriting the freetype wrapper, and removing ttconv.

a) Once (1) is done, as a proof of concept, we can move to the upstream STIX .otf

fonts

b) Add support for web fonts loaded from a remote URL. (Enabled by using freetype for font subsetting).

2) Refactoring the existing "builtin" and "usetex" code into separate text engines and to follow the API outlined above.

3) Implementing support for advanced text layout libraries.

(1) and (2) are fairly independent, though having (1) done first will allow (2) to be simpler. (3) is dependent on (1) and (2), but even if it doesn't get done (or is postponed), completing (1) and (2) will make it easier to move forward with improving the "builtin" text engine.

39.8.6 Backward compatibility

The layout of text with respect to its anchor and rotation will change in hopefully small, but improved, ways. The layout of multiline text will be much better, as it will respect horizontal alignment. The layout of bidirectional text or other advanced Unicode features will now work inherently, which may break some things if users are currently using their own workarounds.

Fonts will be selected differently. Hacks that used to sort of work between the "builtin" and "usetex" text rendering engines may no longer work. Fonts found by the OS that weren't previously found by matplotlib may be selected.

39.8.7 Alternatives

TBD

39.9 MEP15 - Fix axis autoscaling when limits are specified for one axis only

39.9.1 Status

Discussion

39.9.2 Branches and Pull requests

None so far.

39.9.3 Abstract

When one axis of a 2-dimensional plot if overridden via *set_xlim* or *set_ylim*, automatic scaling of the remaining axis should be based on the data that falls within the specified limits of the first axis.

39.9.4 Detailed description

When axis limits for a 2-D plot are specified for one axis only (via *set_xlim* or *set_ylim*), matplotlib currently does not currently rescale the other axis. The result is that the displayed curves or symbols may be compressed into a tiny portion of the available area, so that the final plot conveys much less information than it would with appropriate axis scaling.

The proposed change of behavior would make matplotlib choose the scale for the remaining axis using only the data that falls within the limits for the axis where limits were specified.

39.9.5 Implementation

I don't know enough about the internals of matplotlib to be able to suggest an implementation.

39.9.6 Backward compatibility

From the standpoint of software interfaces, there would be no break in backward compatibility. Some outputs would be different, but if the user truly desires the previous behavior, he/she can achieve this by overriding the axis scaling for both axes.

39.9.7 Alternatives

The only alternative that I can see is to maintain the status quo.

39.10 MEP19: Continuous Integration

39.10.1 Status

Completed

39.10.2 Branches and Pull requests

39.10.3 Abstract

matplotlib could benefit from better and more reliable continuous integration, both for testing and building installers and documentation.

39.10.4 Detailed description

Current state-of-the-art

Testing

matplotlib currently uses Travis-CI for automated tests. While Travis-CI should be praised for how much it does as a free service, it has a number of shortcomings:

- It often fails due to network timeouts when installing dependencies.
- It often fails for inexplicable reasons.
- build or test products can only be saved from build off of branches on the main repo, not pull requests, so it is often difficult to "post mortem" analyse what went wrong. This is particularly frustrating when the failure can not be subsequently reproduced locally.
- It is not extremely fast. matplotlib's cpu and memory requirements for testing are much higher than the average Python project.
- It only tests on Ubuntu Linux, and we have only minimal control over the specifics of the platform. It can be upgraded at any time outside of our control, causing unexpected delays at times that may not be convenient in our release schedule.

On the plus side, Travis-CI's integration with github -- automatically testing all pending pull requests -- is exceptional.

Builds

There is no centralized effort for automated binary builds for matplotlib. However, the following disparate things are being done [If the authors mentioned here could fill in detail, that would be great!]:

- @sandrotosi: builds Debian packages
- @takluyver: Has automated Ubuntu builds on Launchpad
- @cgohlke: Makes Windows builds (don't know how automated that is)
- @r-owen: Makes OS-X builds (don't know how automated that is)

Documentation

Documentation of master is now built by travis and uploaded to http://matplotlib.org/devdocs/index.html

@NelleV, I believe, generates the docs automatically and posts them on the web to chart MEP10 progress.

Peculiarities of matplotlib

matplotlib has complex requirements that make testing and building more taxing than many other Python projects.

- The CPU time to run the tests is quite high. It puts us beyond the free accounts of many CI services (e.g. ShiningPanda)
- It has a large number of dependencies, and testing the full matrix of all combinations is impractical. We need to be clever about what space we test and guarantee to support.

Requirements

This section outlines the requirements that we would like to have.

1. Testing all pull requests by hooking into the GitHub API, as Travis-CI does
2. Testing on all major platforms: Linux, Mac OS-X, MS Windows (in that order of priority, based on user survey)
3. Retain the last n days worth of build and test products, to aid in post-mortem debugging.
4. Automated nightly binary builds, so that users can test the bleeding edge without installing a complete compilation environment.
5. Automated benchmarking. It would be nice to have a standard benchmark suite (separate from the tests) whose performance could be tracked over time, in different backends and platforms. While this is separate from building and testing, ideally it would run on the same infrastructure.

6. Automated nightly building and publishing of documentation (or as part of testing, to ensure PRs don't introduce documentation bugs). (This would not replace the static documentation for stable releases as a default).
7. The test systems should be manageable by multiple developers, so that no single person becomes a bottleneck. (Travis-CI's design does this well -- storing build configuration in the git repository, rather than elsewhere, is a very good design.)
8. Make it easy to test a large but sparse matrix of different versions of matplotlib's dependencies. The matplotlib user survey provides some good data as to where to focus our efforts: https://docs.google.com/spreadsheet/ccc?key=0AjrPjlTMRTwTdHpQS25pcTZIRWdqX0pNckNSU01sMHc
9. Nice to have: A decentralized design so that those with more obscure platforms can publish build results to a central dashboard.

39.10.5 Implementation

This part is yet-to-be-written.

However, ideally, the implementation would be a third-party service, to avoid adding system administration to our already stretched time. As we have some donated funds, this service may be a paid one if it offers significant time-saving advantages over free offerings.

39.10.6 Backward compatibility

Backward compatibility is not a major concern for this MEP. We will replace current tools and procedures with something better and throw out the old.

39.10.7 Alternatives

39.10.8 Hangout Notes

CI Infrastructure

- We like Travis and it will probably remain part of our arsenal in any event. The reliability issues are being looked into.
- Enable Amazon S3 uploads of testing products on Travis. This will help with post-mortem of failures (@mdboom is looking into this now).
- We want Mac coverage. The best bet is probably to push Travis to enable it for our project by paying them for a Pro account (since they don't otherwise allow testing on both Linux and Mac).
- We want Windows coverage. Shining Panda is an option there.

- Investigate finding or building a tool that would collect and synthesize test results from a number of sources and post it to GitHub using the GitHub API. This may be of general use to the Scipy community.
- For both Windows and Mac, we should document (or better yet, script) the process of setting up the machine for a build, and how to build binaries and installers. This may require getting information from Russel Owen and Christoph Gohlke. This is a necessary step for doing automated builds, but would also be valuable for a number of other reasons.

The test framework itself

- We should investigate ways to make it take less time
  - Eliminating redundant tests, if possible
  - General performance improvements to matplotlib will help
- We should be covering more things, particularly more backends
- We should have more unit tests, fewer integration tests, if possible

39.11 MEP21: color and cm refactor

39.11.1 Status

- **Discussion**: This MEP has not commenced yet, but here are some ongoing ideas which may become a part of this MEP:

39.11.2 Branches and Pull requests

39.11.3 Abstract

- color
  - tidy up the namespace
  - Define a "Color" class
  - make it easy to convert from one color type to another `` `hex -> RGB` ``, `` `RGB -> hex` ``, `` `HSV -> RGB` `` etc.
  - improve the construction of a colormap - the dictionary approach is archaic and overly complex (though incredibly powerful)
  - make it possible to interpolate between two or more color types in different modes, especially useful for construction of colormaps in HSV space for instance
- cm
  - rename the module to something more descriptive - mappables?

