## User’s Guide

### 1 Introduction

### 2 Installing

2.1 Manually installing pre-built packages

2.2 Installing from source

2.3 Build requirements

2.4 Building on OSX

### 3 Pyplot tutorial

3.1 Controlling line properties

3.2 Working with multiple figures and axes

3.3 Working with text

### 4 Interactive navigation

4.1 Navigation Keyboard Shortcuts

### 5 Customizing matplotlib

5.1 The matplotlibrc file

5.2 Dynamic rc settings

### 6 Using matplotlib in a python shell

6.1 Ipython to the rescue

6.2 Other python interpreters

6.3 Controlling interactive updating

### 7 Working with text

7.1 Text introduction

7.2 Basic text commands

7.3 Text properties and layout

7.4 Writing mathematical expressions

7.5 Typesetting With XeLaTeX/LuaLaTeX

7.6 Text rendering With LaTeX

7.7 Annotating text

### 8 Image tutorial

8.1 Startup commands
# Contents

16.3 Placing Artist at the anchored location of the Axes ........................................ 162
16.4 Using Complex Coordinate with Annotation ...................................................... 164
16.5 Using ConnectorPatch ......................................................................................... 167
16.6 Zoom effect between Axes ................................................................................... 168
16.7 Define Custom BoxStyle ...................................................................................... 168
17 Our Favorite Recipes ............................................................................................... 173
  17.1 Sharing axis limits and views .............................................................................. 173
  17.2 Easily creating subplots ..................................................................................... 173
  17.3 Fixing common date annoyances ....................................................................... 174
  17.4 Fill Between and Alpha ..................................................................................... 176
  17.5 Transparent, fancy legends ............................................................................... 180
  17.6 Placing text boxes ............................................................................................. 181
18 Screenshots ............................................................................................................ 183
  18.1 Simple Plot ......................................................................................................... 183
  18.2 Subplot demo ..................................................................................................... 184
  18.3 Histograms ........................................................................................................ 184
  18.4 Path demo .......................................................................................................... 185
  18.5 mplot3d .............................................................................................................. 186
  18.6 Streamplot ......................................................................................................... 187
  18.7 Ellipses .............................................................................................................. 189
  18.8 Bar charts .......................................................................................................... 190
  18.9 Pie charts ........................................................................................................... 191
  18.10 Table demo ...................................................................................................... 192
  18.11 Scatter demo .................................................................................................... 193
  18.12 Slider demo ...................................................................................................... 194
  18.13 Fill demo .......................................................................................................... 195
  18.14 Date demo ........................................................................................................ 196
  18.15 Financial charts ............................................................................................... 197
  18.16 Basemap demo ............................................................................................... 198
  18.17 Log plots .......................................................................................................... 199
  18.18 Polar plots ........................................................................................................ 200
  18.19 Legends ............................................................................................................ 201
  18.20 Mathtext_examples ....................................................................................... 202
  18.21 Native TeX rendering ...................................................................................... 203
  18.22 EEG demo ....................................................................................................... 204
  18.23 XKCD-style sketch plots .................................................................................. 205
19 What’s new in matplotlib ......................................................................................... 209
  19.1 new in matplotlib-1.3 ....................................................................................... 211
  19.2 new in matplotlib 1.2.2 .................................................................................... 221
  19.3 new in matplotlib-1.2 ....................................................................................... 221
  19.4 new in matplotlib-1.1 ....................................................................................... 232
  19.5 new in matplotlib-1.0 ....................................................................................... 238
  19.6 new in matplotlib-0.99 ..................................................................................... 243
  19.7 new in 0.98.4 .................................................................................................... 246
20 Github stats 255

21 License 289
  21.1 Copyright Policy ............................................................... 289
  21.2 License agreement for matplotlib 1.3.0 ........................................ 289
  21.3 License agreement for matplotlib versions prior to 1.3.0 ...................... 290

22 Credits 293

II The Matplotlib FAQ 297

23 Installation 299
  23.1 Report a compilation problem .................................................. 299
  23.2 matplotlib compiled fine, but nothing shows up when I use it .............. 299
  23.3 How to completely remove matplotlib ......................................... 300
  23.4 How to Install ............................................................................ 301
  23.5 Linux Notes ............................................................................... 302
  23.6 OS-X Notes ............................................................................... 302
  23.7 Windows Notes ......................................................................... 304

24 Usage 305
  24.1 General Concepts ....................................................................... 305
  24.2 Matplotlib, pylab, and pyplot: how are they related? ......................... 306
  24.3 Coding Styles ............................................................................ 306
  24.4 What is a backend? ..................................................................... 307
  24.5 What is interactive mode? ............................................................ 309

25 How-To 313
  25.1 Plotting: howto .......................................................................... 314
  25.2 Contributing: howto .................................................................... 324
  25.3 Matplotlib in a web application server .......................................... 325
  25.4 Search examples ......................................................................... 326
  25.5 Cite Matplotlib .......................................................................... 326

26 Troubleshooting 329
  26.1 Obtaining matplotlib version ...................................................... 329
  26.2 matplotlib install location ............................................................ 329
  26.3 .matplotlib directory location ..................................................... 329
  26.4 Getting help .............................................................................. 330
  26.5 Problems with recent git versions ................................................... 331

27 Environment Variables 333
  27.1 Setting environment variables in Linux and OS-X ............................. 333
  27.2 Setting environment variables in windows ....................................... 334
34.1 matplotlib.transforms .............................................. 381

35 Adding new scales and projections to matplotlib 403
35.1 Creating a new scale .................................................. 403
35.2 Creating a new projection .......................................... 404
35.3 API documentation .................................................. 405

IV Matplotlib AxesGrid Toolkit 415

36 Overview of AxesGrid toolkit 419
36.1 What is AxesGrid toolkit? ........................................... 419
36.2 AXES_GRID1 ............................................................ 421
36.3 AXISARTIST ............................................................ 435

37 The Matplotlib AxesGrid Toolkit User’s Guide 441
37.1 AxesDivider .............................................................. 441
37.2 AXISARTIST namespace ......................................... 444

38 The Matplotlib AxesGrid Toolkit API 457
38.1 mpl_toolkits.axes_grid.axes_size ................................. 457
38.2 mpl_toolkits.axes_grid.axes_divider ......................... 458
38.3 mpl_toolkits.axes_grid.axes_grid ......................... 462
38.4 mpl_toolkits.axes_grid.axis_artist ................... 463

V mplot3d 467

39 Matplotlib mplot3d toolkit 469
39.1 mplot3d tutorial ...................................................... 469
39.2 mplot3d API .......................................................... 491
39.3 mplot3d FAQ .......................................................... 518

VI Toolkits 521

40 Basemap (Not distributed with matplotlib) 525
41 Cartopy (Not distributed with matplotlib) 527
42 GTK Tools 529
43 Excel Tools 531
44 Natgrid (Not distributed with matplotlib) 533
45 mplot3d 535
46 AxesGrid 537
<table>
<thead>
<tr>
<th>Section</th>
<th>Subsection</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>52</td>
<td>artists</td>
<td>599</td>
</tr>
<tr>
<td>52.1</td>
<td>matplotlib.artist</td>
<td>599</td>
</tr>
<tr>
<td>52.2</td>
<td>matplotlib.lines</td>
<td>608</td>
</tr>
<tr>
<td>52.3</td>
<td>matplotlib.patches</td>
<td>616</td>
</tr>
<tr>
<td>52.4</td>
<td>matplotlib.text</td>
<td>653</td>
</tr>
<tr>
<td>53</td>
<td>axes</td>
<td>665</td>
</tr>
<tr>
<td>53.1</td>
<td>matplotlib.axes</td>
<td>665</td>
</tr>
<tr>
<td>54</td>
<td>axis</td>
<td>831</td>
</tr>
<tr>
<td>54.1</td>
<td>matplotlib.axis</td>
<td>831</td>
</tr>
<tr>
<td>55</td>
<td>backends</td>
<td>841</td>
</tr>
<tr>
<td>55.1</td>
<td>matplotlib.backend_bases</td>
<td>841</td>
</tr>
<tr>
<td>55.2</td>
<td>matplotlib.backends.backend_gtkagg</td>
<td>860</td>
</tr>
<tr>
<td>55.3</td>
<td>matplotlib.backends.backend_qt4agg</td>
<td>860</td>
</tr>
<tr>
<td>55.4</td>
<td>matplotlib.backends.backend_wxagg</td>
<td>861</td>
</tr>
<tr>
<td>55.5</td>
<td>matplotlib.backends.backend_pdf</td>
<td>862</td>
</tr>
<tr>
<td>55.6</td>
<td>matplotlib.dviread</td>
<td>864</td>
</tr>
<tr>
<td>55.7</td>
<td>matplotlib.typelfont</td>
<td>867</td>
</tr>
<tr>
<td>56</td>
<td>cbook</td>
<td>869</td>
</tr>
<tr>
<td>56.1</td>
<td>matplotlib.cbook</td>
<td>869</td>
</tr>
<tr>
<td>57</td>
<td>cm (colormap)</td>
<td>881</td>
</tr>
<tr>
<td>57.1</td>
<td>matplotlib.cm</td>
<td>881</td>
</tr>
<tr>
<td>58</td>
<td>collections</td>
<td>885</td>
</tr>
<tr>
<td>58.1</td>
<td>matplotlib.collections</td>
<td>885</td>
</tr>
<tr>
<td>59</td>
<td>colorbar</td>
<td>901</td>
</tr>
<tr>
<td>59.1</td>
<td>matplotlib.colorbar</td>
<td>901</td>
</tr>
<tr>
<td>60</td>
<td>colors</td>
<td>905</td>
</tr>
<tr>
<td>60.1</td>
<td>matplotlib.colors</td>
<td>905</td>
</tr>
<tr>
<td>61</td>
<td>dates</td>
<td>915</td>
</tr>
<tr>
<td>61.1</td>
<td>matplotlib.dates</td>
<td>915</td>
</tr>
<tr>
<td>62</td>
<td>figure</td>
<td>925</td>
</tr>
<tr>
<td>62.1</td>
<td>matplotlib.figure</td>
<td>925</td>
</tr>
<tr>
<td>63</td>
<td>font_manager</td>
<td>947</td>
</tr>
<tr>
<td>63.1</td>
<td>matplotlib.font_manager</td>
<td>947</td>
</tr>
<tr>
<td>63.2</td>
<td>matplotlib.fontconfig_pattern</td>
<td>952</td>
</tr>
<tr>
<td>64</td>
<td>gridspec</td>
<td>955</td>
</tr>
<tr>
<td>64.1</td>
<td>matplotlib.gridspec</td>
<td>955</td>
</tr>
<tr>
<td>65</td>
<td>legend</td>
<td>959</td>
</tr>
<tr>
<td>65.1 matplotlib.legend</td>
<td>959</td>
<td></td>
</tr>
<tr>
<td>------------------------</td>
<td>----</td>
<td></td>
</tr>
<tr>
<td>66 Markers</td>
<td>963</td>
<td></td>
</tr>
<tr>
<td>66.1 matplotlib.markers</td>
<td>963</td>
<td></td>
</tr>
<tr>
<td>67 mathtext</td>
<td>967</td>
<td></td>
</tr>
<tr>
<td>67.1 matplotlib.mathtext</td>
<td>968</td>
<td></td>
</tr>
<tr>
<td>68 mlab</td>
<td>983</td>
<td></td>
</tr>
<tr>
<td>68.1 matplotlib.mlab</td>
<td>983</td>
<td></td>
</tr>
<tr>
<td>69 path</td>
<td>1005</td>
<td></td>
</tr>
<tr>
<td>69.1 matplotlib.path</td>
<td>1005</td>
<td></td>
</tr>
<tr>
<td>70 pyplot</td>
<td>1013</td>
<td></td>
</tr>
<tr>
<td>70.1 matplotlib.pyplot</td>
<td>1013</td>
<td></td>
</tr>
<tr>
<td>71 sankey</td>
<td>1189</td>
<td></td>
</tr>
<tr>
<td>71.1 matplotlib.sankey</td>
<td>1189</td>
<td></td>
</tr>
<tr>
<td>72 spines</td>
<td>1197</td>
<td></td>
</tr>
<tr>
<td>72.1 matplotlib.spines</td>
<td>1197</td>
<td></td>
</tr>
<tr>
<td>73 ticker</td>
<td>1201</td>
<td></td>
</tr>
<tr>
<td>73.1 matplotlib.ticker</td>
<td>1201</td>
<td></td>
</tr>
<tr>
<td>74 tight_layout</td>
<td>1211</td>
<td></td>
</tr>
<tr>
<td>74.1 matplotlib.tight_layout</td>
<td>1211</td>
<td></td>
</tr>
<tr>
<td>75 triangular grids</td>
<td>1213</td>
<td></td>
</tr>
<tr>
<td>75.1 matplotlib.tri</td>
<td>1213</td>
<td></td>
</tr>
<tr>
<td>76 units</td>
<td>1225</td>
<td></td>
</tr>
<tr>
<td>76.1 matplotlib.units</td>
<td>1225</td>
<td></td>
</tr>
<tr>
<td>77 widgets</td>
<td>1227</td>
<td></td>
</tr>
<tr>
<td>77.1 matplotlib.widgets</td>
<td>1227</td>
<td></td>
</tr>
</tbody>
</table>

VIII Glossary 1237

Bibliography 1241

Python Module Index 1243

Python Module Index 1245

Index 1247
Part I

User’s Guide
matplotlib is a library for making 2D plots of arrays in Python. Although it has its origins in emulating the MATLAB® 1 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 and 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 experience 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 `matplotlib.pylab` which allow the user to create plots with code quite similar to MATLAB

---

1 MATLAB is a registered trademark of The MathWorks, Inc.
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 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.
There are many different ways to install matplotlib, and the best way depends on what operating system you are using, what you already have installed, and how you want to use it. To avoid wading through all the details (and potential complications) on this page, the easiest thing for you to do is use one of the prepackaged python distributions that already provide matplotlib built-in. The Enthought Python Distribution (EPD) for Windows, OS X or Redhat is an excellent choice that “just works” out of the box. Another excellent alternative for Windows users is Python (x, y) which tends to be updated a bit more frequently. Both of these packages include matplotlib and pylab, and lots of other useful tools. matplotlib is also packaged for almost every major Linux distribution. So if you are on Linux, your package manager will probably provide matplotlib prebuilt.

2.1 Manually installing pre-built packages

2.1.1 General instructions

For some people, the prepackaged pythons discussed above are not an option. That’s OK, it’s usually pretty easy to get a custom install working. You will first need to find out if you have python installed on your machine, and if not, install it. The official python builds are available for download here, but OS X users please read Which python for OS X?.

Once you have python up and running, you will need to install numpy. numpy provides high-performance array data structures and mathematical functions, and is a requirement for matplotlib. You can test your progress:

```python
>>> import numpy
>>> print numpy.__version__
```

matplotlib requires numpy version 1.5 or later. Although it is not a requirement to use matplotlib, we strongly encourage you to install ipython, which is an interactive shell for python that is matplotlib-aware.

Next, we need to get matplotlib installed. We provide prebuilt binaries for OS X and Windows on the matplotlib download page. Click on the latest release of the “matplotlib” package, choose your python version (2.6, 2.7 or 3.2) and your platform (macosx or win32). If you have any problems, please check the Installation, search using Google, and/or post a question the mailing list.

If you are on Debian/Ubuntu linux, it suffices to do:
> sudo apt-get install python-matplotlib

Instructions for installing our OSX binaries are found in the FAQ Installing OSX binaries.

Once you have ipython, numpy and matplotlib installed, you can use ipython’s “pylab” mode to have a MATLAB-like environment that automatically handles most of the configuration details for you, so you can get up and running quickly:

johnh@flag:~> ipython -pylab
Python 2.4.5 (#4, Apr 12 2008, 09:09:16)
IPython 0.9.0 -- An enhanced Interactive Python.

Welcome to pylab, a matplotlib-based Python environment.
For more information, type 'help(pylab)'.

In [1]: x = randn(10000)

In [2]: hist(x, 100)

Note that when testing matplotlib installations from the interactive python console, there are some issues relating to user interface toolkits and interactive settings that are discussed in Using matplotlib in a python shell.

2.1.2 Installing on Windows

If you don’t already have python installed, you may want to consider using the Enthought edition of python, which has scipy, numpy, and wxpython, plus many other useful packages, preinstalled - Enthought Python. With the Enthought edition of python + matplotlib installer, the following backends should work out of the box: agg, wx, wxagg, tkagg, ps, pdf and svg.

For standard python installations, you will also need to install numpy in addition to the matplotlib installer. On some systems you will also need to download msvcp71.dll library, which you can download from http://www.dll-files.com/dllindex/dll-files.shtml?msvcp71 or other sites. You will need to unzip the archive and drag the dll into c:\windows\system32.

All of the GUI backends run on Windows, but TkAgg is probably the best for interactive use from the standard python shell or ipython. The Windows installer (*.exe) on the download page contains all the code you need to get up and running. However, there are many examples that are not included in the Windows installer. If you want to try the many demos that come in the matplotlib source distribution, download the zip file and look in the examples subdirectory.

2.2 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 download page, or if you want to develop matplotlib or just need the latest bugfixed version, grab the latest git version Source install from git.
Once you have satisfied the requirements detailed below (mainly python, numpy, libpng and freetype), you can build matplotlib:

```
cd matplotlib
python setup.py build
python setup.py install
```

We provide a `setup.cfg` file that goes with `setup.py` 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.

If you have installed prerequisites to nonstandard places and need to inform matplotlib where they are, edit `setupext.py` and add the base dirs to the `basedir` dictionary entry for your `sys.platform`, e.g., if the header to some required library is in `/some/path/include/someheader.h`, put `/some/path` in the `basedir` list for your platform.

## 2.3 Build requirements

These are external packages which you will need to install before installing matplotlib. Windows users only need the first two (python and numpy) since the others are built into the matplotlib Windows installers available for download at the download page <https://github.com/matplotlib/matplotlib/downloads>_ If you are building on OSX, see [Building on OSX]. If you are installing dependencies with a package manager on Linux, you may need to install the development packages (look for a “-dev” postfix) in addition to the libraries themselves.

**Note:** If you are on debian/ubuntu, you can get all the dependencies required to build matplotlib with:

```
sudo apt-get build-dep python-matplotlib
```

If you are on Fedora/RedHat, you can get all the dependencies required to build matplotlib by first installing `yum-builddep` and then running:

```
su -c "yum-builddep python-matplotlib"
```

This does not build matplotlib, but it does get the install the build dependencies, which will make building from source easier.

**python 2.6, 2.7, 3.1 or 3.2** Download python.

**numpy 1.5 (or later)** array support for python (download numpy)

**libpng 1.2 (or later)** library for loading and saving PNG files (download). libpng requires zlib. If you are a Windows user, you can ignore this because we build support into the matplotlib single-click installer

**freetype 1.4 (or later)** library for reading true type font files. If you are a windows user, you can ignore this since we build support into the matplotlib single click installer.

**dateutil 1.1 or later** Provides extensions to python datetime handling. If using pip, easy_install or installing from source, the installer will attempt to download and install python_dateutil from PyPI.
**pyparsing** Required for matplotlib’s mathtext math rendering support. If using pip, easy_install or installing from source, the installer will attempt to download and install pyparsing from PyPI.

**Optional**

These are optional packages which you may want to install to use matplotlib with a user interface toolkit. See *What is a backend?* for more details on the optional matplotlib backends and the capabilities they provide.

- **tk 8.3 or later** The TCL/Tk widgets library used by the TkAgg backend
- **pyqt 4.0 or later** The Qt4 widgets library python wrappers for the Qt4Agg backend
- **pygtk 2.4 or later** The python wrappers for the GTK widgets library for use with the GTK or GTKAgg backend
- **wxpython 2.8 or later** The python wrappers for the wx widgets library for use with the WX or WXAgg backend

**Required libraries that ship with matplotlib**

- **agg 2.4** The antigrain C++ rendering engine. matplotlib links against the agg template source statically, so it will not affect anything on your system outside of matplotlib.
- **PyCXX 6.2.4** A library for writing Python extensions in C++.

### 2.4 Building on OSX

The build situation on OSX is complicated by the various places one can get the libpng and freetype requirements (darwinports, fink, /usr/X11R6) and the different architectures (e.g., x86, ppc, universal) and the different OSX version (e.g., 10.4 and 10.5). We recommend that you build the way we do for the OSX release: get the source from the tarball or the git repository and follow the instruction in README.osx.
**matplotlib.pyplot** is a collection of command style functions that make matplotlib work like MATLAB. Each `pyplot` function makes some change to a figure: eg, create a figure, create a plotting area in a figure, plot some lines in a plotting area, decorate the plot with labels, etc.... *matplotlib.pyplot* is stateful, in that it keeps track of the current figure and plotting area, and the plotting functions are directed to the current axes.

```python
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 the `plot()` command, 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 command, and will take an arbitrary number of arguments. For example, to plot x versus y, you can issue the command:

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

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

```python
import matplotlib.pyplot as plt
plt.plot([1,2,3,4], [1,4,9,16], 'ro')
plt.axis([0, 6, 0, 20])
```

See the `plot()` documentation for a complete list of line styles and format strings. The `axis()` command 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 a plotting several lines with different format styles in one command using arrays.

```python
import matplotlib.pyplot as plt
plt.plot([1,2,3,4], [1,4,9,16], 'ro')
plt.axis([0, 6, 0, 20])
```
import numpy as np
import matplotlib.pyplot as plt

# 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^')

3.1 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 the Line2D instance. plot returns a list of lines; eg `line1, line2 = plot(x1,y1,x2,y2)`. Below I have only one line so it is a list of length 1. I use tuple unpacking in the line, `= plot(x, y, 'o')` to get the first element of the list:
Matplotlib, Release 1.3.0

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

- Use the `setp()` command. The example below uses a MATLAB-style command 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:

```python
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.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>alpha</td>
<td>float</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transform.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>a Path instance and a Transform instance, a Patch</td>
</tr>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>the hit testing function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>data</td>
<td>(np.array xdata, np.array ydata)</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>label</td>
<td>any string</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['.-'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>marker</td>
<td>['+'</td>
</tr>
<tr>
<td>markeredgecolor or mec</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markeredgewidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>None</td>
</tr>
<tr>
<td>picker</td>
<td>used in interactive line selection</td>
</tr>
<tr>
<td>pickradius</td>
<td>the line pick selection radius</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>np.array</td>
</tr>
<tr>
<td>ydata</td>
<td>np.array</td>
</tr>
</tbody>
</table>

Continued on next page
Table 3.1 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Value Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

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]
   ...

3.2 Working with multiple figures and axes

MATLAB, and `pyplot`, have the concept of the current figure and the current axes. All plotting commands apply to the current axes. The function `gca()` returns the current axes (a `matplotlib.axes.Axes` instance), and `gcf()` returns the current figure (`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.

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

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(1)
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--')
```
The `figure()` command 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 an axes. The `subplot()` command specifies `numrows`, `numcols`, `fignum` where `fignum` ranges from 1 to `numrows*numcols`. The commas in the subplot command 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, ie, not on a rectangular grid, use the `axes()` command, which allows you to specify the location as `axes([left, bottom, width, height])` where all values are in fractional (0 to 1) coordinates. See `pylab_examples-axes_demo` for an example of placing axes manually and `pylab_examples-line_styles` for an example with lots-o-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:

```python
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
```
Matplotlib, Release 1.3.0

plt.subplot(211)
plt.title(’Easy as 1,2,3’)

# make subplot(211) in figure1 current
# subplot 211 title

You can clear the current figure with clf() and the current axes with cla(). If you find this statefulness,
annoying, 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 a long sequence 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.

3.3 Working with text
The text() command can be used to add text in an arbitrary location, and the xlabel(), ylabel() and
title() are used to add text in the indicated locations (see Text introduction for a more detailed example)
import numpy as np
import matplotlib.pyplot as plt
mu, sigma = 100, 15
x = mu + sigma * np.random.randn(10000)
# the histogram of the data
n, bins, patches = plt.hist(x, 50, normed=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)

3.3. Working with text

15


All of the `text()` commands return a `matplotlib.text.Text` instance. Just as with lines above, you can customize the properties by passing keyword arguments into the text functions or using `setp()`:

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

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

### 3.3.1 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:

```python
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 and 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*. 
3.3.2 Annotating text

The uses of the basic `text()` command above 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.