Overall, there are a lot of improvements that can be made with matplotlib color handling - managing backwards compatibility will be difficult as there are some badly named variables/modules which really shouldn't exist - but a clear path and message for migration should be available, with a large amount of focus on this in the API changes documentation.

39.11.4 Detailed description

39.11.5 Implementation

39.11.6 Backward compatibility

39.11.7 Alternatives

39.12 MEP22: Toolbar rewrite

39.12.1 Status

Progress

39.12.2 Branches and Pull requests

Previous work

- https://github.com/matplotlib/matplotlib/pull/1849
- https://github.com/matplotlib/matplotlib/pull/2557
- https://github.com/matplotlib/matplotlib/pull/2465

Pull Requests:

- Removing the NavigationToolbar classes https://github.com/matplotlib/matplotlib/pull/2740 **CLOSED**
- Keeping the NavigationToolbar classes https://github.com/matplotlib/matplotlib/pull/2759 **CLOSED**
- Navigation by events: https://github.com/matplotlib/matplotlib/pull/3652

39.12.3 Abstract

The main goal of this MEP is to make it easier to modify (add, change, remove) the way the user interacts with the figures.

The user interaction with the figure is deeply integrated within the Canvas and Toolbar. Making extremely difficult to do any modification.

This MEP proposes the separation of this interaction into Toolbar, Navigation and Tools to provide independent access and reconfiguration.

This approach will make easier to create and share tools among users. In the far future, we can even foresee a kind of Marketplace for `Tools` where the most popular can be added into the main distribution.

39.12.4 Detailed description

The reconfiguration of the Toolbar is complex, most of the time it requires a custom backend.

The creation of custom Tools sometimes interferes with the Toolbar, as example see https://github.com/matplotlib/matplotlib/issues/2694 also the shortcuts are hardcoded and again not easily modifiable https://github.com/matplotlib/matplotlib/issues/2699

The proposed solution is to take the actions out of the `Toolbar` and the shortcuts out of the `Canvas`. This actions and shortcuts will be in the form of `Tools`.

A new class `Navigation` will be the bridge between the events from the `Canvas` and `Toolbar` and redirect them to the appropriate `Tool`.

At the end the user interaction will be divided into three classes:

- NavigationBase: This class is instantiated for each FigureManager and connect the all user interactions with the Tools
- ToolbarBase: This existing class is relegated only as a GUI access to Tools.
- ToolBase: Is the basic definition of Tools.

39.12.5 Implementation

ToolBase(object)

Tools can have a graphical representation as the `SubplotTool` or not even be present in the Toolbar as `Quit`

The *`ToolBase`* has the following class attributes for configuration at definition time

- keymap = None: Key(s) to be used to trigger the tool
- description = '': Small description of the tool
- image = None: Image that is used in the toolbar

The following instance attributes are set at instantiation:

- name
- navigation

Methods

- trigger(self, event): This is the main method of the Tool, it is called when the Tool is triggered by: * Toolbar button click * keypress associated with the Tool Keymap * Call to navigation.trigger_tool(name)
- set_figure(self, figure): Set the figure and navigation attributes
- `destroy(self, *args)`: Destroy the `Tool` graphical interface (if exists)

Available Tools

- ToolQuit
- ToolEnableAllNavigation
- ToolEnableNavigation
- ToolToggleGrid
- ToolToggleFullScreen
- ToolToggleYScale
- ToolToggleXScale
- ToolHome
- ToolBack
- ToolForward
- SaveFigureBase
- ConfigureSubplotsBase

ToolToggleBase(ToolBase)

The *ToolToggleBase* has the following class attributes for configuration at definition time

- radio_group = None: Attribute to group 'radio' like tools (mutually exclusive)
- cursor = None: Cursor to use when the tool is active

The **Toggleable** Tools, can capture keypress, mouse moves, and mouse button press

It defines the following methods

- enable(self, event): Called by *ToolToggleBase.trigger* method
- disable(self, event): Called when the tool is untoggled
- toggled : **Property** True or False

Available Tools

- ToolZoom
- ToolPan

NavigationBase

Defines the following attributes

- canvas:
- **keypresslock: Lock to know if the `canvas` key_press_event` is**

  available and process it
- messagelock: Lock to know if the message is available to write

Public methods for User use:

- nav_connect(self, s, func): Connect to to navigation for events
- nav_disconnect(self, cid): Disconnect from navigation event
- message_event(self, message, sender=None): Emit a tool_message_event event
- active_toggle(self): **Property** The currently toggled tools or None
- get_tool_keymap(self, name): Return a list of keys that are associated with the tool
- set_tool_keymap(self, name, `*keys`): Set the keys for the given tool
- remove_tool(self, name): Removes tool from the navigation control.
- add_tools(self, tools): Add multiple tools to `Navigation`
- add_tool(self, name, tool, group=None, position=None): Add a tool to the Navigation
- tool_trigger_event(self, name, sender=None, canvasevent=None, data=None): Trigger a tool and fire the event
- tools(self) **Property**: Return a dict with available tools with corresponding keymaps, descriptions and objects
- get_tool(self, name): Return the tool object

ToolbarBase

Methods for Backend implementation

- add_toolitem(self, name, group, position, image, description, toggle): Add a toolitem to the toolbar. This method is a callback from `tool_added_event` (emitted by navigation)
- set_message(self, s): Display a message on toolbar or in status bar
- toggle_toolitem(self, name): Toggle the toolitem without firing event.
- remove_toolitem(self, name): Remove a toolitem from the `Toolbar`

39.12.6 Backward compatibility

For backward compatibility added 'navigation' to the list of values supported by `rcParams["toolbar"]` (default: `'toolbar2'`), that is used for Navigation classes instantiation instead of the NavigationToolbar classes

With this parameter, it makes it transparent to anyone using the existing backends.

[@pelson comment: This also gives us an opportunity to avoid needing to implement all of this in the same PR - some backends can potentially exist without the new functionality for a short while (but it must be done at some point).]

39.13 MEP23: Multiple Figures per GUI window

39.13.1 Status

Discussion

39.13.2 Branches and Pull requests

Previous work - https://github.com/matplotlib/matplotlib/pull/2465 **To-delete**

39.13.3 Abstract

Add the possibility to have multiple figures grouped under the same `FigureManager`

39.13.4 Detailed description

Under the current structure, every canvas has its own window.

This is and may continue to be the desired method of operation for most use cases.

Sometimes when there are too many figures open at the same time, it is desirable to be able to group these under the same window [see](https://github.com/matplotlib/matplotlib/issues/2194).

The proposed solution modifies *`FigureManagerBase`* to contain and manage more than one `canvas`. The settings parameter `rcParams["backend.multifigure"]` control when the **MultiFigure** behaviour is desired.

Note

It is important to note, that the proposed solution, assumes that the [MEP22](https://github.com/matplotlib/matplotlib/wiki/Mep22) is already in place. This is simply because the actual implementation of the `Toolbar` makes it pretty hard to switch between canvases.

39.13.5 Implementation

The first implementation will be done in GTK3 using a Notebook as canvas container.