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

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 *Annotating text* and *Annotating Axes* for details. More examples can be found in *pylab_examples-annotation_demo*.
All figure windows come with a navigation toolbar, which can be used to navigate through the data set. Here is a description of each of the buttons at the bottom of the toolbar.

The **Forward** and **Back** buttons  These are akin to the web browser forward and back buttons. They are used to navigate back and forth between previously defined views. They have no meaning unless you have already navigated somewhere else using the pan and zoom buttons. This is analogous to trying to click **Back** on your web browser before visiting a new page –nothing happens. **Home** always takes you to the first, default view of your data. For **Home**, **Forward** and **Back**, think web browser where data views are web pages. Use the pan and zoom to rectangle to define new views.

The **Pan/Zoom button**  This button has two modes: pan and zoom. Click the toolbar button to activate panning and zooming, then put your mouse somewhere over an axes. Press the left mouse button and hold it to pan the figure, dragging it to a new position. When you release it, the data under the point where you pressed will be moved to the point where you released. If you press ‘x’ or ‘y’ while panning the motion will be constrained to the x or y axis, respectively. Press the right mouse button to zoom, dragging it to a new position. The x axis will be zoomed in proportionate to the rightward movement and zoomed out proportionate to the leftward movement. Ditto for the y axis and up/down motions. The point under your mouse when you begin the zoom remains stationary, allowing you to zoom to an arbitrary point in the figure. You can use the modifier keys ‘x’, ‘y’ or ‘CONTROL’ to constrain the zoom to the x axis, the y axis, or aspect ratio preserve, respectively.

With polar plots, the pan and zoom functionality behaves differently. The radius axis labels can be dragged using the left mouse button. The radius scale can be zoomed in and out using the right mouse button.
The **Zoom-to-rectangle button** Click this toolbar button to activate this mode. Put your mouse somewhere over and axes and press the left mouse button. Drag the mouse while holding the button to a new location and release. The axes view limits will be zoomed to the rectangle you have defined. There is also an experimental ‘zoom out to rectangle’ in this mode with the right button, which will place your entire axes in the region defined by the zoom out rectangle.

The **Subplot-configuration button** Use this tool to configure the parameters of the subplot: the left, right, top, bottom, space between the rows and space between the columns.

The **Save button** Click this button to launch a file save dialog. You can save files with the following extensions: png, ps, eps, svg and pdf.

4.1 Navigation Keyboard Shortcuts

The following table holds all the default keys, which can be overwritten by use of your matplotlibrc (#keymap.*).

<table>
<thead>
<tr>
<th>Command</th>
<th>Keyboard Shortcut(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Home/Reset</td>
<td>h or r or home</td>
</tr>
<tr>
<td>Back</td>
<td>e or left arrow or backspace</td>
</tr>
<tr>
<td>Forward</td>
<td>v or right arrow</td>
</tr>
<tr>
<td>Pan/Zoom</td>
<td>p</td>
</tr>
<tr>
<td>Zoom-to-rect</td>
<td>o</td>
</tr>
<tr>
<td>Save</td>
<td>ctrl + s</td>
</tr>
<tr>
<td>Toggle fullscreen</td>
<td>ctrl + f</td>
</tr>
<tr>
<td>Close plot</td>
<td>ctrl + w</td>
</tr>
<tr>
<td>Constrain pan/zoom to x axis</td>
<td>hold x when panning/zooming with mouse</td>
</tr>
<tr>
<td>Constrain pan/zoom to y axis</td>
<td>hold y when panning/zooming with mouse</td>
</tr>
<tr>
<td>Preserve aspect ratio</td>
<td>hold CONTROL when panning/zooming with mouse</td>
</tr>
<tr>
<td>Toggle grid</td>
<td>g when mouse is over an axes</td>
</tr>
<tr>
<td>Toggle x axis scale (log/linear)</td>
<td>L or k when mouse is over an axes</td>
</tr>
<tr>
<td>Toggle y axis scale (log/linear)</td>
<td>l when mouse is over an axes</td>
</tr>
</tbody>
</table>

If you are using `matplotlib.pyplot` the toolbar will be created automatically for every figure. If you are writing your own user interface code, you can add the toolbar as a widget. The exact syntax depends on
your UI, but we have examples for every supported UI in the matplotlib/examples/user_interfaces directory. Here is some example code for GTK:

```python
from matplotlib.figure import Figure
from matplotlib.backends.backend_gtkagg import FigureCanvasGTKAgg as FigureCanvas
from matplotlib.backends.backend_gtkagg import NavigationToolbar2GTKAgg as NavigationToolbar

win = gtk.Window()
win.connect("destroy", lambda x: gtk.main_quit())
win.set_default_size(400,300)
win.set_title("Embedding in GTK")

vbox = gtk.VBox()
win.add(vbox)

fig = Figure(figsize=(5,4), dpi=100)
ax = fig.add_subplot(111)
ax.plot([1,2,3])

canvas = FigureCanvas(fig)  # a gtk.DrawingArea
vbox.pack_start(canvas)
toolbar = NavigationToolbar(canvas, win)
vbox.pack_start(toolbar, False, False)

win.show_all()
gtk.main()
```

4.1. Navigation Keyboard Shortcuts
CHAPTER
FIVE

CUSTOMIZING MATPLOTLIB

5.1 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. matplotlib looks for matplotlibrc in three 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. It next looks in a user-specific place, depending on your platform:
   - On Linux, 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 directory location.

3. INSTALL/matplotlib/mpl-data/matplotlibrc, where INSTALL is something like /usr/lib/python2.5/site-packages on Linux, and maybe C:\Python25\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.

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

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

See below for a sample matplotlibrc file.

5.2 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:
Matplotlib also provides a couple of convenience functions for modifying rc settings. The `matplotlib.rc()` command can be used to modify multiple settings in a single group at once, using keyword arguments:

```python
import matplotlib as mpl
mpl.rc('lines', linewidth=2, color='r')
```

There `matplotlib.rcdefaults()` command will restore the standard matplotlib default settings.

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

### 5.2.1 A sample matplotlibrc file

#### MATPLOTLIBRC FORMAT

```python
# This file is best viewed in an editor which supports python mode syntax highlighting. 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
#
# 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 or #ff00ff - a scalar grayscale intensity such as 0.75 - a legal html color name, eg red, blue, darkslategray

### CONFIGURATION BEGINS HERE

# the default backend; one of GTK GTKAgg GTKCairo GTK3Agg GTK3Cairo CocoaAgg MacOSX Qt4Agg TkAgg WX WXAgg Agg Cairo GDK 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 : qt4agg

# If you are using the Qt4Agg backend, you can choose here
```
# to use the PyQt4 bindings or the newer PySide bindings to
# the underlying Qt4 toolkit.
#backend.qt4 : PyQt4 # PyQt4 | PySide

# Note that this can be overridden by the environment variable
# QT_API used by Enthought Tool Suite (ETS); valid values are
# "pyqt" and "pyside". The "pyqt" setting has the side effect of
# forcing the use of Version 2 API for QString and QVariant.

# The port to use for the web server in the WebAgg backend.
# webagg.port : 8888

# 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 ("classic" is deprecated)
#timezone : UTC # a pytz timezone string, eg US/Central or Europe/Paris

# Where your matplotlib data lives if you installed to a non-default
# location. This is where the matplotlib fonts, bitmaps, etc reside
#datapath : /home/jdhunter/mpldata

### LINES
# See http://matplotlib.org/api/artist_api.html#module-matplotlib.lines for more
# information on line properties.
#lines.linewidth : 1.0 # line width in points
#lines.linestyle : - # solid line
#lines.color : blue # has no affect on plot(); see axes.color_cycle
#lines.marker : None # the default marker
#lines.markeredgewidth : 0.5 # the line width around the marker symbol
#lines.markersize : 6 # markersize, in points
#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
#lines.antialiased : True # render lines in antialiased (no jaggies)

### PATCHES
# Patches are graphical objects that fill 2D space, like polygons or
# circles. See
# http://matplotlib.org/api/artist_api.html#module-matplotlib.patches
# information on patch properties
#patch.linewidth : 1.0  # edge width in points
#patch.facecolor : blue
#patch.edgecolor : black
#patch.antialiased : True  # render patches in antialiased (no jaggies)

### FONT

# font properties used by text.Text. See
# http://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
# 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.
# 12pt is the standard value.
#
# font.family : sans-serif
# font.style : normal
# font.variant : normal
# font.weight : medium
# font.stretch : normal
# 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.size : 12.0
# serif : Bitstream Vera Serif, New Century Schoolbook, Century Schoolbook L, Utopia, ITC Palatino Linotype, serif
Matplotlib, Release 1.3.0

# font.sans-serif : Bitstream Vera Sans, Lucida Grande, Verdana, Geneva, Lucid, Arial, Helvetica, Avant Garde
# font.cursive : Apple Chancery, Textile, Zapf Chancery, Sand, cursive
# font.fantasy : Comic Sans MS, Chicago, Charcoal, Impact, Western, fantasy
# font.monospace : Bitstream Vera Sans Mono, Andale Mono, Nimbus Mono L, Courier New, Courier, Fixed

### TEXT
# text properties used by text.Text. See
# http://matplotlib.org/api/artist_api.html#module-matplotlib.text for more
# information on text properties

#text.color : black

### LaTeX customizations. See http://www.scipy.org/Wiki/Cookbook/Matplotlib/UsingTex
#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.unicode : False # use "ucs" and "inputenc" LaTeX packages for handling
# unicode strings.

#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.
# preamble is a comma separated list of LaTeX statements
# that are included in the LaTeX document preamble.
# An example:
# text.latex.preamble : \usepackage{bm},\usepackage{euler}
# 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

#text.dvipnghack : None # some versions of dvipng don't handle alpha
# channel properly. Use True to correct
# and flush ~/.matplotlib/tex.cache
# before testing and False to force
# correction off. None will try and
# guess based on your dvipng version

#text.hinting : 'auto' # May be one of the following:
# 'none': Perform no hinting
# 'auto': Use freetype's autohinter
# 'native': Use the hinting information in the
# font file, if available, and if your
# freetype library supports it
# 'either': Use the native hinting information,
# or the autohinter if none is available.
# For backward compatibility, this value may also be
# True == 'auto' or False == 'none'.

5.2. Dynamic rc settings 27
#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.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.
# They map from a TeX font name to a fontconfig font pattern.
# These settings are only used if mathtext.fontset is 'custom'.
# Note that this "custom" mode is unsupported and may go away in the future.
#mathtext.cal : cursive
#mathtext.rm : serif
#mathtext.tt : monospace
#mathtext.it : serif:italic
#mathtext.bf : serif:bold
#mathtext.sf : sans
mathtext.fontset : cm # Should be 'cm' (Computer Modern), 'stix',
   # 'stixsans' or 'custom'
mathtext.fallback_to_cm : True # When True, use symbols from the Computer Modern fonts when a symbol can not be found in one of
   # the custom math fonts.

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
# default face and edge color, default tick sizes,
# default fontsize for ticklabels, and so on. See
# http://matplotlib.org/api/axes_api.html#module-matplotlib.axes
#axes.hold : True # whether to clear the axes by default on
#axes.facecolor : white # axes background color
#axes.edgecolor : black # axes edge color
#axes.linewidth : 1.0 # edge linewidth
#axes.grid : False # display grid or not
#axes.titlesize : large # fontsize of the axes title
#axes.labelsize : medium # fontsize of the x and y labels
#axes.labelweight : normal # weight of the x and y labels
#axes.labelcolor : black
#axes.axisbelow : False # whether axis gridlines and ticks are below
   # the axes elements (lines, text, etc)
#axes.formatter.limits : -7, 7 # 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.unicode_minus : True # use unicode for the minus symbol
# rather than hyphen. See
# http://en.wikipedia.org/wiki/Plus_and_minus_signs#Character_codes

#axes.color_cycle : b, g, r, c, m, y, k # color cycle for plot lines
# as list of string colorspecs:
# single letter, long name, or
# web-style hex

#axes.xaxis : 0 # x margin. See 'axes.Axes.margins'
#axes.yaxis : 0 # y margin See 'axes.Axes.margins'

#polaraxes.grid : True # display grid on polar axes
#axes3d.grid : True # display grid on 3d axes

### TICKS
# see http://matplotlib.org/api/axis_api.html#matplotlib.axis.Tick
#xtick.major.size : 4 # major tick size in points
#xtick.minor.size : 2 # minor tick size in points
#xtick.major.width : 0.5 # major tick width in points
#xtick.minor.width : 0.5 # minor tick width in points
#xtick.major.pad : 4 # distance to major tick label in points
#xtick.minor.pad : 4 # distance to the minor tick label in points
#xtick.color : k # color of the tick labels
#xtick.labelsize : medium # fontsize of the tick labels
#xtick.direction : in # direction: in, out, or inout

#ytick.major.size : 4 # major tick size in points
#ytick.minor.size : 2 # minor tick size in points
#ytick.major.width : 0.5 # major tick width in points
#ytick.minor.width : 0.5 # minor tick width in points
#ytick.major.pad : 4 # distance to major tick label in points
#ytick.minor.pad : 4 # distance to the minor tick label in points
#ytick.color : k # color of the tick labels
#ytick.labelsize : medium # fontsize of the tick labels
#ytick.direction : in # direction: in, out, or inout

### GRIDS
#grid.color : black # grid color
#grid.linestyle : : # dotted
#grid.linewidth : 0.5 # in points
#grid.alpha : 1.0 # transparency, between 0.0 and 1.0

### Legend
#legend.fancybox : False # if True, use a rounded box for the
# legend, else a rectangle
#legend.isaxes : True
#legend.numpoints : 2 # the number of points in the legend line
#legend.fontsize : large
#legend.borderpad : 0.0 # border whitespace in fontsize units
#legend.markerscale : 1.0 # the relative size of legend markers vs. original
# the following dimensions are in axes coords
#legend.labelspacing : 0.5 # the vertical space between the legend entries in fraction of fontsize
#legend.handlelength : 2. # the length of the legend lines in fraction of fontsize

5.2. Dynamic rc settings
Matplotlib, Release 1.3.0

#legend.handleheight : 0.7    # the height of the legend handle in fraction of fontsize
#legend.handletextpad : 0.8    # the space between the legend line and legend text in fraction of fontsize
#legend.borderaxespad : 0.5    # the border between the axes and legend edge in fraction of fontsize
#legend.columnspacing : 2.0    # the border between the axes and legend edge in fraction of fontsize
#legend.shadow : False
#legend.frameon : True        # whether or not to draw a frame around legend
#legend.scatterpoints : 3     # number of scatter points

### FIGURE
# See http://matplotlib.org/api/figure_api.html#matplotlib.figure.Figure
#figure.figsize : 8, 6 # figure size in inches
#figure.dpi : 80 # figure dots per inch
#figure.facecolor : 0.75 # figure facecolor; 0.75 is scalar gray
#figure.edgecolor : white # figure edgecolor
#figure.autolayout : False  # When True, automatically adjust subplot parameters to make the plot fit the figure
#figure.max_open_warning : 20 # The maximum number of figures to open through the pyplot interface before emitting a warning.
#figure.autolayout : False  # If less than one this feature is disabled.

# The figure subplot parameters.  All dimensions are a fraction of the
# figure width or 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 : 0.1 # the bottom of the subplots of the figure
#figure.subplot.top : 0.9   # the top of the subplots of the figure
#figure.subplot.wspace : 0.2 # the amount of width reserved for blank space between subplots
#figure.subplot.hspace : 0.2 # the amount of height reserved for white space between subplots

### IMAGES
#image.aspect : equal  # equal | auto | a number
#image.interpolation : bilinear # see help(imshow) for options
#image.cmap : jet # gray | jet etc...
#image.lut : 256 # the size of the colormap lookup table
#image.origin : upper # lower | upper
#image.resample : False

### CONTOUR PLOTS
#contour.negative linestyle : dashed # dashed | solid

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

### SAVING FIGURES
#path.simplify : True    # When True, simplify paths by removing "invisible"
# points to reduce file size and increase rendering
# speed
#path.simplify_threshold : 0.1 # 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.

# 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 : 100 # figure dots per inch
#savefig.facecolor : white # figure facecolor when saving
#savefig.edgecolor : white # figure edgecolor when saving
#savefig.format : png # png, ps, pdf, svg
#savefig.bbox : standard # 'tight' or 'standard'.
#savefig.pad_inches : 0.1 # Padding to be used when bbox is set to 'tight'
#savefig.jpeg_quality: 95 # when a jpeg is saved, the default quality parameter.
#savefig.directory : ~ # default directory in savefig dialog box,
# leave empty to always use current working directory

# tk backend params
#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 # can be: None, ghostscript or xpdf
#    # 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)

# svg backend params
#svg.image_inline : True # write raster image data directly into the svg file
#svg.image_noscale : False # suppress scaling of raster data embedded in SVG
#svg.fonttype : 'path' # How to handle SVG fonts:
#    # 'none': Assume fonts are installed on the machine where the SVG will be viewed.
#    # 'path': Embed characters as paths -- supported by most SVG renderers
#    # 'svgfont': Embed characters as SVG fonts -- supported only by Chrome,
# docstring params
#docstring.hardcopy = False  # set this when you want to generate hardcopy docstring

# Set the verbose flags. This controls how much information
# matplotlib gives you at runtime and where it goes. The verbosity
# levels are: silent, helpful, debug, debug-annoying. Any level is
# inclusive of all the levels below it. If your setting is "debug",
# you'll get all the debug and helpful messages. When submitting
# problems to the mailing-list, please set verbose to "helpful" or "debug"
# and paste the output into your report.
#
# The "fileo" gives the destination for any calls to verbose.report.
# These objects can a filename, or a filehandle like sys.stdout.
#
# You can override the rc default verbosity from the command line by
# giving the flags --verbose-LEVEL where LEVEL is one of the legal
# levels, eg --verbose-helpful.
#
# You can access the verbose instance in your code
# from matplotlib import verbose.
#verbose.level : silent  # one of silent, helpful, debug, debug-annoying
#verbose.fileo : sys.stdout # a log filename, sys.stdout or sys.stderr

# Event keys to interact with figures/plots via keyboard.
# 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       # toggling
#keymap.home : h, r, home    # home or reset mnemonic
#keymap.back : left, c, backspace # forward / backward keys to enable
#keymap.forward : right, v   # left handed quick navigation
#keymap.pan : p              # pan mnemonic
#keymap.zoom : o             # zoom mnemonic
#keymap.save : s             # saving current figure
#keymap.quit : ctrl+w, cmd+w  # close the current figure
#keymap.grid : g             # switching on/off a grid in current axes
#keymap.yscale : l           # toggle scaling of y-axes ('log'/'linear')
#keymap.xscale : L, k        # toggle scaling of x-axes ('log'/'linear')
#keymap.all_axes : a         # enable all axes

# Control location of examples data files
#examples.directory : ''     # directory to look in for custom installation

###ANIMATION settings
#animation.writer : ffmpeg    # MovieWriter 'backend' to use
#animation.codec : mp4       # 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.ffmpegs: ''  # Additional arguments to pass to ffmpeg
#animation.avconv_path: 'avconv'  # Path to avconv binary. Without full path
# $PATH is searched
#animation.avconv_args: ''  # Additional arguments to pass to avconv
#animation.mencoder_path: 'mencoder'  # Path to mencoder binary. Without full path
# $PATH is searched
#animation.mencoder_args: ''  # Additional arguments to pass to mencoder

5.2. Dynamic rc settings
CHAPTER SIX

USING MATPLOTLIB IN A PYTHON SHELL

By default, matplotlib defers drawing until the end of the script because drawing can be an expensive operation, and you may not want to update the plot every time a single property is changed, only once after all the properties have changed.

But when working from the python shell, you usually do want to update the plot with every command, eg, after changing the `xlabel()`, or the marker style of a line. While this is simple in concept, in practice it can be tricky, because matplotlib is a graphical user interface application under the hood, and there are some tricks to make the applications work right in a python shell.

### 6.1 Ipython to the rescue

Fortunately, *ipython*, an enhanced interactive python shell, has figured out all of these tricks, and is matplotlib aware, so when you start ipython in the *pylab* mode.

```
johnh@flag:~> ipython -pylab
Python 2.4.5 (#4, Apr 12 2008, 09:09:16)
IPython 0.9.0 -- An enhanced Interactive Python.

Welcome to pylab, a matplotlib-based Python environment.
For more information, type 'help(pylab)'.
```

```
In [1]: x = randn(10000)
```

```
In [2]: hist(x, 100)
```

it sets everything up for you so interactive plotting works as you would expect it to. Call `figure()` and a figure window pops up, call `plot()` and your data appears in the figure window.

Note in the example above that we did not import any matplotlib names because in *pylab* mode, ipython will import them automatically. *ipython* also turns on `interactive` mode for you, which causes every pyplot command to trigger a figure update, and also provides a matplotlib aware `run` command to run matplotlib scripts efficiently. *ipython* will turn off interactive mode during a `run` command, and then restore the interactive state at the end of the run so you can continue tweaking the figure manually.

There has been a lot of recent work to embed *ipython*, with *pylab* support, into various GUI applications, so check on the *ipython* mailing list for the latest status.
6.2 Other python interpreters

If you can’t use ipython, and still want to use matplotlib/pylab from an interactive python shell, e.g., the
plain-ole standard python interactive interpreter, you are going to need to understand what a matplotlib
backend is What is a backend?.

With the TkAgg backend, which uses the Tkinter user interface toolkit, you can use matplotlib from an
arbitrary non-gui python shell. Just set your backend : TkAgg and interactive : True in your
matplotlibrc file (see Customizing matplotlib) and fire up python. Then:

```python
>>> from pylab import *
>>> plot([1,2,3])
>>> xlabel('hi mom')
```

should work out of the box. This is also likely to work with recent versions of the qt4agg and gtkagg
backends, and with the macosx backend on the Macintosh. Note, in batch mode, i.e. when making figures
from scripts, interactive mode can be slow since it redraws the figure with each command. So you may want
to think carefully before making this the default behavior via the matplotlibrc file instead of using the
functions listed in the next section.

Gui shells are at best problematic, because they have to run a mainloop, but interactive plotting also involves
a mainloop. Ipython has sorted all this out for the primary matplotlib backends. There may be other shells
and IDEs that also work with matplotlib in interactive mode, but one obvious candidate does not: the python
IDLE IDE is a Tkinter gui app that does not support pylab interactive mode, regardless of backend.

6.3 Controlling interactive updating

The interactive property of the pyplot interface controls whether a figure canvas is drawn on every pyplot
command. If interactive is False, then the figure state is updated on every plot command, but will only be
drawn on explicit calls to draw(). When interactive is True, then every pyplot command triggers a draw.

The pyplot interface provides 4 commands that are useful for interactive control.

- `isinteractive()` returns the interactive setting True|False
- `ion()` turns interactive mode on
- `ioff()` turns interactive mode off
- `draw()` forces a figure redraw

When working with a big figure in which drawing is expensive, you may want to turn matplotlib’s interactive
setting off temporarily to avoid the performance hit:

```python
>>> #create big-expensive-figure
>>> ioff()  # turn updates off
>>> title('now how much would you pay?')
>>> xticklabels(fontsize=20, color='green')
>>> draw()  # force a draw
>>> savefig('alldone', dpi=300)
>>> close()
```
>>> ion()  # turn updating back on
>>> plot(rand(20), mfc='g', mec='r', ms=40, mew=4, ls='--', lw=3)
7.1 Text introduction

matplotlib has excellent text support, including mathematical expressions, truetype support for raster and vector outputs, newline separated text with arbitrary rotations, and unicode support. Because we embed the fonts directly in the output documents, eg for postscript or PDF, what you see on the screen is what you get in the hardcopy. freetype2 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.

You have total control over every text property (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, to support mathematical expressions anywhere in your figure.

7.2 Basic text commands

The following commands are used to create text in the pyplot interface

- `text()` - add text at an arbitrary location to the Axes; matplotlib.axes.Axes.text() in the API.
- `xlabel()` - add an axis label to the x-axis; matplotlib.axes.Axes.set_xlabel() in the API.
- `ylabel()` - add an axis label to the y-axis; matplotlib.axes.Axes.set_ylabel() in the API.
- `title()` - add a title to the Axes; matplotlib.axes.Axes.set_title() in the API.
- `figtext()` - add text at an arbitrary location to the Figure; matplotlib.figure.Figure.text() in the API.
- `suptitle()` - add a title to the Figure; matplotlib.figure.Figure.suptitle() in the API.
- `annotate()` - add an annotation, with optional arrow, to the Axes; matplotlib.axes.Axes.annotate() in the API.

All of these functions create and return a matplotlib.text.Text() instance, which can be configured with a variety of font and other properties. The example below shows all of these commands in action.
# -*- coding: utf-8 -*-
import matplotlib.pyplot as plt

fig = plt.figure()
fig.suptitle('bold figure suptitle', fontsize=14, fontweight='bold')

ax = fig.add_subplot(111)
fig.subplots_adjust(top=0.85)
ax.set_title('axes title')
ax.set_xlabel('xlabel')
ax.set_ylabel('ylabel')

ax.text(3, 8, 'boxed italics text in data coords', style='italic',
        bbox={'facecolor':'red', 'alpha':0.5, 'pad':10})

ax.text(2, 6, r'an equation: $E=mc^2$', fontsize=15)

ax.text(3, 2, unicode('unicode: Institut f\374r Festk\366rperphysik', 'latin-1'))

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))

ax.axis([0, 10, 0, 10])

plt.show()
7.3 Text properties and layout

The `matplotlib.text.Text` instances have a variety of properties which can be configured via keyword arguments to the text commands (eg `title()`, `xlabel()` and `text()`).
<table>
<thead>
<tr>
<th>Property</th>
<th>Value Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>alpha</td>
<td>float</td>
</tr>
<tr>
<td>backgroundcolor</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>bbox</td>
<td>rectangle prop dict plus key 'pad' which is a pad in points</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transform.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>a Path instance and a Transform instance, a Patch</td>
</tr>
<tr>
<td>color</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>family</td>
<td>[ 'serif'</td>
</tr>
<tr>
<td>fontproperties</td>
<td>a matplotlib.font_manager.FontProperties instance</td>
</tr>
<tr>
<td>horizontalalignment or ha</td>
<td>[ 'center'</td>
</tr>
<tr>
<td>label</td>
<td>any string</td>
</tr>
<tr>
<td>linespacing</td>
<td>float</td>
</tr>
<tr>
<td>multialignment</td>
<td>['left'</td>
</tr>
<tr>
<td>name or fontname</td>
<td>string eg, ['Sans'</td>
</tr>
<tr>
<td>picker</td>
<td>[None]float</td>
</tr>
<tr>
<td>position</td>
<td>(x,y)</td>
</tr>
<tr>
<td>rotation</td>
<td>[ angle in degrees ‘vertical’</td>
</tr>
<tr>
<td>size or fontsize</td>
<td>[ size in points</td>
</tr>
<tr>
<td>style or fontstyle</td>
<td>[ ‘normal’</td>
</tr>
<tr>
<td>text</td>
<td>string or anything printable with ‘%s’ conversion</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transform transformation instance</td>
</tr>
<tr>
<td>variant</td>
<td>[ ‘normal’</td>
</tr>
<tr>
<td>verticalalignment or va</td>
<td>[ ‘center’</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>weight or fontweight</td>
<td>[ ‘normal’</td>
</tr>
<tr>
<td>x</td>
<td>float</td>
</tr>
<tr>
<td>y</td>
<td>float</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

You can layout 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.

```python
import matplotlib.pyplot as plt
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 are 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',
    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)
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 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).

**Note:** On “narrow” builds of Python, if you use the STIX fonts you should also set `ps.fonttype` and `pdf.fonttype` to 3 (the default), not 42. Otherwise some characters will not be visible.

Here is a simple example:

```python
# plain text
plt.title('alpha > beta')
```

produces “alpha > beta”.

Whereas this:

```python
# math text
plt.title(r'$\alpha > \beta$')
```

produces “α > β”.

**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:

```python
# $ % & ^ _ \ { } \( \) \[ \]
```

have special meaning outside of math mode in TeX. Therefore, these characters will behave differently depending on the rcParam `text.usetex` flag. See the `usetex tutorial` for more information.

### 7.4.1 Subscripts and superscripts

To make subscripts and superscripts, use the ‘_’ and ‘^’ symbols:

```python
r'$\alpha_i > \beta_i$'
```

\[ \alpha_i > \beta_i \quad (7.1) \]

Some symbols automatically put their sub/superscripts under and over the operator. For example, to write the sum of \( x_i \) from 0 to \( \infty \), you could do:
Fractions, binomials and stacked numbers

Fractions, binomials and stacked numbers can be created with the `\frac{}{}`, `\binom{}{}` and `\stackrel{}{}` commands, respectively:

\[ \frac{3}{4} \binom{3}{4} \stackrel{3}{4} \]

produces

\[ \frac{3}{4} \binom{3}{4} \stackrel{3}{4} \]

(7.3)

Fractions can be arbitrarily nested:

\[ \frac{5 - \frac{1}{x}}{4} \]

produces

\[ \frac{5 - \frac{1}{x}}{4} \]

(7.4)

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:

\[ (\frac{5 - \frac{1}{x}}{4}) \]

(7.5)

The solution is to precede the bracket with `\left` and `\right` to inform the parser that those brackets encompass the entire object:

\[ \left( \frac{5 - \frac{1}{x}}{4} \right) \]

(7.6)

7.4.3 Radicals

Radicals can be produced with the `\sqrt{}` command. For example:

\[ \sqrt{2} \]
\sqrt{2} \quad (7.7)

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:

\[ r^{\sqrt[3]{x}} \]

(7.8)

### 7.4.4 Fonts

The default font is *italics* for mathematical symbols.

**Note:** This default can be changed using the `mathtext.default` rcParam. 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, eg, to write “sin” in a Roman font, enclose the text in a font command:

\[ s(t) = A\sin(2\omega t) \]

(7.9)

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:

\[ s(t) = \mathcal{A}\sin(2\omega t) \]

(7.10)

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 caligraphy A is squished into the sin. You can use a spacing command to add a little whitespace between them:

\[ s(t) = \mathcal{A}/\sin(2\omega t) \]

(7.11)

The choices available with all fonts are:

<table>
<thead>
<tr>
<th>Command</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>\text{Roman}</code></td>
<td>Roman</td>
</tr>
<tr>
<td><code>\text{Italic}</code></td>
<td>Italic</td>
</tr>
<tr>
<td><code>\text{Typewriter}</code></td>
<td>Typewriter</td>
</tr>
<tr>
<td><code>\text{CALLIGRAPHY}</code></td>
<td>CALLIGRAPHY</td>
</tr>
</tbody>
</table>

When using the STIX fonts, you also have the choice of:
There are also three global “font sets” to choose from, which are selected using the `mathtext.fontset` parameter in `matplotlibrc`.

**cm:** Computer Modern (TeX)

\[ \mathcal{R} \prod_{i=\alpha_i + 1}^{\infty} a_i \sin(2\pi f x_i) \]

**stix:** STIX (designed to blend well with Times)

\[ \mathcal{R} \prod_{i=\alpha_i + 1}^{\infty} a_i \sin(2\pi f x_i) \]

**stixsans:** STIX sans-serif

\[ \mathcal{R} \prod_{i=\alpha_i + 1}^{\infty} a_i \sin(2\pi f x_i) \]

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 the rcParam `mathtext.fontset` to `custom`, you can then set the following parameters, which control which font file to use for a particular set of math characters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Corresponds to</th>
</tr>
</thead>
<tbody>
<tr>
<td>mathtext.it</td>
<td><code>\mathit{}</code> or default italic</td>
</tr>
<tr>
<td>mathtext.rm</td>
<td><code>\mathrm{}</code> Roman (upright)</td>
</tr>
<tr>
<td>mathtext.tt</td>
<td><code>\mathit{}</code> Typewriter (monospace)</td>
</tr>
<tr>
<td>mathtext.bf</td>
<td><code>\mathbf{}</code> bold italic</td>
</tr>
<tr>
<td>mathtext.cal</td>
<td><code>\mathcal{}</code> calligraphic</td>
</tr>
<tr>
<td>mathtext.sf</td>
<td><code>\mathsf{}</code> sans-serif</td>
</tr>
</tbody>
</table>
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 the rcParam `mathtext.fallback_to_cm` to True which will cause the mathtext system to use characters from the default Computer Modern fonts 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.

### 7.4.5 Accents

An accent command may precede any symbol to add an accent above it. There are long and short forms for some of them.

<table>
<thead>
<tr>
<th>Command</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>\acute a or '{a}</td>
<td>\acute a</td>
</tr>
<tr>
<td>\bar a</td>
<td>\bar a</td>
</tr>
<tr>
<td>\breve a</td>
<td>\breve a</td>
</tr>
<tr>
<td>\ddot a or &quot;a</td>
<td>\ddot a</td>
</tr>
<tr>
<td>\dot a or .a</td>
<td>\dot a</td>
</tr>
<tr>
<td>\grave a or '{a}</td>
<td>\grave a</td>
</tr>
<tr>
<td>\hat a or ^a</td>
<td>\hat a</td>
</tr>
<tr>
<td>\tilde a or ~a</td>
<td>\tilde a</td>
</tr>
<tr>
<td>\vec a</td>
<td>\vec a</td>
</tr>
<tr>
<td>\overline{abc}</td>
<td>\overline{abc}</td>
</tr>
</tbody>
</table>

In addition, there are two special accents that automatically adjust to the width of the symbols below:

<table>
<thead>
<tr>
<th>Command</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>\widehat{xyz}</td>
<td>\widehat{xyz}</td>
</tr>
<tr>
<td>\widetilde{xyz}</td>
<td>\widetilde{xyz}</td>
</tr>
</tbody>
</table>

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"$\hat i \ \ \hat \imath$"
```

\[ \hat i \ \ \hat \imath \]  \hspace{1cm} (7.12)

### 7.4.6 Symbols

You can also use a large number of the TeX symbols, as in `\infty`, `\leftarrow`, `\sum`, `\int`.

**Lower-case Greek**
\[ \alpha \ \beta \ \chi \ \delta \ \gamma \ \delta \ \kappa \ \mu \ \nu \ \omega \ \phi \ \theta \ \upsilon \ \varepsilon \ \varkappa \ \varphi \ \varpi \ \varrho \ \varsigma \ \vartheta \ \zeta \ \Delta \ \Gamma \ \Lambda \ \Omega \ \Phi \ \Pi \ \Psi \ \Theta \ \Upsilon \ \Xi \ \mho \ \nabla \ \aleph \ \beth \ \daleth \ \gimel \ \bigcap \ \bigcup \ \bigodot \ \bigoplus \ \bigotimes \ \biguplus \ \bigvee \ \bigwedge \ \coprod \ \int \ \Pr \ \arccos \ \arcsin \ \arctan \ \arg \ \cos \ \cosh \ \cot \ \csc \ \deg \ \det \ \exp \ \gcd \ \ker \ \lg \ \lim \ \max \ \min \ \sin \ \sinh \ \tan \ \tanh \ \backslash \ \text{Delimiters} \ \bigcap \ \bigcup \ \bigodot \ \bigoplus \ \bigotimes \ \biguplus \ \bigvee \ \bigwedge \ \coprod \ \int \ \Pr \ \arccos \ \arcsin \ \arctan \ \arg \ \cos \ \cosh \ \cot \ \csc \ \deg \ \det \ \exp \ \gcd \ \ker \ \lg \ \lim \ \max \ \min \ \sin \ \sinh \ \tan \ \tanh \ \backslash \ \text{Standard function names} \ \text{Binary operation and relation symbols} \]
<table>
<thead>
<tr>
<th>Command</th>
<th>Rendered</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Bumpeq]</td>
<td>≎</td>
</tr>
<tr>
<td>[\Bumpeq]</td>
<td>≍</td>
</tr>
<tr>
<td>[\Cap]</td>
<td>⋒</td>
</tr>
<tr>
<td>[\Cup]</td>
<td>⋓</td>
</tr>
<tr>
<td>[\Doteq]</td>
<td>≑</td>
</tr>
<tr>
<td>[\Join]</td>
<td>☉</td>
</tr>
<tr>
<td>[\Subset]</td>
<td>⊥</td>
</tr>
<tr>
<td>[\Supset]</td>
<td>⊩</td>
</tr>
<tr>
<td>[\Vdash]</td>
<td>⊪</td>
</tr>
<tr>
<td>[\approx]</td>
<td>≈</td>
</tr>
<tr>
<td>[\approxeq]</td>
<td>≌</td>
</tr>
<tr>
<td>[\asymp]</td>
<td>≍</td>
</tr>
<tr>
<td>[\backepsilon]</td>
<td>⋁</td>
</tr>
<tr>
<td>[\backsimeq]</td>
<td>⋍</td>
</tr>
<tr>
<td>[\barwedge]</td>
<td>⊼</td>
</tr>
<tr>
<td>[\because]</td>
<td>∴</td>
</tr>
<tr>
<td>[\between]</td>
<td>≬</td>
</tr>
<tr>
<td>[\bigcirc]</td>
<td>△</td>
</tr>
<tr>
<td>[\bigtriangledown]</td>
<td>▽</td>
</tr>
<tr>
<td>[\bowtie]</td>
<td>⊥</td>
</tr>
<tr>
<td>[\boxdot]</td>
<td>⊟</td>
</tr>
<tr>
<td>[\boxminus]</td>
<td>⊞</td>
</tr>
<tr>
<td>[\boxplus]</td>
<td>⊠</td>
</tr>
<tr>
<td>[\boxtimes]</td>
<td>•</td>
</tr>
<tr>
<td>[\bullet]</td>
<td>≈</td>
</tr>
<tr>
<td>[\cap]</td>
<td>ℭ</td>
</tr>
<tr>
<td>[\cdot]</td>
<td>·</td>
</tr>
<tr>
<td>[\circ]</td>
<td>◦</td>
</tr>
<tr>
<td>[\circeq]</td>
<td>⩞</td>
</tr>
<tr>
<td>[\cong]</td>
<td>≃</td>
</tr>
<tr>
<td>[\cup]</td>
<td>⊃</td>
</tr>
<tr>
<td>[\curlyeqprec]</td>
<td>⬇</td>
</tr>
<tr>
<td>[\curlyeqsucc]</td>
<td>⬆</td>
</tr>
<tr>
<td>[\curlyvee]</td>
<td>⋯</td>
</tr>
<tr>
<td>[\curlywedge]</td>
<td>≃</td>
</tr>
<tr>
<td>[\dag]</td>
<td>†</td>
</tr>
<tr>
<td>[\ddag]</td>
<td>‡</td>
</tr>
<tr>
<td>[\div]</td>
<td>†</td>
</tr>
<tr>
<td>[\divideontimes]</td>
<td>⊠</td>
</tr>
<tr>
<td>[\doteq]</td>
<td>≐</td>
</tr>
<tr>
<td>[\doteqdot]</td>
<td>≑</td>
</tr>
<tr>
<td>[\eqcirc]</td>
<td>⩽</td>
</tr>
<tr>
<td>[\eqcolon]</td>
<td>≈</td>
</tr>
<tr>
<td>[\eqsim]</td>
<td>≡</td>
</tr>
<tr>
<td>[\eqslantgtr]</td>
<td>≳</td>
</tr>
<tr>
<td>[\eqslantless]</td>
<td>⩽</td>
</tr>
<tr>
<td>[\fallingdotseq]</td>
<td>≏</td>
</tr>
<tr>
<td>[\frown]</td>
<td>≥</td>
</tr>
<tr>
<td>[\geq]</td>
<td>⩾</td>
</tr>
<tr>
<td>[\geqq]</td>
<td>≻</td>
</tr>
<tr>
<td>[\geqslant]</td>
<td>≳</td>
</tr>
<tr>
<td>[\gnapprox]</td>
<td>⩾</td>
</tr>
<tr>
<td>[\gneqq]</td>
<td>⩾</td>
</tr>
<tr>
<td>[\gnsim]</td>
<td>⩾</td>
</tr>
<tr>
<td>[\gtrapprox]</td>
<td>⩾</td>
</tr>
<tr>
<td>[\gtrdot]</td>
<td>⩾</td>
</tr>
<tr>
<td>[\gtreqless]</td>
<td>⩾</td>
</tr>
<tr>
<td>[\gtreqqless]</td>
<td>⩾</td>
</tr>
<tr>
<td>[\gtreqqless]</td>
<td>⩾</td>
</tr>
<tr>
<td>[\gtrless]</td>
<td>⩾</td>
</tr>
<tr>
<td>[\gg]</td>
<td>⩾</td>
</tr>
<tr>
<td>[\ggg]</td>
<td>⩾</td>
</tr>
<tr>
<td>[\lnapprox]</td>
<td>⩾</td>
</tr>
<tr>
<td>[\lneq]</td>
<td>⩾</td>
</tr>
<tr>
<td>[\lesseqgtr]</td>
<td>⩾</td>
</tr>
<tr>
<td>[\lesseqqgtr]</td>
<td>⩾</td>
</tr>
<tr>
<td>[\lesseqqgtr]</td>
<td>⩾</td>
</tr>
<tr>
<td>[\ll]</td>
<td>⩾</td>
</tr>
<tr>
<td>[\lll]</td>
<td>⩾</td>
</tr>
<tr>
<td>[\lnapprox]</td>
<td>⩾</td>
</tr>
<tr>
<td>[\lnsim]</td>
<td>⩾</td>
</tr>
<tr>
<td>[\ltimes]</td>
<td>⩾</td>
</tr>
<tr>
<td>[\mid]</td>
<td>⩾</td>
</tr>
<tr>
<td>[\models]</td>
<td>⩾</td>
</tr>
<tr>
<td>[\mp]</td>
<td>⩾</td>
</tr>
<tr>
<td>[\nVDash]</td>
<td>⩾</td>
</tr>
<tr>
<td>[\nVdash]</td>
<td>⩾</td>
</tr>
<tr>
<td>[\napprox]</td>
<td>⩾</td>
</tr>
<tr>
<td>[\ncong]</td>
<td>⩾</td>
</tr>
<tr>
<td>[\ne]</td>
<td>⩾</td>
</tr>
<tr>
<td>[\neq]</td>
<td>⩾</td>
</tr>
<tr>
<td>[\nparallel]</td>
<td>⩾</td>
</tr>
<tr>
<td>[\nprec]</td>
<td>⩾</td>
</tr>
<tr>
<td>[\nsim]</td>
<td>⩾</td>
</tr>
<tr>
<td>[\nsupset]</td>
<td>⩾</td>
</tr>
<tr>
<td>[\nsupseteq]</td>
<td>⩾</td>
</tr>
<tr>
<td>[\ntriangleleft]</td>
<td>⩾</td>
</tr>
</tbody>
</table>

7.4. Writing mathematical expressions
### Arrow symbols

<p>| \ntrianglelefteq | \ntriangleright | \ntrianglerighteq |
| \nvDash | \nvdash | \odot |
| \ominus | \oplus | \oslash |
| \otimes | || \parallel | \perp |
| \pitchfork | ± \pm | &lt; \prec |
| \preccurlyeq | \preceq |
| \precapprox | \precapprox |
| \preccurlyeq | \preccurlyeq |
| \precsim | \precsim |
| \propto | \rightthreetimes | \risingdotseq |
| \sim | \simeq |
| / \slash | ⊁ \smile |
| \sqcup | \sqcap |
| \sqsubset | \sqsupset |
| \sqsubseteq | \sqsupseteq |
| \subset | \subseteq |
| \subsetneq | \subsetneqq |
| \succ | \succeq |
| \succapprox | \succapprox |
| \succcurlyeq | \succcurlyeq |
| \supset | \supseteq |
| \supsetneq | \supsetneqq |
| \therefore | \times |
| \top | \triangleleft | \trianglelefteq |
| \uplus | \varpropto |
| \vartriangleleft | \vartriangleright |
| \vdash | \vDash |
| \vee | \veebar |
| \wedge | \wr |</p>
<table>
<thead>
<tr>
<th>Math Symbol</th>
<th>LaTeX Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>\Downarrow</td>
<td>\leftarrow</td>
</tr>
<tr>
<td>⇔</td>
<td>\Leftarrow</td>
</tr>
<tr>
<td>⇞</td>
<td>\Lleftarrow</td>
</tr>
<tr>
<td>⇞</td>
<td>\Longleftarrow</td>
</tr>
<tr>
<td>⇞</td>
<td>\Longleftrightarrow</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
<tr>
<td>⇞</td>
<td>⇞</td>
</tr>
</tbody>
</table>
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:

ur'\$\u23ce$'

### 7.4.7 Example

Here is an example illustrating many of these features in context.

```python
import numpy as np
import matplotlib.pyplot as plt
t = np.arange(0.0, 2.0, 0.01)
s = np.sin(2*np.pi*t)
plt.plot(t,s)
plt.title(r'$\alpha_i > \beta_i$', fontsize=20)
plt.text(1, -0.6, r'$\sum_{i=0}^\infty x_i$', fontsize=20)
plt.text(0.6, 0.6, r'$\mathcal{A}\mathrm{sin}(2 \omega t)$', fontsize=20)
plt.xlabel('time (s)')
plt.ylabel('volts (mV)')
```
7.5 Typesetting With XeLaTeX/LuaLaTeX

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 either switching to the backend

```python
matplotlib.use('pgf')
```

or registering it for handling pdf output

```python
from matplotlib.backends.backend_pgf import FigureCanvasPgf
matplotlib.backend_bases.register_backend('pdf', FigureCanvasPgf)
```

The second 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.
Rc parameters that control the behavior of the pgf backend:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Documentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>pgf.preamble</td>
<td>Lines to be included in the LaTeX preamble</td>
</tr>
<tr>
<td>pgf.rcfonts</td>
<td>Setup fonts from rc params using the fontspec package</td>
</tr>
<tr>
<td>pgf.texsystem</td>
<td>Either “xelatex” (default), “lualatex” or “pdflatex”</td>
</tr>
</tbody>
</table>

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.