FigureManagerBase

will add the following new methods

- `add_canvas`: To add a canvas to an existing `FigureManager` object
- `remove_canvas`: To remove a canvas from a `FigureManager` object, if it is the last one, it will be destroyed
- `move_canvas`: To move a canvas from one `FigureManager` to another.
- `set_canvas_title`: To change the title associated with a specific canvas container
- `get_canvas_title`: To get the title associated with a specific canvas container
- `get_active_canvas`: To get the canvas that is in the foreground and is subject to the gui events. There is no `set_active_canvas` because the active canvas, is defined when `show` is called on a `Canvas` object.

new_figure_manager

To control which `FigureManager` will contain the new figures, an extra optional parameter *figuremanager* will be added, this parameter value will be passed to `new_figure_manager_given_figure`

new_figure_manager_given_figure

- If *figuremanager* parameter is given, this `FigureManager` object will be used instead of creating a new one.
- If `rcParams['backend.multifigure']` is True: The last `FigureManager` object will be used instead of creating a new one.

NavigationBase

Modifies the `NavigationBase` to keep a list of canvases, directing the actions to the active one

39.13.6 Backward compatibility

For the **MultiFigure** properties to be visible, the user has to activate them directly setting `rcParams['backend.multifigure'] = True`

It should be backwards compatible for backends that adhere to the current *`FigureManagerBase`* structure even if they have not implemented the **MultiFigure** magic yet.

39.13.7 Alternatives

Instead of modifying the *FigureManagerBase* it could be possible to add a parallel class, that handles the cases where `rcParams['backend.multifigure'] = True`. This will warranty that there won't be any problems with custom made backends, but also makes bigger the code, and more things to maintain.

39.14 MEP24: negative radius in polar plots

39.14.1 Status

Discussion

39.14.2 Branches and Pull requests

None

39.14.3 Abstract

It is clear that polar plots need to be able to gracefully handle negative r values (not by clipping or reflection).

39.14.4 Detailed description

One obvious application that we should support is bB plots (see https://github.com/matplotlib/matplotlib/issues/1730#issuecomment-40815837), but this seems more generally useful (for example growth rate as a function of angle). The assumption in the current code (as I understand it) is that the center of the graph is `r==0`, however it would be good to be able to set the center to be at any `r` (with any value less than the offset clipped).

39.14.5 Implementation

39.14.6 Related Issues

#1730, #1603, #2203, #2133

39.14.7 Backward compatibility

39.14.8 Alternatives

39.15 MEP25: Serialization

39.15.1 Status

Rejected

This work is important, but this particular effort has stalled.

39.15.2 Branches and Pull requests

- development branches:
- related pull requests:

39.15.3 Abstract

This MEP aims at adding a serializable `Controller` objects to act as an `Artist` managers. Users would then communicate changes to an `Artist` via a `Controller`. In this way, functionality of the `Controller` objects may be added incrementally since each `Artist` is still responsible for drawing everything. The goal is to create an API that is usable both by graphing libraries requiring high-level descriptions of figures and libraries requiring low-level interpretations.

39.15.4 Detailed description

Matplotlib is a core plotting engine with an API that many users already understand. It's difficult/impossible for other graphing libraries to (1) get a complete figure description, (2) output raw data from the figure object as the user has provided it, (3) understand the semantics of the figure objects without heuristics, and (4) give matplotlib a complete figure description to visualize. In addition, because an `Artist` has no conception of its own semantics within the figure, it's difficult to interact with them in a natural way.

In this sense, matplotlib will adopt a standard Model-View-Controller (MVC) framework. The *Model* will be the user defined data, style, and semantics. The *Views* are the ensemble of each individual `Artist`, which are responsible for producing the final image based on the *model*. The *Controller* will be the `Controller` object managing its set of `Artist` objects.

The `Controller` must be able to export the information that it's carrying about the figure on command, perhaps via a `to_json` method or similar. Because it would be extremely extraneous to duplicate all of the information in the model with the controller, only user-specified information (data + style) are explicitly kept. If a user wants more information (defaults) from the view/model, it should be able to query for it.

- This might be annoying to do, non-specified kwargs are pulled from the rcParams object which is in turn created from reading a user specified file and can be dynamically changed at run time. I suppose we could keep a dict of default

defaults and compare against that. Not clear how this will interact with the style sheet [[MEP26]] - @tacaswell

Additional Notes:

- The "raw data" does not necessarily need to be a `list`, `ndarray`, etc. Rather, it can more abstractly just have a method to yield data when needed.
- Because the `Controller` will contain extra information that users may not want to keep around, it should *not* be created by default. You should be able to both (a) instantiate a `Controller` with a figure and (b) build a figure with a `Controller`.

Use Cases:

- Export all necessary informat
- Serializing a matplotlib figure, saving it, and being able to rerun later.
- Any other source sending an appropriately formatted representation to matplotlib to open

39.15.5 Examples

Here are some examples of what the controllers should be able to do.

1. Instantiate a matplotlib figure from a serialized representation (e.g., JSON):

```
import json
from matplotlib.controllers import Controller
with open('my_figure') as f:
    o = json.load(f)
c = Controller(o)
fig = c.figure
```

2. Manage artists from the controller (e.g., Line2D):

```
# not really sure how this should look
c.axes[0].lines[0].color = 'b'
# ?
```

3. Export serializable figure representation:

```
o = c.to_json()
# or... we should be able to throw a figure object in there too
o = Controller.to_json(mpl_fig)
```

39.15.6 Implementation

1. Create base `Controller` objects that are able to manage `Artist` objects (e.g., `Hist`)

   Comments:

   - initialization should happen via unpacking `**`, so we need a copy of call signature parameter for the `Artist` we're ultimately trying to control. Unfortunate hard-coded repetition...
   - should the additional `**kwargs` accepted by each `Artist` be tracked at the `Controller`
   - how does a `Controller` know which artist belongs where? E.g., do we need to pass `axes` references?

   Progress:

   - A simple NB demonstrating some functionality for `Line2DController` objects: https://nbviewer.jupyter.org/gist/theengineear/f0aa8d79f64325e767c0

2. Write in protocols for the `Controller` to *update* the model.

   Comments:

   - how should containers be dealt with? E.g., what happens to old patches when we re-bin a histogram?
   - in the link from (1), the old line is completely destroyed and re-drawn, what if something is referencing it?

3. Create method by which a json object can be assembled from the `Controllers`
4. Deal with serializing the unserializable aspects of a figure (e.g., non-affine transforms?)
5. Be able to instantiate from a serialized representation
6. Reimplement the existing pyplot and Axes method, e.g. `pyplot.hist` and `Axes.hist` in terms of the new controller class.

> @theengineer: in #2 above, what do you mean by *get updates* from each `Artist`?

^ Yup. The `Controller` *shouldn't* need to get updated. This just happens in #3. Delete comments when you see this.

39.15.7 Backward compatibility

- pickling will change
- non-affine transformations will require a defined pickling method

39.15.8 Alternatives

PR #3150 suggested adding semantics by parasitically attaching extra containers to axes objects. This is a more complete solution with what should be a more developed/flexible/powerful framework.

39.16 MEP26: Artist styling

39.16.1 Status

Rejected

39.16.2 Branches and Pull requests

39.16.3 Abstract

This MEP proposes a new stylesheet implementation to allow more comprehensive and dynamic styling of artists.

The current version of matplotlib (1.4.0) allows stylesheets based on the rcParams syntax to be applied before creation of a plot. The methodology below proposes a new syntax, based on CSS, which would allow styling of individual artists and properties, which can be applied dynamically to existing objects.

This is related to (and makes steps toward) the overall goal of moving to a DOM/tree-like architecture.