### 7.5.1 Font specification

The fonts used for obtaining the size of text elements or when compiling figures to PDF are usually defined in the matplotlib rc parameters. You can also use the LaTeX default Computer Modern fonts by clearing the lists for `font.serif`, `font.sans-serif` or `font.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.

```python
# -*- coding: utf-8 -*-
import matplotlib as mpl
mpl.use("pgf")
pgf_with_rc_fonts = {
    "font.family": "serif",
    "font.serif": [], # use latex default serif font
    "font.sans-serif": ["DejaVu Sans"], # use a specific sans-serif font
}
mpl.rcParams.update(pgf_with_rc_fonts)

import matplotlib.pyplot as plt
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(u"\(\mu\) is not $\mu\$")
plt.tight_layout(.5)
```
7.5.2 Custom preamble

Full customization is possible by adding your own commands to the preamble. Use the `pgf.preamble` parameter 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 `pgf.rcfonts`.

```python
# -*- coding: utf-8 -*-
import matplotlib as mpl
mpl.use("pgf")
pgf_with_custom_preamble = {
    "font.family": "serif",  # use serif/main font for text elements
    "text.usetex": True,     # use inline math for ticks
    "pgf.rcfonts": False,    # don't setup fonts from rc parameters
    "pgf.preamble": [        # load additional packages
        r"\usepackage{units}",      # load additional packages
        r"\usepackage{metalogo}",  
        r"\usepackage{unicode-math}" , # unicode math setup
        r"\setmathfont{xits-math.otf}" ,
        r"\setmainfont{DejaVu Serif}" , # serif font via preamble
    ]
}
mpl.rcParams.update(pgf_with_custom_preamble)
```

7.5. Typesetting With XeLaTeX/LuaLaTeX 57
7.5.3 Choosing the TeX system

The TeX system to be used by matplotlib is chosen by the `pgf.texsystem` parameter. 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.

```python
# -*- coding: utf-8 -*-
import matplotlib as mpl
mpl.use("pgf")
pgf_with_pdflatex = {
    "pgf.texsystem": "pdflatex",
    "pgf.preamble": [r"\usepackage[utf8x]{inputenc}",
                      r"\usepackage[T1]{fontenc}",
                      r"\usepackage{cmbright}"
                      ]
}
mpl.rcParams.update(pgf_with_pdflatex)

import matplotlib.pyplot as plt
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(u"µ is not $\mu$")
plt.tight_layout(.5)
```
7.5.4 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.

- The pgf.preamble rc setting 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 your 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.

- If you still need help, please see Getting help
7.6 Text rendering With LaTeX

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 8.60 or later is recommended). 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:

```
from matplotlib import rc
rc('font', **{'family': 'sans-serif', 'sans-serif': ['Helvetica']})
## for Palatino and other serif fonts use:
#rc('font', **{'family': 'serif', 'serif': ['Palatino']})
rc('text', usetex=True)
```

Here is the standard example, `tex_demo.py`:

```
from matplotlib import rc
rc('font', **{'family': 'sans-serif', 'sans-serif': ['Helvetica']})
## for Palatino and other serif fonts use:
#rc('font', **{'family': 'serif', 'serif': ['Palatino']})
rc('text', usetex=True)

---

Demo of TeX rendering.

You can use TeX to render all of your matplotlib text if the rc parameter `text.usetex` is set. This works currently on the agg and ps backends, and requires that you have tex and the other dependencies described at http://matplotlib.sf.net/matplotlib.texmanager.html properly installed on your system. The first time you run a script...
you will see a lot of output from tex and associated tools. The next time, the run may be silent, as a lot of the information is cached in ~/.tex.cache

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

# Example data
    t = np.arange(0.0, 1.0 + 0.01, 0.01)
s = np.cos(4 * np.pi * t) + 2

    plt.rc('text', usetex=True)
    plt.rc('font', family='serif')
    plt.plot(t, s)

    plt.xlabel(r'\textbf{time} (s)')
    plt.ylabel(r'\textit{voltage} (mV)', fontsize=16)
    plt.title(r'\TeX\ is Number 
          $\sum_{n=1}^{\infty} \frac{-e^{i\pi}}{2^n}$!',
              fontsize=16, color='gray')
    # Make room for the ridiculously large title.
    plt.subplots_adjust(top=0.8)

    plt.savefig('tex_demo')
    plt.show()
```

7.6. Text rendering With LaTeX
Note that display math mode ($$ e=mc^2 $$) is not supported, but adding the command `\displaystyle`, as in `tex_demo.py`, will produce the same results.

**Note:** Certain characters require special escaping in TeX, such as:
```
# $ % & ~ ^ \ { } \( \) \[ \]
```

Therefore, these characters will behave differently depending on the rcParam `text.usetex` flag.

### 7.6.1 use\text{\texttt{tex}} with unicode

It is also possible to use unicode strings with the LaTeX text manager, here is an example taken from `tex_unicode_demo.py`:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

""
This demo is tex_demo.py modified to have unicode. See that file for more information.
""

from __future__ import unicode_literals
import matplotlib as mpl
mpl.rcParams['text.usetex']=True
mpl.rcParams['text.latex.unicode']=True
```

```python
from __future__ import unicode_literals
import matplotlib as mpl
mpl.rcParams['text.usetex']=True
mpl.rcParams['text.latex.unicode']=True
```
from numpy import arange, cos, pi
from matplotlib.pyplot import (figure, axes, plot, xlabel, ylabel, title,
                               grid, savefig, show)

figure(1, figsize=(6,4))
ax = axes([0.1, 0.1, 0.8, 0.7])
t = arange(0.0, 1.0+0.01, 0.01)
s = cos(2*2*pi*t)+2
plot(t, s)
xlabel(r'	extbf{time (s)}')
ylabel(r'	extit{Velocity (\degree/sec)}', fontsize=16)
title(r'\TeX\ is Number $\sum_{n=1}^\infty \frac{-e^{i\pi}}{2^n}$!',
       fontsize=16, color='r')
grid(True)
show()
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.

### 7.6.3 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.

- The fonts look terrible on screen. You are probably running Mac OS, and there is some funny business with older versions of dvipng on the mac. Set `text.dvipnghack : True` in your matplotlibrc file.

- 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.

### 7.6.4 Troubleshooting

- Try deleting your `.matplotlib/tex.cache` directory. If you don't know where to find `.matplotlib`, see *matplotlib directory location*.

- 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*.

### 7.7 Annotating text

For a more detailed introduction to annotations, see *Annotating Axes*.

The uses of the basic `text()` command above 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.

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

fig = plt.figure()
ax = fig.add_subplot(111)

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

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

ax.set_ylim(-2,2)
plt.show()
```

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’)
Matplotlib, Release 1.3.0

<table>
<thead>
<tr>
<th>argument</th>
<th>coordinate system</th>
</tr>
</thead>
<tbody>
<tr>
<td>'figure points'</td>
<td>points from the lower left corner of the figure</td>
</tr>
<tr>
<td>'figure pixels'</td>
<td>pixels from the lower left corner of the figure</td>
</tr>
<tr>
<td>'figure fraction'</td>
<td>0.0 is lower left of figure and 1,1 is upper right</td>
</tr>
<tr>
<td>'axes points'</td>
<td>points from lower left corner of axes</td>
</tr>
<tr>
<td>'axes pixels'</td>
<td>pixels from lower left corner of axes</td>
</tr>
<tr>
<td>'axes fraction'</td>
<td>0.0 is lower left of axes and 1,1 is upper right</td>
</tr>
<tr>
<td>'data'</td>
<td>use the axes data coordinate system</td>
</tr>
</tbody>
</table>

For example to place the text coordinates in fractional axes coordinates, one could do:

```python
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. If the value is negative, however, the origin is from the (right, top) of the figure or axes, analogous to negative indexing of sequences.

Optionally, you can specify arrow properties which draws an arrow from the text to the annotated point by giving a dictionary of arrow properties in the optional keyword argument `arrowprops`.

<table>
<thead>
<tr>
<th>key</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>width</td>
<td>the width of the arrow in points</td>
</tr>
<tr>
<td>frac</td>
<td>the fraction of the arrow length occupied by the head</td>
</tr>
<tr>
<td>headwidth</td>
<td>the width of the base of the arrow head in points</td>
</tr>
<tr>
<td>shrink</td>
<td>move the tip and base some percent away from the annotated point and text</td>
</tr>
</tbody>
</table>
| **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 args like `horizontalalignment`, `verticalalignment` and `fontsize` are passed from the `~matplotlib.Axes.annotate` to the `~Text` instance.

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

fig = plt.figure()
ax = fig.add_subplot(111, polar=True)
r = np.arange(0,1,0.001)
theta = 2*np.pi*r
line, = ax.plot(theta, r, color='#ee8d18', lw=3)
ind = 800
thisr, thistheta = r[ind], theta[ind]
ax.plot([thistheta], [thisr], 'o')
ax.annotate('a polar annotation',
            xy=(thistheta, thisr),  # theta, radius
            xytext=(0.05, 0.05),   # fraction, fraction
            textcoords='figure fraction',
            arrowprops=dict(facecolor='black', shrink=0.05),
            )
```

Chapter 7. Working with text
For more on all the wild and wonderful things you can do with annotations, including fancy arrows, see
*Annotating Axes* and *pylab_examples-annotation_demo*. 

```python
horizontalalignment='left',
verticalalignment='bottom',
)
plt.show()
```
8.1 Startup commands

At the very least, you’ll need to have access to the `imshow()` function. There are a couple of ways to do it. The easy way for an interactive environment:

```bash
$ipython -pylab
```

The `imshow` function is now directly accessible (it’s in your namespace). See also *Pyplot tutorial*.

The more expressive, easier to understand later method (use this in your scripts to make it easier for others (including your future self) to read) is to use the matplotlib API (see *Artist tutorial*) where you use explicit namespaces and control object creation, etc...

```python
In [1]: import matplotlib.pyplot as plt
In [2]: import matplotlib.image as mpimg
In [3]: import numpy as np
```

Examples below will use the latter method, for clarity. In these examples, if you use the -pylab method, you can skip the “mpimg.” and “plt.” prefixes.

8.2 Importing image data into Numpy arrays

Plotting image data is supported by the Python Image Library (PIL). Natively, matplotlib only supports PNG images. The commands shown below fall back on PIL if the native read fails.

The image used in this example is a PNG file, but keep that PIL requirement in mind for your own 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...

```
In [4]: img = mpimg.imread('stinkbug.png')
Out[4]:
```

```
array([[ 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]],
       [[ 0.41176471,  0.41176471,  0.41176471],
       [ 0.41176471,  0.41176471,  0.41176471],
       [ 0.41176471,  0.41176471,  0.41176471],
       ...,
       [ 0.42745098,  0.42745098,  0.42745098],
       [ 0.42745098,  0.42745098,  0.42745098],
       [ 0.42745098,  0.42745098,  0.42745098]],
```

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 PIL 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.

**8.3 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.

```python
In [5]: imgplot = plt.imshow(img)
```

You can also plot any numpy array - just remember that the datatype must be float32 (and range from 0.0 to 1.0) or uint8.

### 8.3.1 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:

```python
In [6]: lum_img = img[:,:,0]
```

This is array slicing. You can read more in the Numpy tutorial.

```python
In [7]: imgplot = plt.imshow(lum_img)
```
Now, with a luminosity image, the default colormap (aka lookup table, LUT), is applied. The default is called jet. There are plenty of others to choose from. Let’s set some others using the `set_cmap()` method on our image plot object:

In [8]: `imgplot.set_cmap('hot')`
In [9]: imgplot.set_cmap('spectral')
There are many other colormap schemes available. See the list and images of the colormaps.

### 8.3.2 Color scale reference

It’s helpful to have an idea of what value a color represents. We can do that by adding color bars. It’s as easy as one line:

```
In [10]: plt.colorbar()
```
This adds a colorbar to your existing figure. This won’t automatically change if you change you switch to a different colormap - you have to re-create your plot, and add in the colorbar again.

### 8.3.3 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.

```
In[10]: plt.hist(lum_img.flatten(), 256, range=(0.0,1.0), fc='k', ec='k')
```
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 calling the `set_clim()` method of the image plot object.

In[11]: imgplot.set_clim(0.0,0.7)
8.3.4 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.

```
In [8]: import Image
In [9]: img = Image.open('stinkbug.png')  # Open image as PIL image object
In [10]: rsize = img.resize((img.size[0]/10,img.size[1]/10))  # Use PIL to resize
In [11]: rsizeArr = np.asarray(rsize)  # Get array back
In [12]: imgplot = plt.imshow(rsizeArr)
```

Here we have the default interpolation, bilinear, since we did not give `imshow()` any interpolation argument. Let’s try some others:

```
In [10]: imgplot.set_interpolation('nearest')
```
In [10]: imgplot.set_interpolation('bicubic')

Bicubic interpolation is often used when blowing up photos - people tend to prefer blurry over pixelated.
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 his time working with the `Artist`

There are two types of `Artist`

- 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:

```python
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`, `Image`, 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:

```python
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:

def 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 xdata')

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.
9.1 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...
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>alpha</td>
<td>The transparency - a scalar from 0-1</td>
</tr>
<tr>
<td>animated</td>
<td>A boolean that is used to facilitate animated drawing</td>
</tr>
<tr>
<td>axes</td>
<td>The axes that the Artist lives in, possibly None</td>
</tr>
<tr>
<td>clip_box</td>
<td>The bounding box that clips the Artist</td>
</tr>
<tr>
<td>clip_on</td>
<td>Whether clipping is enabled</td>
</tr>
<tr>
<td>clip_path</td>
<td>The path the artist is clipped to</td>
</tr>
<tr>
<td>contains</td>
<td>A picking function to test whether the artist contains the pick point</td>
</tr>
<tr>
<td>figure</td>
<td>The figure instance the artist lives in, possibly None</td>
</tr>
<tr>
<td>label</td>
<td>A text label (e.g., for auto-labeling)</td>
</tr>
<tr>
<td>picker</td>
<td>A python object that controls object picking</td>
</tr>
<tr>
<td>transform</td>
<td>The transformation</td>
</tr>
<tr>
<td>visible</td>
<td>A boolean whether the artist should be drawn</td>
</tr>
<tr>
<td>zorder</td>
<td>A number which determines the drawing order</td>
</tr>
</tbody>
</table>

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:

```python
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:

```python
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 `pylab`), 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
   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))
The docstrings for all of the classes also contain the **Artist** properties, so you can consult the interactive “help” or the **artists** for a listing of properties for a given object.

### 9.2 Object containers

Now that we know how to inspect and set the properties of a given object we want to configure, we need to now 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.

### 9.3 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:

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

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:

```python
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:

```
In [191]: fig = plt.figure()

In [192]: l1 = matplotlib.lines.Line2D([0, 1], [0, 1],
                                 transform=fig.transFigure, figure=fig)

In [193]: l2 = matplotlib.lines.Line2D([0, 1], [1, 0],
                                 transform=fig.transFigure, figure=fig)

In [194]: fig.lines.extend([l1, l2])

In [195]: fig.canvas.draw()
```

Here is a summary of the Artists the figure contains
### Figure attribute

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>axes</td>
<td>A list of Axes instances (includes Subplot)</td>
</tr>
<tr>
<td>patch</td>
<td>The Rectangle background</td>
</tr>
<tr>
<td>images</td>
<td>A list of FigureImages patches - useful for raw pixel display</td>
</tr>
<tr>
<td>legends</td>
<td>A list of Figure Legend instances (different from Axes.legends)</td>
</tr>
<tr>
<td>lines</td>
<td>A list of Figure Line2D instances (rarely used, see Axes.lines)</td>
</tr>
<tr>
<td>patches</td>
<td>A list of Figure patches (rarely used, see Axes.patches)</td>
</tr>
<tr>
<td>texts</td>
<td>A list Figure Text instances</td>
</tr>
</tbody>
</table>

### 9.4 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:

```python
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:

```python
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:

```python
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:

```python
In [233]: n, bins, rectangles = ax.hist(np.random.randn(1000), 50, facecolor='yellow')
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 \texttt{add_line()} and \texttt{add_patch()}. Here is an annotated interactive session illustrating what is going on:

\texttt{In [261]: fig = plt.figure()}
\texttt{In [262]: ax = fig.add_subplot(111)}

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

\texttt{# by default the axes instance is None}
\texttt{In [264]: print rect.get_axes()}
\texttt{None}

\texttt{# and the transformation instance is set to the "identity transform"}
\texttt{In [265]: print rect.get_transform()}
\texttt{<Affine object at 0x13695544>}

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

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

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

\texttt{# the default axes transformation is ax.transData}
\texttt{In [269]: print ax.transData}
\texttt{<Affine object at 0x15009ca4>}

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

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

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

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

\texttt{# we have to manually force a figure draw}
\texttt{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

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

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 size of the XAxis ticklabels using the Axes helper method:

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

Below is a summary of the Artists that the Axes contains

<table>
<thead>
<tr>
<th>Axes attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>artists</td>
<td>A list of Artist instances</td>
</tr>
<tr>
<td>patch</td>
<td>Rectangle instance for Axes background</td>
</tr>
<tr>
<td>collections</td>
<td>A list of Collection instances</td>
</tr>
<tr>
<td>images</td>
<td>A list of AxesImage</td>
</tr>
<tr>
<td>legends</td>
<td>A list of Legend instances</td>
</tr>
<tr>
<td>lines</td>
<td>A list of Line2D instances</td>
</tr>
<tr>
<td>patches</td>
<td>A list of Patch instances</td>
</tr>
<tr>
<td>texts</td>
<td>A list of Text instances</td>
</tr>
<tr>
<td>xaxis</td>
<td>matplotlib.axis.XAxis instance</td>
</tr>
<tr>
<td>yaxis</td>
<td>matplotlib.axis.YAxis instance</td>
</tr>
</tbody>
</table>

### 9.5 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 pylab modifies in calls to xlabel() and ylabel()) as well as a list of major and minor ticks. The ticks are XTick and 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 get_major_ticks() and get_minor_ticks(). Although the ticks contain all the primitives and will be covered below, the Axis methods contain accessor methods to return the tick lines, tick labels, tick locations etc.:

```
In [285]: axis = ax.xaxis

In [286]: axis.get_ticklocs()
Out[286]: array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.])

In [287]: axis.get_ticklabels()
Out[287]: <a list of 10 Text major ticklabel objects>

# 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; this can be customized
In [288]: axis.get_ticklines()
Out[288]: <a list of 20 Line2D ticklines objects>

# by default you get the major ticks back
In [291]: axis.get_ticklines()
Out[291]: <a list of 20 Line2D ticklines objects>

# but you can also ask for the minor ticks
In [292]: axis.get_ticklines(minor=True)
Out[292]: <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)

<table>
<thead>
<tr>
<th>Accessor method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_scale</td>
<td>The scale of the axis, eg ‘log’ or ‘linear’</td>
</tr>
<tr>
<td>get_view_interval</td>
<td>The interval instance of the axis view limits</td>
</tr>
<tr>
<td>get_data_interval</td>
<td>The interval instance of the axis data limits</td>
</tr>
<tr>
<td>get_gridlines</td>
<td>A list of grid lines for the Axis</td>
</tr>
<tr>
<td>get_label</td>
<td>The axis label - a Text instance</td>
</tr>
<tr>
<td>get_ticklabels</td>
<td>A list of Text instances - keyword minor=True</td>
</tr>
<tr>
<td>get_ticklines</td>
<td>A list of Line2D instances - keyword minor=True</td>
</tr>
<tr>
<td>get_ticklocs</td>
<td>A list of Tick locations - keyword minor=True</td>
</tr>
<tr>
<td>get_major_locator</td>
<td>The matplotlibticker.Locator instance for major ticks</td>
</tr>
<tr>
<td>get_major_formatter</td>
<td>The matplotlibticker.Formatter instance for major ticks</td>
</tr>
<tr>
<td>get_minor_locator</td>
<td>The matplotlibticker.Locator instance for minor ticks</td>
</tr>
<tr>
<td>get_minor_formatter</td>
<td>The matplotlibticker.Formatter instance for minor ticks</td>
</tr>
<tr>
<td>get_major_ticks</td>
<td>A list of Tick instances for major ticks</td>
</tr>
<tr>
<td>get_minor_ticks</td>
<td>A list of Tick instances for minor ticks</td>
</tr>
<tr>
<td>grid</td>
<td>Turn the grid on or off for the major or minor ticks</td>
</tr>
</tbody>
</table>

Here is an example, not recommended for its beauty, which customizes the axes and tick properties
import numpy as np
import matplotlib.pyplot as plt

# 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)
9.6 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`. In addition, there are boolean variables that determine whether the upper labels and ticks are on for the x-axis and whether the right labels and ticks are on for the y-axis.

<table>
<thead>
<tr>
<th>Tick attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tick1line</td>
<td>Line2D instance</td>
</tr>
<tr>
<td>tick2line</td>
<td>Line2D instance</td>
</tr>
<tr>
<td>gridline</td>
<td>Line2D instance</td>
</tr>
<tr>
<td>label1</td>
<td>Text instance</td>
</tr>
<tr>
<td>label2</td>
<td>Text instance</td>
</tr>
<tr>
<td>gridOn</td>
<td>boolean which determines whether to draw the tickline</td>
</tr>
<tr>
<td>tick1On</td>
<td>boolean which determines whether to draw the 1st tickline</td>
</tr>
<tr>
<td>tick2On</td>
<td>boolean which determines whether to draw the 2nd tickline</td>
</tr>
<tr>
<td>label1On</td>
<td>boolean which determines whether to draw tick label</td>
</tr>
<tr>
<td>label2On</td>
<td>boolean which determines whether to draw tick label</td>
</tr>
</tbody>
</table>

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.

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

fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(100*np.random.rand(20))
formatter = ticker.FormatStrFormatter('$.%1.2f')
ax.yaxis.set_major_formatter(formatter)

for tick in ax.yaxis.get_major_ticks():
    tick.label1On = False
    tick.label2On = True
    tick.label2.set_color('green')
```
CHAPTER TEN

CUSTOMIZING LOCATION OF SUBPLOT USING GRIDSPEC

**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 “pyplot.subplot” but uses 0-based indexing and let subplot to occupy multiple cells.

### 10.1 Basic Example of using subplot2grid

To use subplot2grid, you provide geometry of the grid and the location of the subplot in the grid. For a simple single-cell subplot:

```python
ax = plt.subplot2grid((2,2),(0, 0))
```

is identical to

```python
ax = plt.subplot(2,2,1)
```

Note that, unlike matplotlib’s subplot, the index starts from 0 in gridspec.

To create a subplot that spans multiple cells,

```python
ax2 = plt.subplot2grid((3,3), (1, 0), colspan=2)
ax3 = plt.subplot2grid((3,3), (1, 2), rowspan=2)
```

For example, the following commands

```python
ax1 = plt.subplot2grid((3,3), (0,0), colspan=3)
ax2 = plt.subplot2grid((3,3), (1,0), colspan=2)
ax3 = plt.subplot2grid((3,3), (1,2), rowspan=2)
ax4 = plt.subplot2grid((3,3), (2,0))
ax5 = plt.subplot2grid((3,3), (2,1))
```

creates
10.2 GridSpec and SubplotSpec

You can create GridSpec explicitly and use them to create a Subplot.

For example,

\[
\text{ax} = \text{plt.subplot2grid}((2,2),(0, 0))
\]

is equal to

```python
import matplotlib.gridspec as gridspec
gs = gridspec.GridSpec(2, 2)
ax = plt.subplot(gs[0, 0])
```

A gridspec instance provides array-like (2d or 1d) indexing that returns the SubplotSpec instance. For SubplotSpec that spans multiple cells, use slice.

\[
\text{ax2} = \text{plt.subplot}(	ext{gs}[1,:-1])
\]
\[
\text{ax3} = \text{plt.subplot}(	ext{gs}[1:, -1])
\]

The above example becomes

```python
gs = gridspec.GridSpec(3, 3)
ax1 = plt.subplot(gs[0, :])
```
ax2 = plt.subplot(gs[1,:-1])
ax3 = plt.subplot(gs[1:, -1])
ax4 = plt.subplot(gs[-1,0])
ax5 = plt.subplot(gs[-1,-2])

GridSpec

10.3 Adjust GridSpec layout

When a GridSpec is explicitly used, you can adjust the layout parameters of subplots that are created from the gridspec.

gs1 = gridspec.GridSpec(3, 3)
gs1.update(left=0.05, right=0.48, wspace=0.05)

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

The code below

gs1 = gridspec.GridSpec(3, 3)
gs1.update(left=0.05, right=0.48, wspace=0.05)
ax1 = plt.subplot(gs1[:-1, :])
ax2 = plt.subplot(gs1[-1, :-1])
ax3 = plt.subplot(gs1[-1, -1])
gs2 = gridspec.GridSpec(3, 3)
```python
gs2.update(left=0.55, right=0.98, hspace=0.05)
ax4 = plt.subplot(gs2[:, :-1])
ax5 = plt.subplot(gs2[::1, -1])
ax6 = plt.subplot(gs2[-1, -1])
```

This creates

**GridSpec w/ different subplotpars**

```
ax1
ax2  ax3
ax4
ax5
ax6
```

### 10.4 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.

```python
gs0 = gridspec.GridSpec(1, 2)

gs00 = gridspec.GridSpecFromSubplotSpec(3, 3, subplot_spec=gs0[0])
gs01 = gridspec.GridSpecFromSubplotSpec(3, 3, subplot_spec=gs0[1])
```
10.5 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.
10.6 GridSpec with Varying Cell Sizes

By default, GridSpec creates cells of equal sizes. You can adjust relative heights and widths of rows and columns. Note that absolute values are meaningless, only their relative ratios matter.

```python
gs = gridspec.GridSpec(2, 2,
    width_ratios=[1,2],
    height_ratios=[4,1])

ax1 = plt.subplot(gs[0])
```

ax1 = plt.subplot(gs[0])
ax2 = plt.subplot(gs[1])
ax3 = plt.subplot(gs[2])
ax4 = plt.subplot(gs[3])
**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.

### 11.1 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.

```python
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.rcParams['savefig.facecolor'] = '0.8'

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 introduces a new command `tight_layout()` that does this automatically for you.

```python
plt.tight_layout()
```
When you have multiple subplots, often you see labels of different axes overlapping each other.

```python
plt.close('all')
fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(nrows=2, ncols=2)
example_plot(ax1)
example_plot(ax2)
example_plot(ax3)
example_plot(ax4)
```
tight_layout() will also adjust spacing between subplots to minimize the overlaps.

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.

```python
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.

```python
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 Location of Subplot Using GridSpec`) will work.

```python
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).

```python
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()
```
11.1.1 Caveats

- `tight_layout()` only considers ticklabels, axis labels, and titles. 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.

- pad=0 clips some of the 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 at least larger than 0.3 is recommended.

11.1.2 Use with GridSpec

GridSpec has its own `tight_layout()` method (the pyplot api `tight_layout()` also works).

```python
plt.close('all')
fig = plt.figure()
import matplotlib.gridspec as gridspec

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

11.1. Simple Example
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).

```
gs1.tight_layout(fig, rect=[0, 0, 0.5, 1])
```
For example, this can be used for a figure with multiple gridspecs.

```python
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)
```
We may try to match the top and bottom of two grids

```python
top = min(gs1.top, gs2.top)
bottom = max(gs1.bottom, gs2.bottom)

gs1.update(top=top, bottom=bottom)
gs2.update(top=top, bottom=bottom)
```

While this should be mostly good enough, adjusting top and bottom may require adjustment of hspace also. To update hspace & vspace, we call `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.

```python
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)
```
11.1.3 Use with AxesGrid1

While limited, the axes_grid1 toolkit is also supported.

```python
plt.close('all')
fig = plt.figure()

from mpl_toolkits.axes_grid1 import Grid
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()
```

11.1. Simple Example
11.1.4 Colorbar

If you create a colorbar with the `colorbar()` command, the created colorbar is an instance of Axes, *not* Subplot, so `tight_layout` does not work. With Matplotlib v1.1, you may create a colorbar as a subplot using the gridspec.

```python
plt.close('all')
fig = plt.figure(figsize=(4, 4))
im = plt.imshow(arr, interpolation="none")
plt.colorbar(im, use_gridspec=True)
plt.tight_layout()
```

```python
plt.colorbar(im, use_gridspec=True)
plt.tight_layout()
```
Matplotlib, Release 1.3.0

0
2
4
6
8
0

2

4

6

8

90
80
70
60
50
40
30
20
10
0

Another option is to use AxesGrid1 toolkit to explicitly create an axes for colorbar.
plt.close(’all’)
fig = plt.figure(figsize=(4, 4))
im = plt.imshow(arr, interpolation="none")
from mpl_toolkits.axes_grid1 import make_axes_locatable
divider = make_axes_locatable(plt.gca())
cax = divider.append_axes("right", "5%", pad="3%")
plt.colorbar(im, cax=cax)
plt.tight_layout()

11.1. Simple Example

117


Chapter 11. Tight Layout guide
Do not proceed unless you already have read `legend()` and `matplotlib.legend.Legend`!

### 12.1 What to be displayed

The legend command has a following call signature:

```python
legend(*args, **kwargs)
```

If `len(args)` is 2, the first argument should be a list of artist to be labeled, and the second argument should a list of string labels. If `len(args)` is 0, it automatically generate the legend from label properties of the child artists by calling `get_legend_handles_labels()` method. For example, `ax.legend()` is equivalent to:

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

The `get_legend_handles_labels()` method returns a tuple of two lists, i.e., list of artists and list of labels (python string). However, it does not return all of its child artists. It returns artists that are currently supported by matplotlib.

For matplotlib v1.0 and earlier, the supported artists are as follows.

- Line2D
- Patch
- LineCollection
- RegularPolyCollection
- CircleCollection

And, `get_legend_handles_labels()` returns all artists in `ax.lines, ax.patches` and artists in `ax.collection` which are instance of `LineCollection` or `RegularPolyCollection`. The label attributes (returned by `get_label()` method) of collected artists are used as text labels. If label attribute is empty string or starts with “_”, those artists will be ignored.

Therefore, plots drawn by some `pyplot` commands are not supported by legend. For example, `fill_between()` creates `PolyCollection` that is not supported. Also support is limited for some commands that create multiple artists. For example, `errorbar()` creates multiples `Line2D` instances.
Unfortunately, there is no easy workaround when you need legend for an artist not supported by matplotlib. (You may use one of the supported artist as a proxy. See below.)

In newer version of matplotlib (v1.1 and later), the matplotlib internals are revised to support

- complex plots that creates multiple artists (e.g., bar, errorbar, etc)
- custom legend handles

See below for details of new functionality.

### 12.1.1 Adjusting the Order of Legend items

When you want to customize the list of artists to be displayed in the legend, or their order of appearance. There are a two options. First, you can keep lists of artists and labels, and explicitly use these for the first two argument of the legend call:

```python
p1, = plot([1,2,3])
p2, = plot([3,2,1])
p3, = plot([2,3,1])
legend([p2, p1], ["line 2", "line 1"])
```

Or you may use `get_legend_handles_labels()` to retrieve list of artist and labels and manipulate them before feeding them to legend call:

```python
ax = subplot(1,1,1)
p1, = ax.plot([1,2,3], label="line 1")
p2, = ax.plot([3,2,1], label="line 2")
p3, = ax.plot([2,3,1], label="line 3")

handles, labels = ax.get_legend_handles_labels()

# reverse the order
ax.legend(handles[::-1], labels[::-1])

# or sort them by labels
import operator
hl = sorted(zip(handles, labels),
            key=operator.itemgetter(1))
handles2, labels2 = zip(*hl)

ax.legend(handles2, labels2)
```

### 12.1.2 Using Proxy Artist

When you want to display legend for an artist not supported by matplotlib, you may use another artist as a proxy. For example, you may create a proxy artist without adding it to the axes (so the proxy artist will not be drawn in the main axes) and feed it to the legend function:

```python
p = Rectangle((0, 0), 1, 1, fc="r")
legend([p], ["Red Rectangle"])```
12.2 Multicolumn Legend

By specifying the keyword argument `ncol`, you can have a multi-column legend. Also, `mode="expand"` horizontally expand the legend to fill the axes area. See `legend_demo3.py` for example.

12.3 Legend location

The location of the legend can be specified by the keyword argument `loc`, either by string or a integer number.

<table>
<thead>
<tr>
<th>String</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>upper right</td>
<td>1</td>
</tr>
<tr>
<td>upper left</td>
<td>2</td>
</tr>
<tr>
<td>lower left</td>
<td>3</td>
</tr>
<tr>
<td>lower right</td>
<td>4</td>
</tr>
<tr>
<td>right</td>
<td>5</td>
</tr>
<tr>
<td>center left</td>
<td>6</td>
</tr>
<tr>
<td>center right</td>
<td>7</td>
</tr>
<tr>
<td>lower center</td>
<td>8</td>
</tr>
<tr>
<td>upper center</td>
<td>9</td>
</tr>
<tr>
<td>center</td>
<td>10</td>
</tr>
</tbody>
</table>

By default, the legend will anchor to the bbox of the axes (for legend) or the bbox of the figure (figlegend). You can specify your own bbox using `bbox_to_anchor` argument. `bbox_to_anchor` can be an instance of `BboxBase`, a tuple of 4 floats (x, y, width, height of the bbox), or a tuple of 2 floats (x, y with `width=height=0`). Unless `bbox_transform` argument is given, the coordinates (even for the bbox instance) are considered as normalized axes coordinates.

For example, if you want your axes legend located at the figure corner (instead of the axes corner):

```python
l = legend(bbox_to_anchor=(0, 0, 1, 1), bbox_transform=gcf().transFigure)
```

Also, you can place above or outer right-hand side of the axes,

```python
from matplotlib.pyplot import *

subplot(221)
plot([1,2,3], label="test1")
plot([3,2,1], label="test2")
legend(bbox_to_anchor=(0., 1.02, 1., .102), loc=3,
      ncol=2, mode="expand", borderaxespad=0.)

subplot(223)
plot([1,2,3], label="test1")
plot([3,2,1], label="test2")
legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)

show()
```
12.4 Multiple Legend

Sometime, you want to split the legend into multiple ones:

```python
p1, = plot([1,2,3])
p2, = plot([3,2,1])
legend([p1], ["Test1"], loc=1)
legend([p2], ["Test2"], loc=4)
```

However, the above code only shows the second legend. When the legend command is called, a new legend instance is created and old ones are removed from the axes. Thus, you need to manually add the removed legend.

```python
from matplotlib.pyplot import *

p1, = plot([1,2,3], label="test1")
p2, = plot([3,2,1], label="test2")

l1 = legend([p1], ["Label 1"], loc=1)
l2 = legend([p2], ["Label 2"], loc=4) # this removes l1 from the axes.
gca().add_artist(l1) # add l1 as a separate artist to the axes

show()
```
12.5 Legend of Complex Plots

In matplotlib v1.1 and later, the legend is improved to support more plot commands and ease the customization.

12.5.1 Artist Container

The Artist Container is simple class (derived from tuple) that contains multiple artists. This is introduced primarily to support legends for complex plot commands that create multiple artists.

Axes instances now have a “containers” attribute (which is a list, and this is only intended to be used for generating a legend). The items in this attribute are also returned by `get_legend_handles_labels()`.

For example, “bar” command creates a series of Rectangle patches. Previously, it returned a list of these patches. With the current change, it creates a container object of these rectangle patches (and these patches are added to Axes.patches attribute as before) and return it instead. As the container class is derived from a tuple, it should be backward-compatible. Furthermore, the container object is added to the Axes.containers attributes so that legend command can properly create a legend for the bar. Thus, you may do

```python
b1 = bar([0, 1, 2], [0.2, 0.3, 0.1], width=0.4,
         label="Bar 1", align="center")
legend()
```
b1 = bar([0, 1, 2], [0.2, 0.3, 0.1], width=0.4, align="center")
legend([b1], ["Bar 1"])

At this time of writing, however, only “bar”, “errorbar”, and “stem” are supported (hopefully the list will increase). Here is an example.

12.5.2 Legend Handler

One of the changes is that drawing of legend handles has been delegated to legend handlers. For example, Line2D instances are handled by HandlerLine2D. The mapping between the artists and their corresponding handlers are defined in a handler_map of the legend. The handler_map is a dictionary of key-handler pair, where key can be an artist instance or its class. And the handler is a Handler instance.

Let’s consider the following sample code,

```python
legend([p_1, p_2, \ldots, p_i, \ldots], ["Test 1", "Test 2", \ldots, "Test i", \ldots])
```

For each `p_i`, matplotlib

1. check if `p_i` is in the handler_map
2. if not, iterate over `type(p_i).mro()` until a matching key is found in the handler_map
Unless specified, the default handler_map is used. Below is a partial list of key-handler pairs included in the default handler map.

- **Line2D**: `legend_handler.HandlerLine2D()`
- **Patch**: `legend_handler.HandlerPatch()`
- **LineCollection**: `legend_handler.HandlerLineCollection()`
- **...**

The legend() command takes an optional argument of “handler_map”. When provided, the default handler map will be updated (using dict.update method) with the provided one.

```python
p1, = plot(x, "ro", label="test1")
p2, = plot(y, "b+", ms=10, label="test2")

my_handler = HandlerLine2D(numpoints=1)

legend(handler_map={Line2D:my_handler})
```

The above example will use *my_handler* for any Line2D instances (p1 and p2).

```python
legend(handler_map={p1:HandlerLine2D(numpoints=1)})
```

In the above example, only *p1* will be handled by *my_handler*, while others will be handled by default handlers.

The current default handler_map has handlers for errorbar and bar plots. Also, it includes an entry for tuple which is mapped to HandlerTuple. It simply plots over all the handles for items in the given tuple. For example,

```python
z = np.random.randn(10)

pla, = plt.plot(z, "ro", ms=10, mfc="r", mew=2, mec="r") # red filled circle
plb, = plt.plot(z[:5], "w+", ms=10, mec="w", mew=2) # white cross

plt.legend([[pla, (pla, plb)], ["Attr A", "Attr A+B")])
```
12.5.3 Implement a Custom Handler

Handler can be any callable object with following signature.

```python
def __call__(self, legend, orig_handle, fontsize, handlebox):
```

Where `legend` is the legend itself, `orig_handle` is the original plot \((p_i \text{ in the above example})\), `fontsize` is the font size in pixels, and `handlebox` is a OffsetBox instance. Within the call, you 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). See `legend_handler` for more details.
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.

### 13.1 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:

```python
fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(np.random.rand(10))
def onclick(event):
    print ('button=%d, x=%d, y=%d, xdata=%f, ydata=%f' % (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:

```python
fig.canvas.mpl_disconnect(cid)
```

**Note:** The canvas retains only weak references to the callbacks. Therefore if a callback is a method of a class instance, you need to retain a reference to that instance. Otherwise the instance will be garbage-collected and the callback will vanish.

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
<table>
<thead>
<tr>
<th>Event name</th>
<th>Class and description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'button_press_event'</td>
<td><code>MouseEvent</code> - mouse button is pressed</td>
</tr>
<tr>
<td>'button_release_event'</td>
<td><code>MouseEvent</code> - mouse button is released</td>
</tr>
<tr>
<td>'draw_event'</td>
<td><code>DrawEvent</code> - canvas draw</td>
</tr>
<tr>
<td>'key_press_event'</td>
<td><code>KeyEvent</code> - key is pressed</td>
</tr>
<tr>
<td>'key_release_event'</td>
<td><code>KeyEvent</code> - key is released</td>
</tr>
<tr>
<td>'motion_notify_event'</td>
<td><code>MouseEvent</code> - mouse motion</td>
</tr>
<tr>
<td>'pick_event'</td>
<td><code>PickEvent</code> - an object in the canvas is selected</td>
</tr>
<tr>
<td>'resize_event'</td>
<td><code>ResizeEvent</code> - figure canvas is resized</td>
</tr>
<tr>
<td>'scroll_event'</td>
<td><code>MouseEvent</code> - mouse scroll wheel is rolled</td>
</tr>
<tr>
<td>'figure_enter_event'</td>
<td><code>LocationEvent</code> - mouse enters a new figure</td>
</tr>
<tr>
<td>'figure_leave_event'</td>
<td><code>LocationEvent</code> - mouse leaves a figure</td>
</tr>
<tr>
<td>'axes_enter_event'</td>
<td><code>LocationEvent</code> - mouse enters a new axes</td>
</tr>
<tr>
<td>'axes_leave_event'</td>
<td><code>LocationEvent</code> - mouse leaves an axes</td>
</tr>
</tbody>
</table>

### 13.2 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:

```python
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)
```

---

128 Chapter 13. Event handling and picking
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)

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’

13.2.1 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 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

    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 the animation blit techniques discussed in the animations recipe to make the animated drawing faster and smoother.

Extra credit solution:

# draggable rectangle with the animation blit techniques; see
# http://www.scipy.org/Cookbook/Matplotlib/Animations
import numpy as np
import matplotlib.pyplot as plt

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)
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()

13.3 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

```python
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()
```
13.4 Object picking

You can enable picking by setting the `picker` property of an `Artist` (eg 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 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, eg 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.:

```python
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, eg 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 (eg all the points in the line that are within the specified epsilon tolerance)

13.4.1 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:

```python
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
    print 'onpick points:', zip(xdata[ind], ydata[ind])

fig.canvas.mpl_connect('pick_event', onpick)

plt.show()
```

13.4.2 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:

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

X = np.random.rand(100, 1000)
x = np.mean(X, axis=1)
y = 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(x, y, 'o', picker=5)  # 5 points tolerance
```

13.4. Object picking
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
    sigma=%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()
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 existing 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.

<table>
<thead>
<tr>
<th>Coordinate</th>
<th>Transformation Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>ax.transData</td>
<td>The userland data coordinate system, controlled by the xlim and ylim</td>
</tr>
<tr>
<td>axes</td>
<td>ax.transAxes</td>
<td>The coordinate system of the Axes; (0,0) is bottom left of the axes, and (1,1) is top right of the axes</td>
</tr>
<tr>
<td>figure</td>
<td>fig.transFigure</td>
<td>The coordinate system of the Figure; (0,0) is bottom left of the figure, and (1,1) is top right of the figure</td>
</tr>
<tr>
<td>display</td>
<td>None</td>
<td>This is the pixel coordinate system of the display; (0,0) is the bottom left of the display, and (width, height) is the top right of the display in pixels</td>
</tr>
</tbody>
</table>

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.

### 14.1 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.
```python
import numpy as np
import matplotlib.pyplot as plt

x = np.arange(0, 10, 0.005)
y = np.exp(-x/2.) * np.sin(2*np.pi*x)

fig = plt.figure()
ax = fig.add_subplot(111)
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:

```python
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]: array([[ 335.175, 247. ]],
```
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.

---

**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`. 

---

14.1. Data coordinates
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)

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])

14.2 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.

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

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.

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

fig = plt.figure()
ax = fig.add_subplot(111)
x, y = 10 * np.random.rand(2, 1000)
ax.plot(x, y, 'go')  # plot some data in data coordinates

circ = patches.Circle((0.5, 0.5), 0.25, transform=ax.transAxes,
                       facecolor='yellow', alpha=0.5)
ax.add_patch(circ)

plt.show()
```
14.3 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.

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
import matplotlib.transforms as transforms

fig = plt.figure()
ax = fig.add_subplot(111)

x = np.random.randn(1000)

ax.hist(x, 30)
ax.set_title(r'$\sigma=1 \dots \sigma=2$', fontsize=16)
```

142 Chapter 14. Transformations Tutorial
# 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 = patches.Rectangle((1,0), width=1, height=1,
    transform=trans, color='yellow',
    alpha=0.5)

ax.add_patch(rect)

plt.show()
14.4 Using offset transforms to create a shadow effect

One use of transformations 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. The transforms module has a helper transformation `ScaledTranslation`. It is instantiated with:

```python
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. A typical use case is to use the figure `fig.dpi_scale_trans` transformation for the `scale_trans` argument, to first scale `xt` and `yt` specified in points to display space before doing the final offset. 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. But in the example below, we'll create the offset transform ourselves. Note the use of the plus operator in:

```python
offset = transforms.ScaledTranslation(dx, dy, fig.dpi_scale_trans)
shadow_transform = ax.transData + offset
```

showing that can chain transformations using the addition operator. This code says: first apply the data transformation `ax.transData` and then translate the data by `dx` and `dy` points. 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.

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
import matplotlib.transforms as transforms

fig = plt.figure()
ax = fig.add_subplot(111)

# 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()

---

14.5 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 api-
custom_projection_example.
The object underlying all of the `matplotlib.patch` 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

```python
import matplotlib.pyplot as plt
from matplotlib.path import Path
import matplotlib.patches as patches

verts = [
    (0., 0.), # left, bottom
    (0., 1.), # left, top
    (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 = plt.figure()
ax = fig.add_subplot(111)
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

<table>
<thead>
<tr>
<th>Code</th>
<th>Vertices</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STOP</td>
<td>1 (ignored)</td>
<td>A marker for the end of the entire path (currently not required and ignored)</td>
</tr>
<tr>
<td>MOVETO</td>
<td>1</td>
<td>Pick up the pen and move to the given vertex.</td>
</tr>
<tr>
<td>LINETO</td>
<td>1</td>
<td>Draw a line from the current position to the given vertex.</td>
</tr>
<tr>
<td>CURVE3</td>
<td>2 (1 control point, 1 endpoint)</td>
<td>Draw a quadratic Bézier curve from the current position, with the given control point, to the given end point.</td>
</tr>
<tr>
<td>CURVE4</td>
<td>3 (2 control points, 1 endpoint)</td>
<td>Draw a cubic Bézier curve from the current position, with the given control points, to the given end point.</td>
</tr>
<tr>
<td>CLOSEPOLY</td>
<td>(point itself is ignored)</td>
<td>Draw a line segment to the start point of the current polyline.</td>
</tr>
</tbody>
</table>

15.1 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.
```python
import matplotlib.pyplot as plt
from matplotlib.path import Path
import matplotlib.patches as patches

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 = plt.figure()
ax = fig.add_subplot(111)
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')
ax.text(0.85, -0.05, 'P3')

ax.set_xlim(-0.1, 1.1)
ax.set_ylim(-0.1, 1.1)
plt.show()
```
15.2 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, eg 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, eg 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:

```python
# 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)

Here is the result
Do not proceed unless you already have read *Annotating text*, `text()` and `annotate()`!

### 16.1 Annotating with Text with Box

Let’s start with a simple example.

The `text()` function in the pyplot module (or `text` method of the Axes class) takes `bbox` keyword argument, and when given, a box around the text is drawn.
bbox_props = dict(boxstyle="rarrow, pad=0.3", fc="cyan", ec="b", lw=2)
t = ax.text(0, 0, "Direction", ha="center", va="center", rotation=45,
            size=15,
            bbox=bbox_props)

The patch object associated with the text can be accessed by:

bb = t.get_bbox_patch()

The return value is an instance of FancyBboxPatch and the patch properties like facecolor, edgewidth, etc. can be accessed and modified as usual. To change the shape of the box, use set_boxstyle method.

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.

<table>
<thead>
<tr>
<th>Class</th>
<th>Name</th>
<th>Attrs</th>
</tr>
</thead>
<tbody>
<tr>
<td>LArrow</td>
<td>larrow</td>
<td>pad=0.3</td>
</tr>
<tr>
<td>RArrow</td>
<td>rarrow</td>
<td>pad=0.3</td>
</tr>
<tr>
<td>Round</td>
<td>round</td>
<td>pad=0.3, rounding_size=None</td>
</tr>
<tr>
<td>Round4</td>
<td>round4</td>
<td>pad=0.3, rounding_size=None</td>
</tr>
<tr>
<td>Roundtooth</td>
<td>roundtooth</td>
<td>pad=0.3, tooth_size=None</td>
</tr>
<tr>
<td>Sawtooth</td>
<td>sawtooth</td>
<td>pad=0.3, tooth_size=None</td>
</tr>
<tr>
<td>Square</td>
<td>square</td>
<td>pad=0.3</td>
</tr>
</tbody>
</table>
Note that the attributes 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)

```python
bb.set_boxstyle("rarrow,pad=0.6")
```

### 16.2 Annotating with Arrow

The `annotate()` function in the pyplot module (or annotate method of the Axes class) is used to draw an arrow connecting two points on the plot.

```python
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 two points (xy & xytext) can be optionally drawn by specifying the `arrowprops` argument. To draw only an arrow, use empty string as the first argument.
The arrow drawing takes a few steps.

1. A connecting path between two points are created. This is controlled by `connectionstyle` key value.
2. If patch object is given (`patchA` & `patchB`), the path is clipped to avoid the patch.
3. The path is further shrunk by given amount of pixels (`shrinkA` & `shrinkB`)
4. The path is transmuted to arrow patch, which is controlled by the `arrowstyle` key value.

The creation of the connecting path between two points is controlled by `connectionstyle` key and following styles are available.
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 option only can 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).

The connecting path (after clipping and shrinking) is then mutated to an arrow patch, according to the given `arrowstyle`. 
Some arrowstyles only work with connection style that generates a quadratic-spline segment. They are `fancy`, `simple`, and `wedge`. For these arrow styles, you must use “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 in the text command, 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.
16.3 Placing Artist at the anchored location of the Axes

There are class of artist that can be placed at the anchored location of the Axes. A common example is the legend. This type of artists can be created by using the OffsetBox class. A few predefined classes are available in mpl_toolkits.axes_grid.anchored_artists.