39.16.4 Detailed description

Currently, the look and appearance of existing artist objects (figure, axes, Line2D etc...) can only be updated via `set_` and `get_` methods on the artist object, which is quite laborious, especially if no reference to the artist(s) has been stored. The new style sheets introduced in 1.4 allow styling before a plot is created, but do not offer any means to dynamically update plots or distinguish between artists of the same type (i.e. to specify the `line color` and `line style` separately for differing *Line2D* objects).

The initial development should concentrate on allowing styling of artist primitives (those *Artist*s that do not contain other *Artist*s), and further development could expand the CSS syntax rules and parser to allow more complex styling. See the appendix for a list of primitives.

The new methodology would require development of a number of steps:

- A new stylesheet syntax (likely based on CSS) to allow selection of artists by type, class, id etc...
- A mechanism by which to parse a stylesheet into a tree
- A mechanism by which to translate the parse-tree into something which can be used to update the properties of relevant artists. Ideally this would implement a method by which to traverse the artists in a tree-like structure.
- A mechanism by which to generate a stylesheet from existing artist properties. This would be useful to allow a user to export a stylesheet from an existing figure (where the appearance may have been set using the matplotlib API)...

39.16.5 Implementation

It will be easiest to allow a '3rd party' to modify/set the style of an artist if the 'style' is created as a separate class and store against the artist as a property. The `GraphicsContext` class already provides a the basis of a `Style` class and an artist's *`draw`* method can be refactored to use the `Style` class rather than setting up it's own `GraphicsContext` and transferring it's style-related properties to it. A minimal example of how this could be implemented is shown here: https://github.com/JamesRamm/mpl_experiment

IMO, this will also make the API and code base much neater as individual get/set methods for artist style properties are now redundant... Indirectly related would be a general drive to replace get/set methods with properties. Implementing the style class with properties would be a big stride toward this...

For initial development, I suggest developing a syntax based on a much (much much) simplified version of CSS. I am in favour of dubbing this Artist Style Sheets :+1: :

BNF Grammar

I propose a very simple syntax to implement initially (like a proof of concept), which can be expanded upon in the future. The BNF form of the syntax is given below and then explained

```
RuleSet ::= SelectorSequence "{"Declaration"}"

SelectorSequence :: = Selector {"," Selector}

Declaration ::= propName":" propValue";"

Selector ::= ArtistIdent{"#"Ident}

propName ::= Ident

propValue ::= Ident | Number | Colour | "None"
```

`ArtistIdent`, `Ident`, `Number` and `Colour` are tokens (the basic building blocks of the expression) which are defined by regular expressions.

Syntax

A CSS stylesheet consists of a series of **rule sets** in hierarchical order (rules are applied from top to bottom). Each rule follows the syntax

```
selector {attribute: value;}
```

Each rule can have any number of `attribute: value` pairs, and a stylesheet can have any number of rules.

The initial syntax is designed only for *Artist* primitives. It does not address the question of how to set properties on *Container* types (whose properties may themselves be *Artist*s with settable properties), however, a future solution to this could simply be nested `RuleSet`s

Selectors

Selectors define the object to which the attribute updates should be applied. As a starting point, I propose just 2 selectors to use in initial development:

Artist Type Selector

Select an *Artist* by it's type. E.g *Line2D* or *Text*:

```
Line2D {attribute: value}
```

The regex for matching the artist type selector (`ArtistIdent` in the BNF grammar) would be:

```
ArtistIdent = r'(?P<ArtistIdent>\bLine2D\b|\bText\b|\bAxesImage\b|\bFigureImage\b|\
↪bPatch\b)'
```

GID selector

Select an *Artist* by its `gid`:

```
Line2D#myGID {attribute: value}
```

A `gid` can be any string, so the regex could be as follows:

```
Ident = r'(?P<Ident>[a-zA-Z_][a-zA-Z_0-9]*)'
```

The above selectors roughly correspond to their CSS counterparts (http://www.w3.org/TR/CSS21/selector.html)

Attributes and values

- `Attributes` are any valid (settable) property for the *Artist* in question.
- `Values` are any valid value for the property (Usually a string, or number).

Parsing

Parsing would consist of breaking the stylesheet into tokens (the python cookbook gives a nice tokenizing recipe on page 66), applying the syntax rules and constructing a `Tree`. This requires defining the grammar of the stylesheet (again, we can borrow from CSS) and writing a parser. Happily, there is a recipe for this in the python cookbook as well.

Visitor pattern for matplotlib figure

In order to apply the stylesheet rules to the relevant artists, we need to 'visit' each artist in a figure and apply the relevant rule. Here is a visitor class (again, thanks to python cookbook), where each `node` would be an artist in the figure. A `visit_` method would need to be implemented for each mpl artist, to handle the different properties for each

```
class Visitor:
    def visit(self, node):
        name = 'visit_' + type(node).__name__
        meth = getattr(self, name, None)
        if meth is None:
            raise NotImplementedError
        return meth(node)
```

An `evaluator` class would then take the stylesheet rules and implement the visitor on each one of them.

39.16.6 Backward compatibility

Implementing a separate `Style` class would break backward compatibility as many get/set methods on an artist would become redundant. While it would be possible to alter these methods to hook into the `Style` class (stored as a property against the artist), I would be in favor of simply removing them to both neaten/simplify the code-base and to provide a simple, uncluttered API...

39.16.7 Alternatives

No alternatives, but some of the ground covered here overlaps with MEP25, which may assist in this development

39.16.8 Appendix

Matplotlib primitives

This will form the initial selectors which stylesheets can use.

- Line2D
- Text
- AxesImage
- FigureImage
- Patch

39.17 MEP27: decouple pyplot from backends

39.17.1 Status

Progress

39.17.2 Branches and Pull requests

Main PR (including GTK3): + https://github.com/matplotlib/matplotlib/pull/4143

Backend specific branch diffs: + https://github.com/OceanWolf/matplotlib/compare/backend-refactor...OceanWolf:backend-refactor-tkagg + https://github.com/OceanWolf/matplotlib/compare/backend-refactor...OceanWolf:backend-refactor-qt + https://github.com/OceanWolf/matplotlib/compare/backend-refactor...backend-refactor-wx

39.17.3 Abstract

This MEP refactors the backends to give a more structured and consistent API, removing generic code and consolidate existing code. To do this we propose splitting:

1. `FigureManagerBase` and its derived classes into the core functionality class `FigureManager` and a backend specific class `WindowBase` and
2. `ShowBase` and its derived classes into `Gcf.show_all` and `MainLoopBase`.

39.17.4 Detailed description

This MEP aims to consolidate the backends API into one single uniform API, removing generic code out of the backend (which includes `_pylab_helpers` and `Gcf`), and push code to a more appropriate level in matplotlib. With this we automatically remove inconsistencies that appear in the backends, such as `FigureManagerBase.resize(w, h)` which sometimes sets the canvas, and other times set the entire window to the dimensions given, depending on the backend.

Two main places for generic code appear in the classes derived from `FigureManagerBase` and `ShowBase`.

1. `FigureManagerBase` has **three** jobs at the moment:
    1. The documentation describes it as a ``*Helper class for pyplot mode, wraps everything up into a neat bundle*''
    2. But it doesn't just wrap the canvas and toolbar, it also does all of the windowing tasks itself. The conflation of these two tasks gets seen the best in the following line: `` `python self.set_window_title("Figure %d" % num) ` `` This combines backend specific code `self.set_window_title(title)` with matplotlib generic code `title = "Figure %d" % num`.
    3. Currently the backend specific subclass of `FigureManager` decides when to end the mainloop. This also seems very wrong as the figure should have no control over the other figures.
2. `ShowBase` has two jobs:

1. It has the job of going through all figure managers registered in `_pylab_helpers.Gcf` and telling them to show themselves.
2. And secondly it has the job of performing the backend specific `mainloop` to block the main programme and thus keep the figures from dying.