```python
from mpl_toolkits.axes_grid.anchored_artists import AnchoredText
at = AnchoredText("Figure 1a",
                   prop=dict(size=8), frameon=True,
                   loc=2,
                   )
at.patch.set_boxstyle("round,pad=0.,rounding_size=0.2")
ax.add_artist(at)
```

![Figure 1a](image)
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 pixel). And user can add arbitrary artist to the drawing area. Note that the extents of the artists that are added to the drawing area has nothing to do with the placement of the drawing area itself. The initial size only matters.

```python
from mpl_toolkits.axes_grid.anchored_artists import AnchoredDrawingArea

ada = AnchoredDrawingArea(20, 20, 0, 0,
                           loc=1, 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 transform set (they will be overridden) and the dimension of those artists are interpreted as a pixel coordinate, i.e., the radius of the circles in above example are 10 pixel and 5 pixel, respectively.

Sometimes, you want to your artists scale with data coordinate (or other coordinate than canvas pixel). 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.

```python
from mpl_toolkits.axes_grid.anchored_artists import AnchoredAuxTransformBox

box = AnchoredAuxTransformBox(ax.transData, loc=2)
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 corresponds to 0.1 and 0.4 in data coordinate and will be automatically scaled when the view limits of the axes change.
As in the legend, the bbox_to_anchor argument can be set. Using the HPacker and VPackers, 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.

### 16.4 Using Complex Coordinate with Annotation

The Annotation in matplotlib support several types of coordinate as described in *Annotation text*. For an advanced user who wants more control, it supports a few other options.

1. **Transform** instance. For example,

```python
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")

With this, you can annotate a point in other axes.

ax1, ax2 = subplot([1, 1]), subplot([1, 2])
ax2.annotate("Test", xy=(0.5, 0.5), xycoords=ax1.transData,
xytext=(0.5, 0.5), textcoords=ax2.transData,
arrowprops=dict(arrowstyle="->"))

2. 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 it is your responsibility that the extent of the coordinate artist (an1 in above example) is
determined before an2 gets drawn. In most cases, it means that an2 needs to be drawn later than an1.

3. A callable object that returns an instance of either BboxBase or Transform. If a transform is returned,
it is same as 1 and if bbox is returned, it is same as 2. The callable object should take a single argument
of renderer instance. For example, following two commands give identical results

an2 = ax.annotate("Test 2", xy=(1, 0.5), xycoords=an1,
xytext=(30,0), textcoords="offset points")
an2 = ax.annotate("Test 2", xy=(1, 0.5), xycoords=an1.get_window_extent,
xytext=(30,0), textcoords="offset points")

4. A tuple of two coordinate specification. The first item is for x-coordinate and the second is for y-coordinate. For example,

annotate("Test", xy=(0.5, 1), xycoords=("data", "axes fraction"))

0.5 is in data coordinate, and 1 is in normalized axes coordinate. You may use an artist or transform
as with a tuple. For example,
import matplotlib.pyplot as plt

plt.figure(figsize=(3,2))
ax=plt.axes([0.1, 0.1, 0.8, 0.7])
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=(0.5, 1.), xycoords=an1,
                 xytext=(0.5,1.1), textcoords=(an1, "axes fraction"),
                 va="bottom", ha="center",
                 bbox=dict(boxstyle="round", fc="w"),
                 arrowprops=dict(arrowstyle="->"))

plt.show()

5. Sometimes, you want your annotation with some “offset points”, but not from the annotated point but from other point. OffsetFrom is a helper class for such case.

import matplotlib.pyplot as plt

plt.figure(figsize=(3,2))
ax=plt.axes([0.1, 0.1, 0.8, 0.7])
an1 = ax.annotate("Test 1", xy=(0.5, 0.5), xycoords="data",
                 va="center", ha="center",
                 bbox=dict(boxstyle="round", fc="w"))

from matplotlib.text import OffsetFrom
offset_from = OffsetFrom(an1, (0.5, 0))
an2 = ax.annotate("Test 2", xy=(0.1, 0.1), xycoords="data",
                 xytext=(0, -10), textcoords=offset_from,
                 # xytext is offset points from "xy=(0.5, 0), xycoords=an1"
                 va="top", ha="center",
                 bbox=dict(boxstyle="round", fc="w"),
                 arrowprops=dict(arrowstyle="->"))

plt.show()
You may take a look at this example `pylab_examples-annotation_demo3`.

### 16.5 Using ConnectorPatch

The ConnectorPatch is like an annotation without a text. While the annotate function is recommended in most of situation, the ConnectorPatch is useful when you want to connect points in different axes.

```python
from matplotlib.patches import ConnectionPatch
xy = (0.2, 0.2)
con = ConnectionPatch(xyA=xy, xyB=xy, coordsA="data", coordsB="data",
                     axesA=ax1, axesB=ax2)
ax2.add_artist(con)
```

The above code connects point xy in data coordinate of ax1 to point xy int data coordinate of ax2. Here is a simple example.

While the ConnectorPatch instance can be added to any axes, but you may want it to be added to the axes in the latter (?) of the axes drawing order to prevent overlap (?) by other axes.
16.5.1 Advanced Topics

16.6 Zoom effect between Axes

mpl_toolkits.axes_grid.inset_locator defines some patch classes useful for interconnect two axes. Understanding the code requires some knowledge of how mpl’s transform works. But, utilizing it will be straightforward.

16.7 Define Custom BoxStyle

You can use a custom box style. The value for the boxstyle can be a callable object in following forms:

```python
def __call__(self, x0, y0, width, height, mutation_size,
             aspect_ratio=1.):
    ""
    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.

![Image of Test figure](image.jpg)

However, it is recommended that you derive from the matplotlib.patches.BoxStyle._Base as demonstrated below.

```python
from matplotlib.path import Path
from matplotlib.patches import BoxStyle
import matplotlib.pyplot as plt

# we may derive from matplotlib.patches.BoxStyle._Base class.
# You need to override transmute method in this case.

class MyStyle(BoxStyle._Base):
    
    A simple box.

    def __init__(self, pad=0.3):
        
        The arguments need to be floating numbers and need to have default values.

        *pad*
        amount of padding

        self.pad = pad
        super(MyStyle, self).__init__()
```

16.7. Define Custom BoxStyle
```python
def transmute(self, x0, y0, width, height, mutation_size):
    """
    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.
    """

    # padding
    pad = mutation_size * self.pad

    # width and height with padding added.
    width, height = width + 2.*pad, \  
                    height + 2.*pad,

    # boundary of the padded box
    x0, y0 = x0-pad, y0-pad,
    x1, y1 = x0+width, y0 + height

    cp = [(x0, y0),
          (x1, y0), (x1, y1), (x0, y1),
          (x0-pad, (y0+y1)/2.), (x0, y0),
          (x0, y0)]

    com = [Path.MOVETO,
            Path.LINETO, Path.LINETO, Path.LINETO,
            Path.LINETO, Path.LINETO,
            Path.CLOSEPOLY]

    path = Path(cp, com)

    return path

# register the custom style
BoxStyle._style_list["angled"] = MyStyle

plt.figure(1, figsize=(3,3))
ax = plt.subplot(111)
ax.text(0.5, 0.5, "Test", size=30, va="center", ha="center", rotation=30,
        bbox=dict(boxstyle="angled,pad=0.5", alpha=0.2))

del BoxStyle._style_list["angled"]

plt.show()
```

# register the custom style
BoxStyle._style_list["angled"] = MyStyle

plt.figure(1, figsize=(3,3))
ax = plt.subplot(111)
ax.text(0.5, 0.5, "Test", size=30, va="center", ha="center", rotation=30,
        bbox=dict(boxstyle="angled,pad=0.5", alpha=0.2))

del BoxStyle._style_list["angled"]

plt.show()
Similarly, you can define custom ConnectionStyle and custom ArrowStyle. See the source code of lib/matplotlib/patches.py and check how each style class is defined.
Here is a collection of short tutorials, examples and code snippets that illustrate some of the useful idioms and tricks to make snazzier figures and overcome some matplotlib warts.

### 17.1 Sharing axis limits and views

It’s common to make two or more plots which share an axis, eg two subplots with time as a common axis. When you pan and zoom around on one, you want the other to move around with you. To facilitate this, matplotlib Axes support a `sharex` and `sharey` attribute. When you create a `subplot()` or `axes()` instance, you can pass in a keyword indicating what axes you want to share with.

```
In [96]: t = np.arange(0, 10, 0.01)

In [97]: ax1 = plt.subplot(211)

In [98]: ax1.plot(t, np.sin(2*np.pi*t))
Out[98]: [<matplotlib.lines.Line2D object at 0x98719ec>]

In [99]: ax2 = plt.subplot(212, sharex=ax1)

In [100]: ax2.plot(t, np.sin(4*np.pi*t))
Out[100]: [<matplotlib.lines.Line2D object at 0xb7d8fec>]
```

### 17.2 Easily creating subplots

In early versions of matplotlib, if you wanted to use the pythonic API and create a figure instance and from that create a grid of subplots, possibly with shared axes, it involved a fair amount of boilerplate code. e.g.

```
# old style
fig = plt.figure()
ax1 = fig.add_subplot(221)
ax2 = fig.add_subplot(222, sharex=ax1, sharey=ax1)
ax3 = fig.add_subplot(223, sharex=ax1, sharey=ax1)
ax3 = fig.add_subplot(224, sharex=ax1, sharey=ax1)
```
Fernando Perez has provided a nice top level method to create in subplots() (note the “s” at the end) everything at once, and turn off x and y sharing for the whole bunch. You can either unpack the axes individually:

```python
# new style method 1; unpack the axes
fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, sharex=True, sharey=True)
ax1.plot(x)
```
or get them back as a numrows x numcolumns object array which supports numpy indexing:

```python
# new style method 2; use an axes array
fig, axs = plt.subplots(2, 2, sharex=True, sharey=True)
axs[0,0].plot(x)
```

### 17.3 Fixing common date annoyances

matplotlib allows you to natively plots python datetime instances, and for the most part does a good job picking tick locations and string formats. There are a couple of things it does not handle so gracefully, and here are some tricks to help you work around them. We’ll load up some sample date data which contains datetime.date objects in a numpy record array:

```python
In [63]: datafile = cbook.get_sample_data('goog.npy')
In [64]: r = np.load(datafile).view(np.recarray)
In [65]: r.dtype
```

The dtype of the numpy record array for the field date is |O4 which means it is a 4-byte python object pointer; in this case the objects are datetime.date instances, which we can see when we print some samples in the ipython terminal window.

```python
In [66]: r.date
Out[66]:
array([2004-08-19, 2004-08-20, 2004-08-23, ..., 2008-10-10, 2008-10-13,
       2008-10-14], dtype=object)
```

The dttype of the numpy record array for the field date is |O4 which means it is a 4-byte python object pointer; in this case the objects are datetime.date instances, which we can see when we print some samples in the ipython terminal window.

If you plot the data,

```python
In [67]: plot(r.date, r.close)
Out[67]: [<matplotlib.lines.Line2D object at 0x92a6b6c>]
```

you will see that the x tick labels are all squashed together.
Another annoyance is that if you hover the mouse over a the window and look in the lower right corner of the matplotlib toolbar (Interactive navigation) at the x and y coordinates, you see that the x locations are formatted the same way the tick labels are, eg “Dec 2004”. What we’d like is for the location in the toolbar to have a higher degree of precision, eg giving us the exact date out mouse is hovering over. To fix the first problem, we can use `matplotlib.figure.Figure.autofmt_xdate()` and to fix the second problem we can use the `ax.fmt_xdata` attribute which can be set to any function that takes a scalar and returns a string. matplotlib has a number of date formatters built in, so we’ll use one of those.

```python
plt.close('all')
fig, ax = plt.subplots(1)
ax.plot(r.date, r.close)
# rotate and align the tick labels so they look better
fig.autofmt_xdate()

# use a more precise date string for the x axis locations in the toolbar
import matplotlib.dates as mdates
ax.fmt_xdata = mdates.DateFormatter('%Y-%m-%d')
plt.title('fig.autofmt_xdate fixes the labels')
```

17.3. Fixing common date annoyances
Now when you hover your mouse over the plotted data, you’ll see date format strings like 2004-12-01 in the toolbar.

17.4 Fill Between and Alpha

The `fill_between()` function generates a shaded region between a min and max boundary that is useful for illustrating ranges. It has a very handy `where` argument to combine filling with logical ranges, eg to just fill in a curve over some threshold value.

At its most basic level, `fill_between` can be use to enhance a graphs visual appearance. Let’s compare two graphs of a financial times with a simple line plot on the left and a filled line on the right.

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

import matplotlib.cbook as cbook

# load up some sample financial data
datafile = cbook.get_sample_data('goog.npy')
r = np.load(datafile).view(np.recarray)

# create two subplots with the shared x and y axes
fig, (ax1, ax2) = plt.subplots(1,2, sharex=True, sharey=True)

# fig.autofmt_xdate fixes the labels
```

```python
# load up some sample financial data
datafile = cbook.get_sample_data('goog.npy')
r = np.load(datafile).view(np.recarray)

# create two subplots with the shared x and y axes
fig, (ax1, ax2) = plt.subplots(1,2, sharex=True, sharey=True)
```
pricemin = r.close.min()

ax1.plot(r.date, r.close, lw=2)
ax2.fill_between(r.date, pricemin, r.close, facecolor='blue', alpha=0.5)

for ax in ax1, ax2:
    ax.grid(True)

ax1.set_ylabel('price')
for label in ax2.get_yticklabels():
    label.set_visible(False)

fig.suptitle('Google (GOOG) daily closing price')
fig.autofmt_xdate()

The alpha channel is not necessary here, but it can be used to soften colors for more visually appealing plots. In other examples, as we’ll see below, the alpha channel is functionally useful as the shaded regions can overlap and alpha allows you to see both. Note that the postscript format does not support alpha (this is a postscript limitation, not a matplotlib limitation), so when using alpha save your figures in PNG, PDF or SVG.

Our next example computes two populations of random walkers with a different mean and standard deviation of the normal distributions from which the steps are drawn. We use shared regions to plot +/- one standard deviation of the mean position of the population. Here the alpha channel is useful, not just aesthetic.
import matplotlib.pyplot as plt
import numpy as np

Nsteps, Nwalkers = 100, 250
t = np.arange(Nsteps)

# an (Nsteps x Nwalkers) array of random walk steps
S1 = 0.002 + 0.01*np.random.randn(Nsteps, Nwalkers)
S2 = 0.004 + 0.02*np.random.randn(Nsteps, Nwalkers)

# an (Nsteps x Nwalkers) array of random walker positions
X1 = S1.cumsum(axis=0)
X2 = S2.cumsum(axis=0)

# Nsteps length arrays empirical means and standard deviations of both
# populations over time
mu1 = X1.mean(axis=1)
sigma1 = X1.std(axis=1)
mu2 = X2.mean(axis=1)
sigma2 = X2.std(axis=1)

# plot it!
fig, ax = plt.subplots(1)
ax.plot(t, mu1, lw=2, label='mean population 1', color='blue')
ar.plot(t, mu1, lw=2, label='mean population 2', color='yellow')
ax.fill_between(t, mu1+sigma1, mu1-sigma1, facecolor='blue', alpha=0.5)
ax.fill_between(t, mu2+sigma2, mu2-sigma2, facecolor='yellow', alpha=0.5)
ax.set_title('random walkers empirical $\mu$ and $\pm \sigma$ interval')
ar.legend(loc='upper left')
ar.set_xlabel('num steps')
ar.set_ylabel('position')
ar.grid()
The `where` keyword argument is very handy for highlighting certain regions of the graph. `where` takes a boolean mask the same length as the x, ymin and ymax arguments, and only fills in the region where the boolean mask is True. In the example below, we simulate a single random walker and compute the analytic mean and standard deviation of the population positions. The population mean is shown as the black dashed line, and the plus/minus one sigma deviation from the mean is shown as the yellow filled region. We use the where mask `X>upper_bound` to find the region where the walker is above the one sigma boundary, and shade that region blue.

```python
np.random.seed(1234)
Nsteps = 500
t = np.arange(Nsteps)
mu = 0.002
sigma = 0.01

# the steps and position
S = mu + sigma*np.random.randn(Nsteps)
X = S.cumsum()

# the 1 sigma upper and lower analytic population bounds
lower_bound = mu*t - sigma*np.sqrt(t)
upper_bound = mu*t + sigma*np.sqrt(t)

fig, ax = plt.subplots(1)
```

17.4. Fill Between and Alpha
Another handy use of filled regions is to highlight horizontal or vertical spans of an axes – for that matplotlib has some helper functions \texttt{axhspan()} and \texttt{axvspan()} and example \texttt{pylab_examples-axhspan_demo}.

17.5 Transparent, fancy legends

Sometimes you know what your data looks like before you plot it, and may know for instance that there won’t be much data in the upper right hand corner. Then you can safely create a legend that doesn’t overlay your data:
ax.legend(loc='upper right')

Other times you don’t know where your data is, and loc='best' will try and place the legend:
ax.legend(loc='best')

but still, your legend may overlap your data, and in these cases it’s nice to make the legend frame transparent.

np.random.seed(1234)
fig, ax = plt.subplots(1)
ax.plot(np.random.randn(300), 'o-', label='normal distribution')
ax.plot(np.random.rand(300), 's-', label='uniform distribution')
ax.set_ylim(-3, 3)
leg = ax.legend(loc='best', fancybox=True)
leg.get_frame().set_alpha(0.5)
ax.set_title('fancy, transparent legends')

17.6 Placing text boxes

When decorating axes with text boxes, two useful tricks are to place the text in axes coordinates (see Transformations Tutorial), so the text doesn’t move around with changes in x or y limits. You can also use the bbox property of text to surround the text with a Patch instance – the bbox keyword argument takes a
dictionary with keys that are Patch properties.

```python
np.random.seed(1234)
fig, ax = plt.subplots(1)
x = 30*np.random.randn(10000)
mu = x.mean()
median = np.median(x)
sigma = x.std()
textstr = '$\mu=%.2f$
$\mathrm{median}=%.2f$
$\sigma=%.2f$'%(mu, median, sigma)

ax.hist(x, 50)
# these are matplotlib.patch.Patch properties
props = dict(boxstyle='round', facecolor='wheat', alpha=0.5)
# place a text box in upper left in axes coords
ax.text(0.05, 0.95, textstr, transform=ax.transAxes, fontsize=14,
verticalalignment='top', bbox=props)
```

150
100
50
0
50
100
0
100
200
300
400
500
600
$\mu = 0.48$
median = 0.54
$\sigma = 29.86$
Here you’ll find a host of example plots with the code that generated them.

### 18.1 Simple Plot

Here’s a very basic `plot()` with text labels:
18.2 Subplot demo

Multiple axes (i.e. subplots) are created with the `subplot()` command:

![A tale of 2 subplots](image)

18.3 Histograms

The `hist()` command automatically generates histograms and returns the bin counts or probabilities:
18.4 Path demo

You can add arbitrary paths in matplotlib using the `matplotlib.path` module:
18.5 mplot3d

The mplot3d toolkit (see mplot3d tutorial and mplot3d-examples-index) has support for simple 3d graphs including surface, wireframe, scatter, and bar charts.
Thanks to John Porter, Jonathon Taylor, Reinier Heeres, and Ben Root for the mplot3d toolkit. This toolkit is included with all standard matplotlib installs.

18.6 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.

### 18.7 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.
18.8 Bar charts

Bar charts are simple to create using the `bar()` command, which includes customizations such as error bars:
It’s also simple to create stacked bars (bar_stacked.py), candlestick bars (finance_demo.py), and horizontal bar charts (barh_demo.py).

### 18.9 Pie charts

The `pie()` command allows you to easily 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.
18.10 Table demo

The `table()` command adds a text table to an axes.
18.11 Scatter demo

The scatter() command 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.
18.12 Slider demo

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.
18.13 Fill demo

The `fill()` command lets you plot filled curves and polygons:
Thanks to Andrew Straw for adding this function.

18.14 Date demo

You can plot date data with major and minor ticks and custom tick formatters for both.
See `matplotlib.ticker` and `matplotlib.dates` for details and usage.

### 18.15 Financial charts

You can make sophisticated financial plots by combining the various plot functions, layout commands, and labeling tools provided by matplotlib. The following example emulates one of the financial plots in ChartDirector:

![Financial chart example](chart.png)
18.16 Basemap demo

Jeff Whitaker’s Basemap (Not distributed with matplotlib) add-on toolkit makes it possible to plot data on many different map projections. This example shows how to plot contours, markers and text on an orthographic projection, with NASA’s “blue marble” satellite image as a background.
18.17 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.

18.18 Polar plots

The `polar()` command generates polar plots.
18.19 Legends

The `legend()` command automatically generates figure legends, with MATLAB-compatible legend placement commands.
Thanks to Charles Twardy for input on the legend command.

18.20 Mathtext_examples

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 freetype2 and the BaKoMa computer modern or STIX fonts. See the matplotlib.mathtext module for additional details.
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.

18.21 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 use\_tex option.
18.22 EEG demo

You can embed matplotlib into pygtk, wx, Tk, FLTK, or Qt applications. Here is a screenshot of an EEG viewer called pbrain, which is part of the NeuroImaging in Python suite NIPY.
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:

- `user_interfaces-embedding_in_gtk2`
- `user_interfaces-embedding_in_wx2`
- `user_interfaces-mpl_with_glade`
- `user_interfaces-embedding_in_qt`
- `user_interfaces-embedding_in_tk`

### 18.23 XKCD-style sketch plots

matplotlib supports plotting in the style of xkcd.
The day I realized I could cook bacon whenever I wanted

"STOVE OWNERSHIP" FROM XKCD BY RANDALL MONROE
"THE DATA SO FAR" FROM XKCD BY RANDALL MONROE
WHAT’S NEW IN MATPLOTLIB

This page just covers the highlights – for the full story, see the CHANGELOG.

For a list of all of the issues and pull requests since the last revision, see the github-stats.