39.17.5 Implementation

The description of this MEP gives us most of the solution:

1. To remove the windowing aspect out of `FigureManagerBase` letting it simply wrap this new class along with the other backend classes. Create a new `WindowBase` class that can handle this functionality, with pass-through methods (:arrow_right:) to `WindowBase`. Classes that subclass `WindowBase` should also subclass the GUI specific window class to ensure backward compatibility (`manager.window == manager.window`).
2. Refactor the mainloop of `ShowBase` into `MainLoopBase`, which encapsulates the end of the loop as well. We give an instance of `MainLoop` to `FigureManager` as a key unlock the exit method (requiring all keys returned before the loop can die). Note this opens the possibility for multiple backends to run concurrently.
3. Now that `FigureManagerBase` has no backend specifics in it, to rename it to `FigureManager`, and move to a new file `backend_managers.py` noting that:
    1. This allows us to break up the conversion of backends into separate PRs as we can keep the existing `FigureManagerBase` class and its dependencies intact.
    2. and this also anticipates MEP22 where the new `NavigationBase` has morphed into a backend independent `ToolManager`.

FigureManagerBase(canvas, num)	FigureManager(figure, num)	`WindowBase(title)`	Notes
show		show	
destroy	calls destroy on all components	destroy	
full_screen_toggle	handles logic	set_fullscreen	
resize		resize	
key_press	key_press		
get_window_title		get_window_title	
set_window_title		set_window_title	
	_get_toolbar		A common method to all subclasses of FigureManagerBase
		set_default_size	
		add_element_to_window	

Show-Base	MainLoop-Base	Notes
main-loop	begin	
	end	Gets called automagically when no more instances of the subclass exist
__call__		Method moved to Gcf.show_all

39.17.6 Future compatibility

As eluded to above when discussing MEP 22, this refactor makes it easy to add in new generic features. At the moment, MEP 22 has to make ugly hacks to each class extending from `FigureManagerBase`. With this code, this only needs to get made in the single `FigureManager` class. This also makes the later deprecation of `NavigationToolbar2` very straightforward, only needing to touch the single `FigureManager` class

MEP 23 makes for another use case where this refactored code will come in very handy.

39.17.7 Backward compatibility

As we leave all backend code intact, only adding missing methods to existing classes, this should work seamlessly for all use cases. The only difference will lie for backends that used `FigureManager.resize` to resize the canvas and not the window, due to the standardisation of the API.

I would envision that the classes made obsolete by this refactor get deprecated and removed on the same timetable as `NavigationToolbar2`, also note that the change in call signature to the `FigureCanvasWx` constructor, while backward compatible, I think the old (imho ugly style) signature should get deprecated and removed in the same manner as everything else.

back-end	man-ager.resize(w,h)	Extra
gtk3	window	
Tk	canvas	
Qt	window	
Wx	canvas	FigureManagerWx had `frame` as an alias to window, so this also breaks BC.

39.17.8 Alternatives

If there were any alternative solutions to solving the same problem, they should be discussed here, along with a justification for the chosen approach.

39.17.9 Questions

Mdehoon: Can you elaborate on how to run multiple backends concurrently?

OceanWolf: @mdehoon, as I say, not for this MEP, but I see this MEP opens it up as a future possibility. Basically the `MainLoopBase` class acts a per backend Gcf, in this MEP it tracks the number of figures open per backend, and manages the mainloops for those backends. It closes the backend specific mainloop when it detects that no figures remain open for that backend. Because of this I imagine that with only a small amount of tweaking that we can do full-multi-backend matplotlib. No idea yet why one would want to, but I leave the possibility there in MainLoopBase. With all the backend-code specifics refactored out of `FigureManager` also aids in this, one manager to rule them (the backends) all.

Mdehoon: @OceanWolf, OK, thanks for the explanation. Having a uniform API for the backends is very important for the maintainability of matplotlib. I think this MEP is a step in the right direction.

39.18 MEP28: Remove Complexity from Axes.boxplot

39.18.1 Status

Discussion

39.18.2 Branches and Pull requests

The following lists any open PRs or branches related to this MEP:

1. Deprecate redundant statistical kwargs in `Axes.boxplot`: https://github.com/phobson/matplotlib/tree/MEP28-initial-deprecations
2. Deprecate redundant style options in `Axes.boxplot`: https://github.com/phobson/matplotlib/tree/MEP28-initial-deprecations
3. Deprecate passings 2D NumPy arrays as input: None
4. Add pre- & post-processing options to `cbook.boxplot_stats`: https://github.com/phobson/matplotlib/tree/boxplot-stat-transforms
5. Exposing `cbook.boxplot_stats` through `Axes.boxplot` kwargs: None
6. Remove redundant statistical kwargs in `Axes.boxplot`: None
7. Remove redundant style options in `Axes.boxplot`: None
8. Remaining items that arise through discussion: None

39.18.3 Abstract

Over the past few releases, the `Axes.boxplot` method has grown in complexity to support fully customizable artist styling and statistical computation. This lead to `Axes.boxplot` being split off into multiple parts. The statistics needed to draw a boxplot are computed in `cbook.boxplot_stats`, while the actual artists are drawn by `Axes.bxp`. The original method, `Axes.boxplot` remains as the most public API that handles passing the user-supplied data to `cbook.boxplot_stats`, feeding the results to `Axes.bxp`, and pre-processing style information for each facet of the boxplot plots.

This MEP will outline a path forward to rollback the added complexity and simplify the API while maintaining reasonable backwards compatibility.

39.18.4 Detailed description

Currently, the `Axes.boxplot` method accepts parameters that allow the users to specify medians and confidence intervals for each box that will be drawn in the plot. These were provided so that avdanced users could provide statistics computed in a different fashion that the simple method provided by matplotlib. However, handling this input requires complex logic to make sure that the forms of the data structure match what needs to be drawn. At the moment, that logic contains 9 separate if/else statements nested up to 5 levels deep with a for loop, and may raise up to 2 errors. These parameters were added prior to the creation of the `Axes.bxp` method, which draws boxplots from a list of dictionaries containing the relevant statistics. Matplotlib also provides a function that computes these statistics via `cbook.boxplot_stats`. Note that advanced users can now either a) write their own function to compute the stats required by `Axes.bxp`, or b) modify the output returned by `cbook.boxplots_stats` to fully customize the position of the artists of the plots. With this flexibility, the parameters to manually specify only the medians and their confidences intervals remain for backwards compatibility.

Around the same time that the two roles of `Axes.boxplot` were split into `cbook.boxplot_stats` for computation and `Axes.bxp` for drawing, both `Axes.boxplot` and `Axes.bxp` were written to accept parameters that individually toggle the drawing of all components of the boxplots, and parameters that individually configure the style of those artists. However, to maintain backwards compatibility, the `sym` parameter (previously used to specify the symbol of the fliers) was retained. This parameter itself requires fairly complex logic to reconcile the `sym` parameters with the newer `flierprops` parameter at the default style specified by `matplotlibrc`.

This MEP seeks to dramatically simplify the creation of boxplots for novice and advanced users alike. Importantly, the changes proposed here will also be available to downstream packages like seaborn, as seaborn smartly allows users to pass arbitrary dictionaries of parameters through the seaborn API to the underlying matplotlib functions.

This will be achieved in the following way:

1. `cbook.boxplot_stats` will be modified to allow pre- and post- computation transformation functions to be passed in (e.g., `np.log` and `np.exp` for lognormally distributed data)
2. `Axes.boxplot` will be modified to also accept and naïvely pass them to `cbook.boxplots_stats` (Alt: pass the stat function and a dict of its optional parameters).
3. Outdated parameters from `Axes.boxplot` will be deprecated and later removed.

Importance

Since the limits of the whiskers are computed arithmetically, there is an implicit assumption of normality in box and whisker plots. This primarily affects which data points are classified as outliers.

Allowing transformations to the data and the results used to draw boxplots will allow users to opt-out of that assumption if the data are known to not fit a normal distribution.

Below is an example of how `Axes.boxplot` classifies outliers of lognormal data differently depending one these types of transforms.

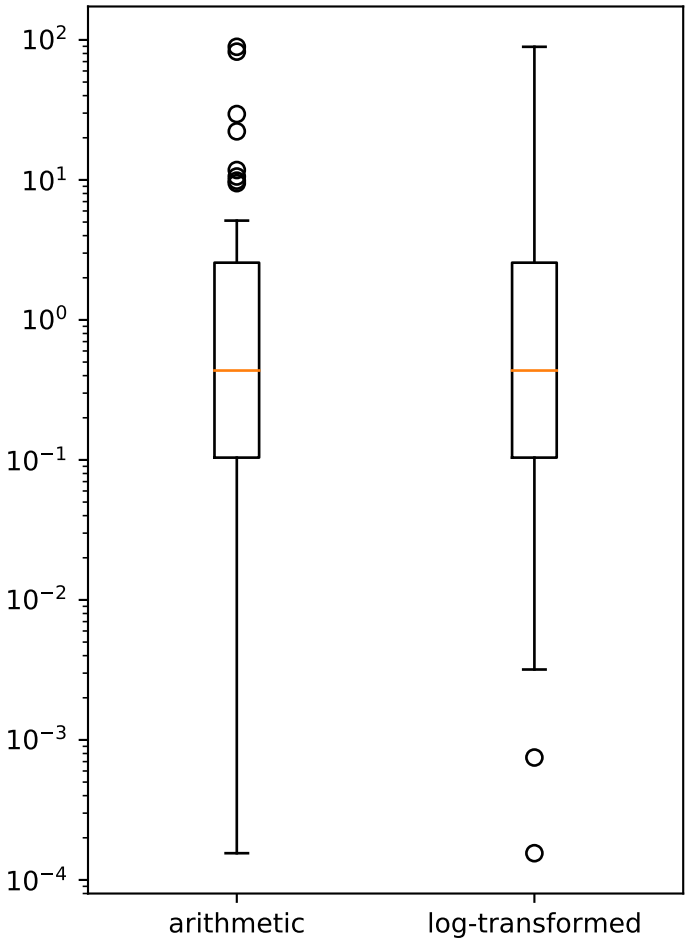
```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cbook
np.random.seed(0)

fig, ax = plt.subplots(figsize=(4, 6))
ax.set_yscale('log')
data = np.random.lognormal(-1.75, 2.75, size=37)

stats = cbook.boxplot_stats(data, labels=['arithmetic'])
logstats = cbook.boxplot_stats(np.log(data), labels=['log-transformed'])

for lsdict in logstats:
    for key, value in lsdict.items():
        if key != 'label':
            lsdict[key] = np.exp(value)

stats.extend(logstats)
ax.bxp(stats)
fig.show()
```

39.18.5 Implementation

Passing transform functions to `cbook.boxplots_stats`

This MEP proposes that two parameters (e.g., `transform_in` and `transform_out` be added to the cookbook function that computes the statistics for the boxplot function. These will be optional keyword-only arguments and can easily be set to `lambda x: x` as a no-op when omitted by the user. The `transform_in` function will be applied to the data as the `boxplot_stats` function loops through each subset of the data passed to it. After the list of statistics dictionaries are computed the `transform_out` function is applied to each value in the dictionaries.

These transformations can then be added to the call signature of `Axes.boxplot` with little impact to that method's complexity. This is because they can be directly passed to `cbook.boxplot_stats`. Alternatively, `Axes.boxplot` could be modified to accept an op-


10^2
10^1
10^0
10^{-1}
10^{-2}
10^{-3}
10^{-4}
arithmetic
log-transformed

tional statistical function kwarg and a dictionary of parameters to be directly passed to it.

At this point in the implementation users and external libraries like seaborn would have complete control via the `Axes.boxplot` method. More importantly, at the very least, seaborn would require no changes to its API to allow users to take advantage of these new options.

Simplifications to the `Axes.boxplot` API and other functions

Simplifying the boxplot method consists primarily of deprecating and then removing the redundant parameters. Optionally, a next step would include rectifying minor terminological inconsistencies between `Axes.boxplot` and `Axes.bxp`.

The parameters to be deprecated and removed include:

1. `usermedians` - processed by 10 SLOC, 3 `if` blocks, a `for` loop
2. `conf_intervals` - handled by 15 SLOC, 6 `if` blocks, a `for` loop
3. `sym` - processed by 12 SLOC, 4 `if` blocks

Removing the `sym` option allows all code in handling the remaining styling parameters to be moved to `Axes.bxp`. This doesn't remove any complexity, but does reinforce the single responsibility principle among `Axes.bxp`, `cbook.boxplot_stats`, and `Axes.boxplot`.

Additionally, the `notch` parameter could be renamed `shownotches` to be consistent with `Axes.bxp`. This kind of cleanup could be taken a step further and the `whis`, `bootstrap`, `autorange` could be rolled into the kwargs passed to the new `statfxn` parameter.

39.18.6 Backward compatibility

Implementation of this MEP would eventually result in the backwards incompatible deprecation and then removal of the keyword parameters `usermedians`, `conf_intervals`, and `sym`. Cursory searches on GitHub indicated that `usermedians`, `conf_intervals` are used by few users, who all seem to have a very strong knowledge of matplotlib. A robust deprecation cycle should provide sufficient time for these users to migrate to a new API.

Deprecation of `sym` however, may have a much broader reach into the matplotlib user-base.

Schedule

An accelerated timeline could look like the following:

1. v2.0.1 add transforms to `cbook.boxplots_stats`, expose in `Axes.boxplot`
2. v2.1.0 Initial Deprecations , and using 2D NumPy arrays as input
    a. Using 2D NumPy arrays as input. The semantics around 2D arrays are generally confusing.
    b. `usermedians`, `conf_intervals`, `sym` parameters
3. v2.2.0
    a. remove `usermedians`, `conf_intervals`, `sym` parameters
    b. deprecate `notch` in favor of `shownotches` to be consistent with other parameters and `Axes.bxp`
4. **v2.3.0**
    a. remove `notch` parameter
    b. move all style and artist toggling logic to `Axes.bxp` such `Axes.boxplot` is little more than a broker between `Axes.bxp` and `cbook.boxplots_stats`

Anticipated Impacts to Users

As described above deprecating `usermedians` and `conf_intervals` will likely impact few users. Those who will be impacted are almost certainly advanced users who will be able to adapt to the change.

Deprecating the `sym` option may import more users and effort should be taken to collect community feedback on this.

Anticipated Impacts to Downstream Libraries

The source code (GitHub master as of 2016-10-17) was inspected for seaborn and python-ggplot to see if these changes would impact their use. None of the parameters nominated for removal in this MEP are used by seaborn. The seaborn APIs that use matplotlib's boxplot function allow user's to pass arbitrary `**kwargs` through to matplotlib's API. Thus seaborn users with modern matplotlib installations will be able to take full advantage of any new features added as a result of this MEP.