Note: Matplotlib version 1.1 is the last major release compatible with Python versions 2.4 to 2.7. Matplotlib 1.2 and later require versions 2.6, 2.7, and 3.1 and higher.
Table of Contents

- What’s new in matplotlib
  - new in matplotlib-1.3
    * New plotting features
    * Drawing
    * Text
    * Configuration (rcParams)
    * Backends
    * Documentation and examples
    * Infrastructure
  - new in matplotlib 1.2.2
    * Improved collections
    * Multiple images on same axes are correctly transparent
  - new in matplotlib-1.2
    * Python 3.x support
    * PGF/TikZ backend
    * Locator interface
    * Tri-Surface Plots
    * Control the lengths of colorbar extensions
    * Figures are picklable
    * Set default bounding box in matplotlibrc
    * New Boxplot Functionality
    * New RC parameter functionality
    * Streamplot
    * New hist functionality
    * Updated shipped dependencies
    * Face-centred colors in tripcolor plots
    * Hatching patterns in filled contour plots, with legends
    * Known issues in the matplotlib-1.2 release
  - new in matplotlib-1.1
    * Sankey Diagrams
    * Animation
    * Tight Layout
    * PyQt4, PySide, and IPython
    * Legend
    * mplot3d
    * Numerix support removed
    * Markers
    * Other improvements
  - new in matplotlib-1.0
    * HTML5/Canvas backend
    * Sophisticated subplot grid layout
    * Easy pythonic subplots
    * Contour fixes and and triplot
    * multiple calls to show supported
    * mplot3d graphs can be embedded in arbitrary axes
    * tick_params
    * Lots of performance and feature enhancements
    * Much improved software carpentry
    * Bugfix marathon
  - new in matplotlib-0.99
    * New documentation
    * mplot3d
19.1 new in matplotlib-1.3

19.1.1 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()`.
New eventplot plot type

Todd Jennings added a `eventplot()` function to create multiple rows or columns of identical line segments.
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).
Baselines for stackplot

Till Stensitzki added non-zero baselines to `stackplot()`. They may be symmetric or weighted.
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().
19.1.2 Drawing

Independent alpha values for face and edge colors

Wes Campaigane 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.

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 contourf().

Full control of the background color

Wes Campaigane 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 plt.save() 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.

19.1.3 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.

19.1.4 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:
```python
>>> 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

rcParam values (axes.xmargin and axes.ymargin) 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.

savefig.jpeg_quality added to rcParams

rcParam value savefig.jpeg_quality 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.

19.1.5 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 `savefig.directory` rcParam in `matplotlibrc`.

**19.1.6 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.

19.1.7 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 and pyparsing 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_num_figures rcParam.
19.2 new in matplotlib 1.2.2

19.2.1 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.

19.2.2 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.

19.3 new in matplotlib-1.2

19.3.1 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.

19.3.2 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*. 
19.3.3 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:

```python
>>> import matplotlib.ticker as mticker
>>> locator = mticker.MaxNLocator(nbins=5)
>>> print(locator.tick_values(10, 19.5))
```

19.3.4 Tri-Surface Plots

Damon McDougall added a new plotting method for the mplot3d toolkit called `plot_trisurf()`.

\[ \lambda = \sum_{i}^{\infty} \mu_{i}^{2} \]
19.3.5 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.
19.3.6 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.

19.3.7 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.

19.3.8 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.
19.3.9 New RC parameter functionality

Matthew Emmett added a function and a context manager to help manage RC parameters: \texttt{rc\_file()} and \texttt{rc\_context}. To load RC parameters from a file:

\begin{verbatim}
>>> mpl.rc\_file('mpl.rc')
\end{verbatim}

To temporarily use RC parameters:

\begin{verbatim}
>>> with mpl.rc\_context(fname='mpl.rc', rc={'text\_usetex': True}):
>>>     ...
\end{verbatim}

19.3.10 Streamplot

Tom Flannaghan and Tony Yu have added a new \texttt{streamplot()} function to plot the streamlines of a vector field. This has been a long-requested feature and complements the existing \texttt{quiver()} function for plotting vector fields. In addition to simply plotting the streamlines of the vector field, \texttt{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.
Chapter 19. What's new in matplotlib
19.3.11 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.

19.3.12 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

19.3.13 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.
tripcolor of Delaunay triangulation, flat shading
pcolor of Delaunay triangulation, gouraud shading

19.3. new in matplotlib-1.2
19.3.14 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.
19.3. new in matplotlib-1.2
19.3.15 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.

19.4 new in matplotlib-1.1

19.4.1 Sankey Diagrams

Kevin Davies has extended Yannick Copin’s original Sankey example into a module (sankey) and provided new examples (api-sankey_demo_basic, api-sankey_demo_links, api-sankey_demo_rankine).
19.4.2 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 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.

### 19.4.3 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

```python
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.
19.4.4 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.

19.4.5 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 Legend of Complex Plots for more detailed explanation and examples.

19.4.6 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 *mpl3d* was significantly expanded
- Axis labels and orientation improved
- Most 3D plotting functions now support empty inputs
- Ticker offset display added:

![3D plot example](image)

- **contourf()** gains *zdir* and *offset* keyword arguments. You can now do this:
19.4.7 Numerix support removed

After more than two years of deprecation warnings, Numerix support has now been completely removed from matplotlib.

19.4.8 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.

19.4.9 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.
Matplotlib, Release 1.3.0

- 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 `HBox` and `VBox` 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.

19.5 new in matplotlib-1.0

19.5.1 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.

19.5.2 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 Location of Subplot Using GridSpec](#) for a tutorial overview.
19.5.3 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.:

```python
fig, axarr = plt.subplots(2, 2)
axarr[0,0].plot([1,2,3])  # upper, left
```

See `pylab_examples-subplots_demo` for several code examples.

19.5.4 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.
19.5.5 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.

19.5.6 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.
19.5.7 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()`.

19.5.8 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

19.5.9 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.

19.5.10 Bugfix marathon

Eric Firing went on a bug fixing and closing marathon, closing over 100 bugs on the bug tracker with help from Jae-Joon Lee, Michael Droettboom, Christoph Gohlke and Michiel de Hoon.

19.6 new in matplotlib-0.99

19.6.1 New documentation


19.6.2 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 mplot3d tutorial
19.6.3 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_grid-examples-index and The Matplotlib AxesGrid Toolkit User’s Guide.
19.6.4 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 `pylab_examples-spine_placement_demo` and `matplotlib.spines.Spine`.
19.7 new in 0.98.4

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 contributed 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.