Python-ggplot has implemented its own function to draw boxplots. Therefore, no impact can come to it as a result of implementing this MEP.

39.18.7 Alternatives

Variations on the theme

This MEP can be divided into a few loosely coupled components:

1. Allowing pre- and post-computation transformation function in `cbook.boxplot_stats`
2. Exposing that transformation in the `Axes.boxplot` API
3. Removing redundant statistical options in `Axes.boxplot`
4. Shifting all styling parameter processing from `Axes.boxplot` to `Axes.bxp`.

With this approach, #2 depends and #1, and #4 depends on #3.

There are two possible approaches to #2. The first and most direct would be to mirror the new `transform_in` and `transform_out` parameters of `cbook.boxplot_stats` in `Axes.boxplot` and pass them directly.

The second approach would be to add `statfxn` and `statfxn_args` parameters to `Axes.boxplot`. Under this implementation, the default value of `statfxn` would be `cbook.boxplot_stats`, but users could pass their own function. Then `transform_in` and `transform_out` would then be passed as elements of the `statfxn_args` parameter.

```
def boxplot_stats(data, ..., transform_in=None, transform_out=None):
    if transform_in is None:
        transform_in = lambda x: x

    if transform_out is None:
        transform_out = lambda x: x

    output = []
    for _d in data:
        d = transform_in(_d)
        stat_dict = do_stats(d)
        for key, value in stat_dict.item():
            if key != 'label':
                stat_dict[key] = transform_out(value)
        output.append(d)
    return output


class Axes(...):
    def boxplot_option1(data, ..., transform_in=None, transform_out=None):
        stats = cbook.boxplot_stats(data, ...,
                                    transform_in=transform_in,
                                    transform_out=transform_out)
        return self.bxp(stats, ...)

    def boxplot_option2(data, ..., statfxn=None, **statopts):
        if statfxn is None:
```

(continues on next page)

(continued from previous page)

```
            statfxn = boxplot_stats
        stats = statfxn(data, **statopts)
        return self.bxp(stats, ...)
```

Both cases would allow users to do the following:

```
fig, ax1 = plt.subplots()
artists1 = ax1.boxplot_optionX(data, transform_in=np.log,
                               transform_out=np.exp)
```

But Option Two lets a user write a completely custom stat function (e.g., `my_box_stats`) with fancy BCA confidence intervals and the whiskers set differently depending on some attribute of the data.

This is available under the current API:

```
fig, ax1 = plt.subplots()
my_stats = my_box_stats(data, bootstrap_method='BCA',
                        whisker_method='dynamic')
ax1.bxp(my_stats)
```

And would be more concise with Option Two

```
fig, ax = plt.subplots()
statopts = dict(transform_in=np.log, transform_out=np.exp)
ax.boxplot(data, ..., **statopts)
```

Users could also pass their own function to compute the stats:

```
fig, ax1 = plt.subplots()
ax1.boxplot(data, statfxn=my_box_stats, bootstrap_method='BCA',
            whisker_method='dynamic')
```

From the examples above, Option Two seems to have only marginal benefit, but in the context of downstream libraries like seaborn, its advantage is more apparent as the following would be possible without any patches to seaborn:

```
import seaborn
tips = seaborn.load_data('tips')
g = seaborn.factorplot(x="day", y="total_bill", hue="sex", data=tips,
                       kind='box', palette="PRGn", shownotches=True,
                       statfxn=my_box_stats, bootstrap_method='BCA',
                       whisker_method='dynamic')
```

This type of flexibility was the intention behind splitting the overall boxplot API in the current three functions. In practice however, downstream libraries like seaborn support versions of matplotlib dating back well before the split. Thus, adding just a bit more flexibility to the `Axes.boxplot` could expose all the functionality to users of the downstream libraries with modern matplotlib installation without intervention from the downstream library maintainers.

Doing less

Another obvious alternative would be to omit the added pre- and post- computation transform functionality in `cbook.boxplot_stats` and `Axes.boxplot`, and simply remove the redundant statistical and style parameters as described above.

Doing nothing

As with many things in life, doing nothing is an option here. This means we simply advocate for users and downstream libraries to take advantage of the split between `cbook.boxplot_stats` and `Axes.bxp` and let them decide how to provide an interface to that.

39.19 MEP29: Text light markup

39.19.1 Status

Discussion

39.19.2 Branches and Pull requests

None at the moment, proof of concept only.

39.19.3 Abstract

This MEP proposes to add lightweight markup to the text artist.

39.19.4 Detailed description

Using different size/color/family in a text annotation is difficult because the *text* method accepts argument for size/color/family/weight/etc. that are used for the whole text. But, if one wants, for example, to have different colors, one has to look at the gallery where one such example is provided: http://matplotlib.org/examples/text_labels_and_annotations/rainbow_text.html

This example takes a list of strings as well as a list of colors which makes it cumbersome to use. An alternative would be to use a restricted set of pango-like markup and to interpret this markup.

Some markup examples:

```
Hello <b>world!</b>`
Hello <span color="blue">world!</span>
```

39.19.5 Implementation

A proof of concept is provided in markup_example.py but it currently only handles the horizontal direction.

Improvements

- This proof of concept uses regex to parse the text but it may be better to use the html.parser from the standard library.
- Computation of text fragment positions could benefit from the OffsetFrom class. See for example item 5 in Using Complex Coordinates with Annotations

Problems

- One serious problem is how to deal with text having both latex and html-like tags. For example, consider the following:

```
$<b>Bold$</b>
```

  Recommendation would be to have mutual exclusion.

39.19.6 Backward compatibility

None at the moment since it is only a proof of concept

39.19.7 Alternatives

As proposed by @anntzer, this could be also implemented as improvements to mathtext. For example:

```
r"$\text{Hello \textbf{world}}$"
r"$\text{Hello \textcolor{blue}{world}}$"
r"$\text{Hello \textsf{\small world}}$"
```

CHAPTER

FORTY

LICENSES

Matplotlib only uses BSD compatible code. If you bring in code from another project make sure it has a PSF, BSD, MIT or compatible license (see the Open Source Initiative licenses page for details on individual licenses). If it doesn't, you may consider contacting the author and asking them to relicense it. GPL and LGPL code are not acceptable in the main code base, though we are considering an alternative way of distributing L/GPL code through an separate channel, possibly a toolkit. If you include code, make sure you include a copy of that code's license in the license directory if the code's license requires you to distribute the license with it. Non-BSD compatible licenses are acceptable in Matplotlib toolkits (e.g., basemap), but make sure you clearly state the licenses you are using.

40.1 Why BSD compatible?

The two dominant license variants in the wild are GPL-style and BSD-style. There are countless other licenses that place specific restrictions on code reuse, but there is an important difference to be considered in the GPL and BSD variants. The best known and perhaps most widely used license is the GPL, which in addition to granting you full rights to the source code including redistribution, carries with it an extra obligation. If you use GPL code in your own code, or link with it, your product must be released under a GPL compatible license. i.e., you are required to give the source code to other people and give them the right to redistribute it as well. Many of the most famous and widely used open source projects are released under the GPL, including linux, gcc, emacs and sage.

The second major class are the BSD-style licenses (which includes MIT and the python PSF license). These basically allow you to do whatever you want with the code: ignore it, include it in your own open source project, include it in your proprietary product, sell it, whatever. python itself is released under a BSD compatible license, in the sense that, quoting from the PSF license page:

```
There is no GPL-like "copyleft" restriction. Distributing
binary-only versions of Python, modified or not, is allowed. There
is no requirement to release any of your source code. You can also
write extension modules for Python and provide them only in binary
form.
```

Famous projects released under a BSD-style license in the permissive sense of the last paragraph are the BSD operating system, python and TeX.

There are several reasons why early Matplotlib developers selected a BSD compatible license. Matplotlib is a python extension, and we choose a license that was based on the python license (BSD compatible). Also, we wanted to attract as many users and developers as possible, and many software companies will not use GPL code in software they plan to distribute, even those that are highly committed to open source development, such as enthought, out of legitimate concern that use of the GPL will "infect" their code base by its viral nature. In effect, they want to retain the right to release some proprietary code. Companies and institutions who use Matplotlib often make significant contributions, because they have the resources to get a job done, even a boring one. Two of the Matplotlib backends (FLTK and WX) were contributed by private companies. The final reason behind the licensing choice is compatibility with the other python extensions for scientific computing: ipython, numpy, scipy, the enthought tool suite and python itself are all distributed under BSD compatible licenses.

CHAPTER

FORTYONE

DEFAULT COLOR CHANGES

As discussed at length elsewhere [insert links], `jet` is an empirically bad color map and should not be the default color map. Due to the position that changing the appearance of the plot breaks backward compatibility, this change has been put off for far longer than it should have been. In addition to changing the default color map we plan to take the chance to change the default color-cycle on plots and to adopt a different color map for filled plots (`imshow`, `pcolor`, `contourf`, etc) and for scatter like plots.

41.1 Default Heat Map Colormap

The choice of a new color map is fertile ground to bike-shedding ("No, it should be _this_ color") so we have a proposed set criteria (via Nathaniel Smith) to evaluate proposed color maps.

- it should be a sequential colormap, because diverging colormaps are really misleading unless you know where the "center" of the data is, and for a default colormap we generally won't.
- it should be perceptually uniform, i.e., human subjective judgments of how far apart nearby colors are should correspond as linearly as possible to the difference between the numerical values they represent, at least locally.
- it should have a perceptually uniform luminance ramp, i.e. if you convert to greyscale it should still be uniform. This is useful both in practical terms (greyscale printers are still a thing!) and because luminance is a very strong and natural cue to magnitude.
- it should also have some kind of variation in hue, because hue variation is a really helpful additional cue to perception, having two cues is better than one, and there's no reason not to do it.
- the hue variation should be chosen to produce reasonable results even for viewers with the more common types of colorblindness. (Which rules out things like red-to-green.)
- For bonus points, it would be nice to choose a hue ramp that still works if you throw away the luminance variation, because then we could use the version with varying luminance for 2d plots, and the version with just hue variation for 3d

plots. (In 3d plots you really want to reserve the luminance channel for lighting/shading, because your brain is *really* good at extracting 3d shape from luminance variation. If the 3d surface itself has massively varying luminance then this screws up the ability to see shape.)

- Not infringe any existing IP

41.1.1 Example script

41.1.2 Proposed Colormaps

41.2 Default Scatter Colormap

For heat-map like applications it can be desirable to cover as much of the luminance scale as possible, however when color mapping markers, having markers too close to white can be a problem. For that reason we propose using a different (but maybe related) color map to the heat map for marker-based. The design parameters are the same as above, only with a more limited luminance variation.

41.2.1 Example script

```
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(1234)

fig, (ax1, ax2) = plt.subplots(1, 2)

N = 50
x = np.random.rand(N)
y = np.random.rand(N)
colors = np.random.rand(N)
area = np.pi * (15 * np.random.rand(N))**2  # 0 to 15 point radiuses

ax1.scatter(x, y, s=area, c=colors, alpha=0.5)


X,Y = np.meshgrid(np.arange(0, 2*np.pi, .2),
                  np.arange(0, 2*np.pi, .2))
U = np.cos(X)
V = np.sin(Y)
Q = ax2.quiver(X, Y, U, V, units='width')
qd = np.random.rand(np.prod(X.shape))
Q.set_array(qd)
```

41.2.2 Proposed Colormaps

41.3 Color Cycle / Qualitative color map

When plotting lines it is frequently desirable to plot multiple lines or artists which need to be distinguishable, but there is no inherent ordering.

41.3.1 Example script

```
import numpy as np
import matplotlib.pyplot as plt

fig, (ax1, ax2) = plt.subplots(1, 2)

x = np.linspace(0, 1, 10)

for j in range(10):
    ax1.plot(x, x * j)


th = np.linspace(0, 2*np.pi, 1024)
for j in np.linspace(0, np.pi, 10):
    ax2.plot(th, np.sin(th + j))

ax2.set_xlim(0, 2*np.pi)
```

41.3.2 Proposed Color cycle

Part VII

Glossary

AGG

The Anti-Grain Geometry (Agg) rendering engine, capable of rendering high-quality images

Cairo

The Cairo graphics engine

FreeType

FreeType is a font rasterization library used by matplotlib which supports TrueType, Type 1, and OpenType fonts.

GTK

The GIMP Toolkit (GTK) graphical user interface library

PyGObject

PyGObject provides Python wrappers for the *GTK* widgets library

pyqt

pyqt provides python wrappers for the *Qt* widgets library and is required by the matplotlib Qt5Agg and Qt4Agg backends. Widely used on linux and windows; many linux distributions package this as 'python-qt5' or 'python-qt4'.

Qt

Qt is a cross-platform application framework for desktop and embedded development.

Qt4

Qt4 is the previous, but most widely used, version of Qt cross-platform application framework for desktop and embedded development.

Qt5

Qt5 is the current version of Qt cross-platform application framework for desktop and embedded development.

Tk

Tk is a graphical user interface for Tcl and many other dynamic languages. It can produce rich, native applications that run unchanged across Windows, Mac OS X, Linux and more.

wxpython

wxpython provides python wrappers for the *wxWidgets* library for use with the WX and WXAgg backends. Widely used on linux, OS-X and windows, it is often packaged by linux distributions as 'python-wxgtk'

wxWidgets

WX is cross-platform GUI and tools library for GTK, MS Windows, and MacOS. It uses native widgets for each operating system, so applications will have the look-and-feel that users on that operating system expect.

BIBLIOGRAPHY

[colorcet] https://colorcet.pyviz.org

[Ware] http://ccom.unh.edu/sites/default/files/publications/Ware_1988_CGA_Color_sequences_univariate_maps.pdf

[Moreland] http://www.kennethmoreland.com/color-maps/ColorMapsExpanded.pdf

[list-colormaps] https://gist.github.com/endolith/2719900#id7

[mycarta-banding] https://mycarta.wordpress.com/2012/10/14/the-rainbow-is-deadlong-live-the-rainbow-part-4-cie-lab-heated-body/

[mycarta-jet] https://mycarta.wordpress.com/2012/10/06/the-rainbow-is-deadlong-live-the-rainbo

[kovesi-colormaps] https://arxiv.org/abs/1509.03700

[bw] http://www.tannerhelland.com/3643/grayscale-image-algorithm-vb6/

[colorblindness] http://www.color-blindness.com/

[IBM] https://doi.org/10.1109/VISUAL.1995.480803

[palettable] https://jiffyclub.github.io/palettable/

[turbo] https://ai.googleblog.com/2019/08/turbo-improved-rainbow-colormap-for.html

[1] Kpathsea documentation The library that `kpsewhich` is part of.

[1] Michel Bernadou, Kamal Hassan, "Basis functions for general Hsieh-Clough-Tocher triangles, complete or reduced.", International Journal for Numerical Methods in Engineering, 17(5):784 - 789. 2.01.

[2] C.T. Kelley, "Iterative Methods for Optimization".

PYTHON MODULE INDEX

m

p

INDEX

Non-alphabetical

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

T

U

V

W

X

Y

Z