19.7.1 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`. 
19.7.2 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`. 
19.7.3 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
```

19.7.4 psd amplitude scaling

Ryan May did a lot of work to rationalize the amplitude scaling of `psd()` and friends. See `pylab_examples-
psd_demo2`. and `pylab_examples-psd_demo3`. The changes should increase MATLAB compatibility and
increase scaling options.

### 19.7.5 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.

![fill between where](image)

### 19.7.6 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 property tables in the rest docs broke or made difficult to use some of the interactive doc helpers, eg 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 output 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. -J JL

Fixed a bug in svg backend that new_figure_manager() ignores keywords arguments such as figsize, etc. -J JL

Fixed a bug that the handlelength of the new legend class set too short when numpoints=1 -J JL

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 are added. -J JL

Fixed a bug in the new legend class that didn't allowed a tuple of coordinate values as loc. -J JL

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 - J JL

Fix crash in log ticking. - MGD

Added static helper method BrokenHBarCollection.span_where and Axes/pyplot method fill_between. See
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 -- 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 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
GitHub stats for 2012/11/08 - 2013/05/29 (tag: v1.2.0)

These lists are automatically generated, and may be incomplete or contain duplicates.

The following 85 authors contributed 1428 commits.

- Adam Ginsburg
- Adrian Price-Whelan
- Alejandro Dubrovsky
- Amit Aronovitch
- Andrew Dawson
- Anton Akhmerov
- Antony Lee
- Ben Root
- Binglin Chang
- Bradley M. Froehle
- Brian Mattern
- Cameron Bates
- Carl Michal
- Chris Beaumont
- Christoph Gohlke
- Cimarron Mittelsteadt
- Damon McDougall
- Daniel Hyams
- David Trémouilles
- Eric Firing
- Francesco Montesano
• Geoffroy Billotey
• Ian Thomas
• Jae-Joon Lee
• Jake Vanderplas
• James R. Evans
• Jan-Philip Gehrcke
• Jeff Bingham
• Jeffrey Bingham
• Jens H. Nielsen
• Jens Hedegaard Nielsen
• Joe Kington
• Julien Schueller
• Julien Woillez
• Kevin Davies
• Leo Singer
• Lodato Luciano
• Martin Spacek
• Martin Teichmann
• Martin Ueding
• Matt Giuca
• Maximilian Albert
• Michael Droettboom
• Michael Welter
• Michiel de Hoon
• Min RK
• MinRK
• Nelle Varoquaux
• Nic Eggert
• Pascal Bugnion
• Paul Hobson
• Paul Ivanov
• Pauli Virtanen
• Peter Würtz
• Phil Elson
• Pierre Haessig
• Piti Ongmongkolkul
• Ryan Dale
• Ryan May
• Sandro Tosi
• Sebastian Pinnau
• Sergey Koposov
• Takeshi Kanmae
• Thomas A Caswell
• Thomas Kluyver
• Thomas Robitaille
• Till Stensitzki
• Tobias Megies
• Todd Jennings
• Tomas Kazmar
• Tony S Yu
• Víctor Terrón
• Wes Campagne
• aseagram
• burrbull
• dhyams
• drevicko
• endolith
• gitj
• jschueller
• krischer
• montefra
• pelson
• pwuertz
• torfbolt
We closed a total of 924 issues, 326 pull requests and 598 regular issues; this is the full list (generated with the script tools/github_stats.py):

Pull Requests (326):

- PR #2082: Data limits (on 1.3.x)
- PR #2070: incorrect bbox of text
- PR #2080: Fixed failing test on python3.
- PR #2079: added some comments
- PR #2077: changed URL to the current CSV API for yahoo finance
- PR #2076: Build the _windowing extension
- PR #2066: [DOC] Mathtext and matshow examples
- PR #2024: Update homepage image
- PR #2074: backend gtk and gtk3: destroy figure save dialog after use; closes #2073
- PR #2050: Added the from_levels_and_colors function.
- PR #454: Use a subdirectory of $XDG_CONFIG_HOME instead of ~/.matplotlibrc on Linux
- PR #1813: GTK segfault with GTK3 and mpl_toolkits
- PR #2069: BUG: pass kwargs to TimedAnimation
- PR #2063: Let _pcolorargs check C for consistency with X and Y; closes #1688
- PR #2065: mlab.FIFOBufferr: remove fossil line referring to nonexistent method
- PR #1975: MixedModeRenderer non-72-dpi fixes & Pgf mixed rendering
- PR #2004: Make wx and wxagg work with wx 2.9.x on Mac.
- PR #2044: Svg rasterize (rebased)
- PR #2056: backend_gtk: don’t hide FileChooserDialog; closes #1530
- PR #2053: sphinxext.ipython_directive broken
- PR #2017: qt4_editor formlayout now works with colour tuples (fixes Issue #1690)
- PR #2057: pep8 fixes in animation.py
- PR #2055: Deprecated the set_colorbar method on a scalar mappable.
- PR #1945: PEP8 testing
- PR #2042: Ensure that PY_ARRAY_UNIQUE_SYMBOL is uniquely defined for each extension
- PR #2041: Fix a number of issues in the doc build
- PR #2049: Fix parallel testing by using the multi-process safe cbook.mkdirs
- PR #2047: Fixed typos in legend docs.
- PR #2048: Tweak image path
• PR #1889: Fixed handling of bar(., bottom=None, log=True)
• PR #2036: Fix missing ticks on inverted log axis
• PR #2038: Added parameters to the xkcd function. Fixed deprecation warning on Path.
• PR #2028: Add a what's new entry for the WebAgg backend
• PR #2002: Added support for providing 1 or 2 extra colours to the contour routines to easily specify the under and over colors.
• PR #2011: Added the “cleared” method to Path, and updated the path module’s documentation.
• PR #2033: fix pstoeps function in backend_ps.py
• PR #2026: Deprecations and housecleaning
• PR #2032: ‘annotate’ ignores path_effects argument.
• PR #2030: Image pep8
• PR #2029: Type correction: float -> double
• PR #1753: Resolving Issue #1737 - MacOSX backend unicode problems in python 3.3
• PR #1925: Supported datetimes with microseconds, and those with long time series (>160 years).
• PR #1951: parallelize_tests
• PR #2020: Fixed call to path.Path.contains_point from pnpoly.
• PR #2019: Build: avoid win32-incompatible functions
• PR #1919: Issue warning if too many figures are open
• PR #1993: PS backend fails to savefig() pcolormesh with gouraud shading
• PR #2005: Fail to export properly to svg and pdf with interactive paths
• PR #2016: Crash when using character with umlaut
• PR #2015: Wrong text baseline with usetex.
• PR #2012: texmanager doesn’t handle list of names for font.famil
• PR #2010: Allow Paths to be marked as readonly
• PR #2003: Fixed hatch clipping.
• PR #2006: ValueError: stretch is invalid
• PR #956: Shared axes colorbars & finer location control
• PR #1329: Add a “sketch” path filter
• PR #1999: Setting dashes to (0,0) results in infinite loop for agg backends
• PR #1092: Better handling of scalars to plt.subplot(). Fixes #880
• PR #1950: Tidy up the matplotlib.__init__ documentation.
• PR #1770: strange output from wx and wxagg when trying to render to JPEG or TIFF
- PR #1998: Wx backend broken
- PR #1917: Make axis.set_scale private
- PR #1927: Workaround for Python 3 with pyparsing <= 2.0.0
- PR #1885: text is not properly clipped in 1.2.1
- PR #1955: Honouring the alpha attribute when creating composite images.
- PR #1136: Configuring automatic use of tight_layout
- PR #1953: New doc build failure
- PR #1896: Doc build is full of lots of irrelevant warnings
- PR #1902: Default quit keymap - support for cmd+w on OSX
- PR #1954: Supporting different alphas for face and edge colours
- PR #1964: Fixes issue #1960. Account for right/top spine data offset on transform ...
- PR #1988: Added bar plot pickle support.
- PR #1989: Log scale pickle
- PR #1990: Fixed tight_layout pickle support.
- PR #1991: bugfix for matplotlib/ticker.py (python 3.3)
- PR #1833: Change hist behavior when normed and stacked to something more sensible
- PR #1985: horizontal histogramm doesn’t work in 1.2 branch
- PR #1984: colors.rgb_to_hsv does not work properly with array of int dtype
- PR #1982: Fix bug in SpanSelector, introduced in commit #dd325759
- PR #1978: Setting font type using rcParams does not work under Python 3.*
- PR #1976: Replace usage of Lena image in the gallery.
- PR #1977: Fix backend_driver.py
- PR #1972: SubplotBase._make_twin_axes always creates a new subplot instance
- PR #1787: Path.contains_points() incorrect return
- PR #1973: Collection’s contains method doesn’t honour offset_position attribute
- PR #1956: imsave should preserve alpha channel
- PR #1967: svg double hyphen in plot title –
- PR #1929: Fixed failing bbox_inches='tight' case when a contour collection is empty
- PR #1968: Rotated text element misalignment in Agg
- PR #1868: Fixed background colour of PNGs saved with a non-zero opacity.
- PR #1965: Make the travis output quieter on v1.2.x
- PR #1946: re-arrange mplDeprecation imports
- PR #1949: Build failes under ubuntu 13.04
- PR #1918: Tidied up some of the documentation.
- PR #1924: MEP 12: Gallery cleanup and reorganization (rebase)
- PR #1884: incorrect linkage if system PyCXX is found
- PR #1936: add pkgconfig to homebrew install instruction
- PR #1941: Use freetype-config if pkg-config is not installed
- PR #1940: Cleanup and what’s new item added for jpeg quality rcParam feature.
- PR #1771: Jpeg quality 95 by default with rendering with PIL
- PR #1935: 1836 latex docs fail
- PR #1932: DOC - two modules link appeared in the documentation
- PR #1810: Cairo + plot_date = misaligned x-axis labels
- PR #1905: Prevent Qt4 from stopping the interpreter
- PR #1861: Added a find_all method to the ReParams dictionary.
- PR #1921: Fix filename decoding when calling fc-match
- PR #1757: DOC improves documentation on the pyplot module and the bar method
- PR #1858: backend_pgf: clip paths within the backend (fixes #1857)
- PR #1913: Fix for issue #1812
- PR #1916: Normalize all ‘e.g.’ instances. Addresses issue #1423.
- PR #1908: added rcParam for x and y margin
- PR #1903: Switching b and c in _transformation_converter to fix issue #1886
- PR #1897: Doc build failure - unicode error in generate_example_rst
- PR #1915: Corrected a wrong numpy record name in documentation.
- PR #1914: Fix texmanager.dvipng_hack_alpha() to correctly use Popen.
- PR #1906: Spectral plot unit tests
- PR #1824: Support environments without a home dir or writable file system
- PR #1878: Webagg changes
- PR #1894: Exporting figure as pdf using savefig() messes up axis background in OS X
- PR #1887: Clarify documentation for FuncAnimation
- PR #1890: Restored inkscape installing on travis-ci.
- PR #1874: Building Matplotlib on Ubuntu
- PR #1186: Make default arrow head width sensible
- PR #1875: [EHN] Add frameon and savefig.frameon to rcParams
• PR #1865: Fix manual contour label positions on sparse contours
• PR #1210: Add dateutil kwargs to csv2rec
• PR #1383: More fixes for doc building with python 3
• PR #1864: fix legend w/ ‘expand’ mode which fails for a single item.
• PR #1448: ‘bbox_inches="tight"‘ support for all figure artists.
• PR #1869: Installed inkscape on the travis-ci vm.
• PR #1870: Testing documentation isn’t clear about which files to copy
• PR #1866: fix the pyplot version of rc_context
• PR #1860: Bug with PatchCollection in PDF output
• PR #1862: Matplotlib savefig() closes BytesIO object when saving in postscript format
• PR #1841: Fixes issue #1259 - Added modifier key handling for macosx backend
• PR #1816: Avoid macosx backend slowdown; issue 1563
• PR #1796: axes.grid lines using lines.marker settings?
• PR #1846: Fix the clippath rendering so that it uses no-clip unsigned chars
• PR #1853: fill_betweenx signature fixed
• PR #1854: BF - prevent a TypeError for lists of vertices
• PR #1843: test_backend_pgf: TypeError
• PR #1848: add flushing of stdout to update on key event
• PR #1802: Step linestyle
• PR #1127: Change spectral to nipy_spectral, update docs, leave aliases
• PR #1804: MEP10 - documentation improvements on set_xlabel and text of axes.py
• PR #1764: Make loc come after fontdict in set_title. Closes #1759
• PR #1825: Work around missing subprocess members on Google App Engine
• PR #1826: backend_ps: Do not write to a temporary file unless using an external distiller
• PR #1827: MEP10 - documentation improvements on many common plots: scatter plots, ...
• PR #1834: finance: Fixed making directories for explicit cachename
• PR #1832: BF - correct return type for Axes.get_title
• PR #1803: Markers module: PEP8 fixes and MEP10 documentation fixes
• PR #1795: MEP10 - refactored hlines and vlines documentation
• PR #1822: Improved triinterp_demo pylab example
• PR #1811: MultiCursor with additionnal optionnal horizontal bar
• PR #1817: Improved test_triinterp_colinear
• PR #1799: Corrupt/invalid PDF and EPS files when saving a logscaled plot made with negative values
• PR #1800: Agg snapping fixes (for the last time...?) :
• PR #1786: Cubic interpolation for triangular grids
• PR #1808: DOC: typo, break lines >80 char, add link to cmaps list
• PR #1801: Add .directory files to .gitignore
• PR #1724: Re-write stacked step histogram
• PR #1790: Fixes problem raised in #1431 ("get_transform" should not affect "is_transform_set")
• PR #1779: Bug in postscript backend in Python 3
• PR #1797: PEP8 on colors module
• PR #1291: Fix image comparison
• PR #1791: Symbol not found: _CGAffineTransformIdentity on MacOS 10.6
• PR #1794: Fix for #1792
• PR #1454: Retool the setup.py infrastructure
• PR #1785: Fix test_bbox_inches_tight
• PR #1784: Attempt to fix Travis “permission denied” error for Python 3
• PR #1775: Issue #1763
• PR #1615: Offset is empty with usetex when offset is equal to 1
• PR #1778: Fix clip_path_to_rect, add convenience method on Path object for it
• PR #1669: Add EventCollection and eventplot
• PR #1725: Fix compiler warnings
• PR #1756: Remove broken printing_in_wx.py example.
• PR #1762: Make cbook safe to import while removing duplicate is_string_like;
• PR #1252: Properly passing on horiz-/vertOn to Cursor()
• PR #1686: Fix lost ticks
• PR #1640: Fix bugs in legend positioning with loc=’best’
• PR #1687: Update lib/matplotlib/backends/backend_cairo.py
• PR #1760: Improved the subplot function documentation and fixed the autogeneration from boilerplate.
• PR #1716: PEP8 fixes on the figure module
• PR #1643: Clean up code in cbook
• PR #1755: Update examples/pylab_examples/histogram_demo_extended.py
• PR #1497: Fix for empty collection check in axes.add_collection
PR #1685: Add default savefig directory
PR #1698: Fix bug updating WeakKeyDictionary during iteration
PR #1743: slight tweak to the documentation of errorbar
PR #1748: Typo in “Annotation” docstring.
PR #1750: Name mismatch in filetypes.rgba and print_rgb of backend_bases.py
PR #1722: Fix sign of infstr in exceltools.rec2exel
PR #1726: stackplot_test_baseline has different results on 32-bit and 64-bit platforms
PR #1577: PEP8 fixes on the line module
PR #1728: Macosx backend: tweak to coordinates position
PR #1718: Fix set dashes for line collections
PR #1721: rcParams.keys() is not Python 3 compatible
PR #1699: Enable to switch off the removal of comments in csv2rec.
PR #1710: Mixing Arial with mathtext on Windows 8 fails
PR #1705: Qt closeevent fixes for v1.2.x
PR #1671: Feature stack base
PR #1684: Fix hist for log=True and histtype='step'
PR #1708: Fix breaking doc build
PR #1644: NF - Left and right side axes titles
PR #1666: Fix USE_FONTCONFIG=True mode
PR #1691: Fix svg flipping (again)
PR #1695: Alpha kwarg fix
PR #1696: Fixed doc dependency on numpy_ext.numpydoc
PR #1665: MEP10: adding numpydoc and activating autosummary
PR #1660: Explain that matplotlib must be built before the HTML documentation
PR #1694: fixes Issue #1693
PR #1682: Fixed the expected output from test_arrow_patches.test_fancyarrow.
PR #1663: Fix suptitle
PR #1675: fix “alpha” kwarg in errorbar plot
PR #1678: added QtGui.QMainWindow.closeEvent() to make sure the close event
PR #1674: Fix SVG flip when svg.image_noscale is True
PR #1680: Ignore lib/dateutil
PR #1626: Add framealpha argument for legend
• PR #1642: remove import new from cbook.py
• PR #1534: Make rc_context available via pyplot interface
• PR #1672: Nuke Travis python 3.1 testing
• PR #1670: Deprecate mpl
• PR #1635: Recompute Wedge path after change of attributes.
• PR #1498: use QMainWindow.closeEvent for close events
• PR #1617: Legend: Also calc the bbox of the legend when the frame is not drawn. (1.2.x)
• PR #1585: Fix Qt canvas resize_event
• PR #1611: change handling of legend labels which are None
• PR #1657: Add EventCollection and eventplot
• PR #1641: PEP8 fixes on the rcsetup module
• PR #1650: _png.read_png crashes on Python 3 with urllib.request object
• PR #1568: removed deprecated methods from the axes module.
• PR #1589: Fix shifted ylabels (Issue #1571)
• PR #1634: add scatterpoints to rcParam
• PR #1654: added explicit ‘zorder’ kwarg to Collection and LineCollection.
• PR #1653: Fix #570 - Reversing a 3d axis should now work properly.
• PR #1651: WebAgg: pylab compatibility
• PR #1505: Issue 1504: changed how draw handles alpha in markerfacecolor
• PR #1655: add get_segments method to collections.LineCollection
• PR #1652: Ignore kdevelop4 project files
• PR #1613: Using a stricter check to see if Python was installed as a framework.
• PR #1599: Ada Lovelace and Grace Murray Hopper images in place of Lena
• PR #1582: Linear tri interpolator
• PR #1637: change cbook to relative import
• PR #1618: Mplot3d/crashfixes
• PR #1636: hexbin log scale is broken in matplotlib 1.2.0
• PR #1624: implemented inverse transform for Mollweide axes
• PR #1630: A disconnected callback cannot be reconnected
• PR #1139: Make Axes.stem take at least one argument.
• PR #1426: WebAgg backend
• PR #1606: Document the C/C++ code guidelines
- PR #1628: Fix errorbar zorder v1.2
- PR #1620: Fix bug in _AnnotationBase
- PR #1587: Mac OS X 10.5 needs an autoreleasepool here to avoid memory leaks. Newer...
- PR #1597: new MatplotlibDeprecationWarning class (against master)
- PR #1596: new MatplotlibDeprecationWarning class (against 1.2.x)
- PR #1532: CXX/Python2/cxx_extensions.cxx:1320: Assertion ‘ob_refcnt == 0’
- PR #1604: Make font_manager ignore KeyErrors for bad fonts
- PR #1605: Change printed -> pretty-printed
- PR #1557: inverting an axis shouldn’t affect the autoscaling setting
- PR #1603: ylim=0.0 is not well handled in polar plots
- PR #1583: Crash with text.useTex=True and plt.annotate
- PR #1602: Fixed typos in docs (squashed version of #1600)
- PR #1592: Fix a syntax error in examples (movie_demo.py)
- PR #1590: Positional argument specifiers are required by Python 2.6
- PR #1579: Updated custom_projection_example.py to work with v1.2 and newer
- PR #1578: Fixed blitting in Gtk3Agg backend
- PR #1573: fix issue #1572 caused by PR #1081
- PR #1562: Mac OS X Backend: Removing clip that is no longer needed
- PR #1574: Improvements to Sankey class
- PR #1536: ENH: add AVConv movie writer for animations
- PR #1570: PEP8 fixes on the tests of the dates module
- PR #1569: FIX Removes code that does work from the axes module
- PR #1531: fix rendering slowdown with big invisible lines (issue #1256)
- PR #1398: PEP8 fixes on dates.py
- PR #1564: PEP8-compliance on axes.py (patch 4 / 4)
- PR #1559: Workaround for QT cursor bug in dock areas
- PR #1560: Remove python2.5 support from texmanager.py
- PR #1555: Geo projections getting clobbered by 2to3 when used when python3
- PR #1477: alternate fix for issue #997
- PR #1522: PEP8-compliance on axes.py (patch 3 / 4)
- PR #1550: PEP8 fixes on the module texmanager
- PR #1289: Autoscaling and limits in mplot3d.
• PR #1551: PEP8 fixes on the spines module
• PR #1537: Fix savefig.extension == “auto”
• PR #1297: pyplot.plotfile. gridon option added with default from rcParam.
• PR #1538: Remove unnecessary clip from Cairo backend; squashed commit
• PR #1544: str.format() doesn’t work on python 2.6
• PR #1549: Add citation page to website
• PR #1514: Fix streamplot when color argument has NaNs
• PR #1081: Propagate mpl.text.Text instances to the backends and fix documentation
• PR #1533: ENH: raise a more informative error
• PR #1540: Changed mailinglist archive link.
• PR #1493: check ret == False in Timer._on_timer
• PR #1523: DOC: github ribbon does not cover up index link
• PR #1515: set_cmap should not require an active image
• PR #1489: Documentation update for specgram
• PR #1527: fix 2 html color names
• PR #1524: Make README.txt consistent reStructuredText
• PR #1525: pgf: documentation enhancements
• PR #1510: pgf: documentation enhancements
• PR #1512: Reorganize the developer docs
• PR #1518: PEP8 compliance on the delaunay module
• PR #1357: PEP8 fixes on text.py
• PR #1469: PEP8-compliance on axes.py (patch 2 / 4)
• PR #1470: Add test and test-coverage to Makefile
• PR #1442: Add savefig_kwargs to Animation.save() method
• PR #1503: DOC: ‘inout’ option for tick_params direction
• PR #1494: Added sphinx documentation for Triangulation
• PR #1480: Remove dead code in patches
• PR #1496: Correct scatter docstring
• PR #1472: FIX extra comma in Sankey.add
• PR #1471: Improved checking logic of _check_xyz in contour.py
• PR #1491: Reintroduce examples.directory rc parameter
• PR #1405: Add angle kwarg to patches.Rectangle
• PR #1278: Make arrow docstring mention data transform
• PR #1355: Add sym-log normalization.
• PR #1474: use an imagemap for the “fork me on github” ribbon
• PR #1485: Fix leak of gc’s in gtkagg backend
• PR #1374: PEP8 fixes on widgets.py
• PR #1379: PEP8 fixes on quiver.py
• PR #1399: PEP8 fixes on patches
• PR #1395: PEP8 fixes on contour.py
• PR #1464: PEP8-compliance on axes.py (patch 1 / 4)
• PR #1400: PEP8 fixes on offsetbox.py
• PR #1463: Document the Gtk3 backends

Issues (598):
• #2075: Test failure in matplotlib.tests.test_colors.test_cmap_and_norm_from_levels_and_colors2
• #2061: hist(..., histtype=’step’) does not set ylim properly.
• #2081: AutoDateLocator interval bug
• #2082: Data limits (on 1.3.x)
• #854: Bug in Axes.relim when the first line is y_isdata=False and possible fix
• #2070: incorrect bbox of text
• #1063: PyQt: fill_between => Singular matrix
• #2072: PEP8 conformance tests complain about missing files
• #2080: Fixed failing test on python3.
• #2079: added some comments
• #1876: [WIP] Steppath and Line2D
• #296: 2D imagemap for 3D scatter plot
• #667: hexbin lacks a weights argument
• #2077: changed URL to the current CSV API for yahoo finance
• #602: axisartist incompatible with autofmt_xdate
• #609: Large values in histograms not showing
• #654: autofmt_xdate cropping graph wrongly
• #615: Cannot set label text size or family using axisartist
• #343: Response Spectra Tripartite Plot
• #325: EMF backend does not support bitmaps
• #281: scatter and plot should have the same kwards
• #318: ability to unshare axis
• #227: Set cap and join styles for patches
• #222: Support for amsmath in TexManager
• #214: add quote charater support to csv related functions.
• #161: one pixel error with gtkagg and blitting
• #157: Sphinx plot extension source/build directory issues
• #2076: Build the _windowing extension
• #2066: [DOC] Mathtext and matshow examples
• #2024: Update homepage image
• #2074: backend gtk and gtk3: destroy figure save dialog after use; closes #2073
• #2073: Gtk file save dialog doesn’t go ahead when clicking “Save” or “Cancel”
• #2037: PGF backend doesn’t fire draw_event when not being used as the “primary” backend
• #2050: Added the from_levels_and_colors function.
• #454: Use a subdirectory of $XDG_CONFIG_HOME instead of ~/.matplotlibrc on Linux
• #2043: Use subplots in examples (rebase)
• #1813: GTK segfault with GTK3 and mpl_toolkits
• #2069: BUG: pass kwargs to TimedAnimation
• #2063: Let _pcolorargs check C for consistency with X and Y; closes #1688
• #1688: _pcolorargs should check consistency of argument shapes
• #2065: mlab.FIFOBuффer: remove fossil line referring to nonexistent method
• #2067: Font issue while trying to save PS/EPS/SVG but not PDF
• #1975: MixedModeRenderer non-72-dpi fixes & Pgf mixed rendering
• #1821: WxAgg hangs in interactive mode
• #162: twinx and plot_date
• #1609: test_pcolormesh hangs
• #1598: Use sublots in examples
• #1185: Svg rasterize resolution fix
• #2004: Make wx and wxagg work with wx 2.9.x on Mac.
• #1530: saving a figure triggers (very) excessive IO activity
• #2044: Svg rasterize (rebased)
• #2056: backend_gtk: don’t hideFileChooserDialog; closes #1530
Matplotlib, Release 1.3.0

• #1926: Unable to pickle histogram figure
• #1690: Edit figure parameters: TypeError: argument 1 has unexpected type ‘list’
• #2053: sphinxext.ipython_directive broken
• #1997: eps files stump evince
• #2017: qt4_editor formlayout now works with colour tuples (fixes Issue #1690)
• #2057: pep8 fixes in animation.py
• #2055: Deprecated the set_colorbar method on a scalar mappable.
• #2058: mplot3d: backend_pdf.py problem with last release not present in 1.2.1rc1
• #1391: AutoDateLocator should handle sub-second intervals
• #308: Emf backend should support math text
• #1945: PEP8 testing
• #740: plt.pcolormesh and shape mismatch
• #1734: Y-axis labels are impossible to align by baseline
• #2039: PY_ARRAY_UNIQUE_SYMBOL not unique enough
• #2042: Ensure that PY_ARRAY_UNIQUE_SYMBOL is uniquely defined for each extension
• #2041: Fix a number of issues in the doc build
• #1223: dpi= for bitmaps not handled correctly
• #2049: Fix parallel testing by using the multi-process safe cbook.makedirs
• #1324: backend_pgf: open file handles on Windows
• #2047: Fixed typos in legend docs.
• #2048: Tweak image path
• #1904: Legend kwarg scatteroffsets vs. scatteryoffsets
• #1807: Regression: odd rendering of zordered areas on twinx axes in 1.2 (release) versus 1.1
• #1882: Possible regression in 1.2.1 vs 1.2.0 re bar plot with log=True
• #2031: Update screenshots page
• #1889: Fixed handling of bar(., bottom=None, log=True)
• #2036: Fix missing ticks on inverted log axis
• #2040: Cannot align subplot yaxis labels with PGF backend
• #2038: Added parameters to the xkcd function. Fixed deprecation warning on Path.
• #2028: Add a what’s new entry for the WebAgg backend
• #2009: Deprecate C++ functions in _path.cpp that are imported in path.py
• #1961: All included backends should work or be removed
• #1966: Remove deprecated code we threatened to remove for 1.3.x
• #2002: Added support for providing 1 or 2 extra colours to the contour routines to easily specify the under and over colors.
• #2011: Added the “cleared” method to Path, and updated the path module’s documentation.
• #2033: fix pstoeps function in backend_ps.py
• #2026: Deprecations and housecleaning
• #2032: ‘annotate’ ignores path_effects argument.
• #2030: Image pep8
• #1720: Can’t pickle RendererAgg in tight_layout figures
• #2029: Type correction: float -> double
• #1737: MacOSX backend unicode problems in python 3.3
• #1753: Resolving Issue #1737 - MacOSX backend unicode problems in python 3.3
• #1925: Supported datetimes with microseconds, and those with long time series (>160 years).
• #2023: imshow’s “nearest” and “none” interpolations produce smoothed images
• #1951: parallelize_tests
• #2020: Fixed call to path.Path.contains_point from pnpoly.
• #2019: Build: avoid win32-incompatible functions
• #2018: can’t create single legend line with different point types
• #1919: Issue warning if too many figures are open
• #1993: PS backend fails to savefig() pcolormesh with gouraud shading
• #2005: Fail to export properly to svg and pdf with interactive paths
• #2016: Crash when using character with umlaut
• #2015: Wrong text baseline with usetex.
• #2012: texmanager doesn’t handle list of names for font.family
• #2010: Allow Paths to be marked as readonly
• #2003: Fixed hatch clipping.
• #2006: ValueError: stretch is invalid
• #2014: Possible error in animate.py after commit cc617006f7f0a18396cecf4a9f1e222f1ee5204e
• #2013: Histogram output in PDF is mashed
• #1934: Specifying dictionary argument with dict() or braces matters in set_bbox
• #2000: Plots show up completely white
• #1994: Make wx and wxagg work with wx 2.9.x on Mac.
- #956: Shared axes colorbars & finer location control
- #1329: Add a “sketch” path filter
- #1999: Setting dashes to (0,0) results in infinite loop for agg backends
- #1199: New boxplot features
- #1898: Hatch clipping
- #1092: Better handling of scalars to plt.subplot(). Fixes #880
- #1950: Tidy up the matplotlib.__init__ documentation.
- #1855: BUG: fixed weird case where boxplot whiskers went inside box
- #1831: Unimplemented comparison method for Line3DCollection
- #1909: pathffects for Line2d object : rebase of #1015
- #1770: strange output from wx and wxagg when trying to render to JPEG or TIFF
- #1998: Wx backend broken
- #1871: set_scale and set_xscale
- #1917: Make axis.set_scale private
- #1927: Workaround for Python 3 with pyparsing <= 2.0.0
- #1885: text is not properly clipped in 1.2.1
- #1955: Honouring the alpha attribute when creating composite images.
- #1290: Debundle pyparsing
- #1040: Make ‘rstride’, ‘cstride’ default values smarter.
- #1016: Object oriented way of setting rc parameters, enabling elegant Pythonic syntax.
- #1136: Configuring automatic use of tight_layout
- #1856: Raise exception when user tries to use set_xlim or set_ylim on a geographic projection
- #1953: New doc build failure
- #1896: Doc build is full of lots of irrelevant warnings
- #1974: wx backend changes for wxPython Phoenix
- #1900: Fix building when Gtk doesn’t support version check
- #1902: Default quit keymap - support for cmd+w on OSX
- #1899: Different alphas for lines and fills.
- #1954: Supporting different alphas for face and edge colours
- #1938: Updated patch to not override alpha on edgecolor if set to none
- #1964: Fixes issue #1960. Account for right/top spine data offset on transform ...
• #1539: Pickling of log axes
• #1828: AttributeError with big float Value(s)
• #1971: Fix initialization problem with useblit on SpanSelector instance creatio...
• #1988: Added bar plot pickle support.
• #1989: Log scale pickle
• #1990: Fixed tight_layout pickle support.
• #1991: bugfix for matplotlib/ticker.py (python 3.3)
• #1833: Change hist behavior when normed and stacked to something more sensible
• #1979: developer’s guide: what is the best workflow to test modifications
• #1985: horizontal histogramm doesn’t work in 1.2 branch
• #1984: colors.rgb_to_hsv does not work properly with array of int dtypen
• #1982: Fix bug in SpanSelector, introduced in commit #dd325759
• #1978: Setting font type using reParams does not work under Python 3.*
• #1970: Build: allow local static png dependency
• #1976: Replace usage of Lena image in the gallery.
• #1977: Fix backend_driver.py
• #1944: ValueError exception in drag_zoom (tk backend)
• #1957: matplotlib 1.2 / pylab_examples example code: multiple_yaxis_with_spines.py
• #1972: SubplotBase._make_twin_axes always creates a new subplot instance
• #1787: Path.contains_points() incorrect return
• #1973: Collection’s contains method doesn’t honour offset_position attribute
• #1956: imsave should preserve alpha channel
• #1967: svg double hyphen in plot title –
• #1969: SubplotBase._make_twin_axes always creates a new subplot instance.
• #1837: html documentation: modules table and prev-next links
• #1892: possible 1.2.1 regression in ax.axhline
• #1929: Fixed failing bbox_inches='tight’ case when a contour collection is empty
• #1968: Rotated text element misalignment in Agg
• #1868: Fixed background colour of PNGs saved with a non-zero opacity.
• #1965: Make the travis output quieter on v1.2.x
• #1946: re-arrange mplDeprecation imports
• #1948: Unable to import pylab (matplotlib._png)
#1949: Build fails under ubuntu 13.04
#1918: Tidied up some of the documentation.
#1924: MEP 12: Gallery cleanup and reorganization (rebase)
#1884: incorrect linkage if system PyCXX is found
#1936: add pkgconfig to homebrew install instruction
#1941: Use freetype-config if pkg-config is not installed
#1940: Cleanup and what’s new item added for jpeg quality rcParam feature.
#1937: All text only partially displayed
#1771: Jpeg quality 95 by default with rendering with PIL
#1836: LaTeX docs build blows up
#1835: 1836 latex docs fail
#1932: DOC - two modules link appeared in the documentation
#1930: FIX Latex documentation now builds properly
#1928: Fixed polygon3d rendering bug issue #178
#1810: Cairo + plot_date = misaligned x-axis labels
#1623: MEP 12: Gallery cleanup and reorganization
#1905: Prevent Qt4 from stopping the interpreter
#1923: fix Travis failures on 2.6 and 2.7
#1922: Commit 2415c6200ebdba75a0571d71a4569f18153ff57 introduces syntax error
#1861: Added a find_all method to the RcParams dictionary.
#1879: Decode subprocess output to utf-8 or regex will fail
#1921: Fix filename decoding when calling fc-match
#1859: Fixed a bug in offsetbox
#1757: DOC improves documentation on the pyplot module and the bar method
#1767: bytes regex matching issue in font_manager.py around 1283 (line number)
#1857: pgf backend doesn’t work well with very large numbers
#1858: backend_pgf: clip paths within the backend (fixes #1857)
#1812: Error when setting arrowstyle
#1913: Fix for issue #1812
#1423: Normalize e.g. instances, or reduce them?
#1916: Normalize all ‘e.g.’ instances. Addresses issue #1423.
#1766: add rcParam to set the margin
• #1908: added rcParam for x and y margin
• #691: Inner colorbar & Outer colorbar
• #1886: MacOSX backend incorrectly displays plot/scatter under Affine2D transform
• #1903: Switching b and c in _transformation_converter to fix issue #1886
• #1897: Doc build failure - unicode error in generate_example_rst
• #1915: Corrected a wrong numpy record name in documentation.
• #1911: dvipng_hack_alpha version check is broken
• #1914: Fix texmanager.dvipng_hack_alpha() to correctly use Popen.
• #1823: Cannot import matplotlib on Google App Engine dev appserver
• #1906: Spectral plot unit tests
• #1824: Support environments without a home dir or writable file system
• #1015: patheffects for Line2d object
• #1878: Webagg changes
• #1818: Updated some of the documentation information.
• #1894: Exporting figure as pdf using savefig() messes up axis background in OS X
• #1887: Clarify documentation for FuncAnimation
• #1893: bar plot sets axhline(0) for log plots, mpl 1.2.1 disapproves
• #1890: Restored inkscape installing on travis-ci.
• #1310: Drops last tick label for some ranges
• #1874: Building Matplotlib on Ubuntu
• #1186: Make default arrow head width sensible
• #1875: [EHN] Add frameon and savefig.frameon to rcParams
• #1865: Fix manual contour label positions on sparse contours
• #208: csv2rec imports dates incorrectly and has no option
• #1356: Docs don’t build with Python3 (make.py except)
• #1210: Add dateutil kwargs to csv2rec
• #1383: More fixes for doc building with python 3
• #1864: fix legend w/ ‘expand’ mode which fails for a single item.
• #1763: Matplotlib 1.2.0 no longer respects the “bottom” argument for horizontal histograms
• #1448: ‘bbox_inches="tight"’ support for all figure artists.
• #1869: Installed inkscape on the travis-ci vm.
• #1008: Saving animation with coloured background
• #1870: Testing documentation isn’t clear about which files to copy  
• #1528: Fonts rendered are 25% larger than requested on SVG backend  
• #1256: rendering slowdown with big invisible lines  
• #1287: compare_images computes RMS incorrectly  
• #1866: fix the pyplot version of rc_context  
• #1631: histstack looks bad with alpha.  
• #1867: QT backend changes locale  
• #1860: Bug with PatchCollection in PDF output  
• #1862: Matplotlib savefig() closes BytesIO object when saving in postscript format  
• #1259: MacOS backend modifier keys  
• #1841: Fixes issue #1259 - Added modifier key handling for macosx backend  
• #1563: macosx backend slowdown with 1.2.0  
• #1816: Avoid macosx backend slowdown; issue 1563  
• #1729: request for plotting variable bin size with imshow  
• #1839: matplotlib 1.2.0 doesn’t compile with Solaris Studio 12.3 CC  
• #1796: axes.grid lines using lines.marker settings?  
• #1846: Fix the clippath renderering so that it uses no-clip unsigned chars  
• #1844: 1.2.0 regression: custom scale not working  
• #1768: Build fails on travisCI  
• #1851: Fix for the custom scale example  
• #1853: fill_betweenx signature fixed  
• #1854: BF - prevent a TypeError for lists of vertices  
• #1840: BF - prevent a TypeError for lists of vertices in set_marker  
• #1842: test_backend_pgf errors  
• #1850: fill_betweenx signature fixed  
• #1843: test_backend_pgf: TypeError  
• #1830: Keyboard shortcuts work when toolbar not displayed  
• #1848: add flushing of stdout to update on key event  
• #1802: Step linestyle  
• #879: Two colormaps named “spectral”  
• #1127: Change spectral to nipy_spectral, update docs, leave aliases  
• #1804: MEP10 - documentation improvements on set_xlabel and text of axes.py
• #1764: Make loc come after fontdict in set_title. Closes #1759
• #1759: Axes3d error on set_title
• #800: Still another Agg snapping issue.
• #1727: ‘stepfilled’ histogram is not filled properly when setting yscale(’log’)
• #1612: setupegg is broken on windows
• #1591: Image being snapped erroneously
• #1845: Agg clip rendering fix
• #1838: plot_surface and transposed arrays
• #1825: Work around missing subprocess members on Google App Engine
• #1826: backend_ps: Do not write to a temporary file unless using an external distiller
• #1827: MEP10 - documentation improvements on many common plots: scatter plots, ...
• #1834: finance: Fixed making directories for explicit cachename
• #1714: qt4_editor broken: TransformNode instances can not be copied
• #1832: BF - correct return type for Axes.get_title
• #324: ability to change curves, axes, labels attributes via UI
• #1803: Markers module: PEP8 fixes and MEP10 documentation fixes
• #1795: MEP10 - refactored hlines and vlines documentation
• #1819: Option for disregarding matplotlibrc, for reproducible batch production of plots
• #1822: Improved triinterp_demo pylab example
• #1820: griddata: Allow for easy switching between interpolation mechanisms
• #1811: MultiCursor with additionnal optionnal horizontal bar
• #1817: Improved test_triinterp_colinear
• #1799: Corrupt/invalid PDF and EPS files when saving a logscaled plot made with negative values
• #1800: Agg snapping fixes (for the last time...?) :)
• #1521: Triangular grid interpolation and refinement
• #1786: Cubic interpolation for triangular grids
• #1808: DOC: typo, break lines >80 char, add link to cmaps list
• #1798: MEP10 - documentation improvements on set_xlabel and text of axes.py
• #1801: Add .directory files to .gitignore
• #1765: Unable to Generate Docs
• #1744: bottom keyword doesn’t work for non-stacked histograms
• #1679: matplotlib-1.2.0: regression in histogram with barsticked drawing?
Matplotlib, Release 1.3.0

- #1724: Re-write stacked step histogram
- #1790: Fixes problem raised in #1431 ('get_transform' should not affect 'is_transform_set')
- #1779: Bug in postscript backend in Python 3
- #1797: PEP8 on colors module
- #1291: Fix image comparison
- #1788: Lower minimum pyparsing version to 1.5.2
- #1789: imshow() subplots with shared axes generate unwanted white spaces
- #1793: font_manager unittest errors
- #1791: Symbol not found: _CGAffineTransformIdentity on MacOS 10.6
- #1772: Python 3.3 build failure
- #1794: Fix for #1792
- #1781: Issues with installing matplotlib on Travis with Python 3
- #1792: Matplotlib fails to install pyparsing with Python 2
- #1454: Retool the setup.py infrastructure
- #1776: Documentation style suggestion
- #1785: Fix test_bbox_inches_tight
- #1784: Attempt to fix Travis “permission denied” error for Python 3
- #1775: Issue #1763
- #1615: Offset is empty with usetex when offset is equal to 1
- #1782: fix copy-to-clipboard in example
- #1778: Fix clip_path_to_rect, add convenience method on Path object for it
- #1777: PyList_SetItem return value bug in clip_path_to_rect (_path.cpp).
- #1773:.emf backend doesn’t work with StringIO
- #1669: Add EventCollection and eventplot
- #1774: ignore singleton dimensions in ndarrays passed to imshow
- #1619: Arrow with “simple” style is not robust. Code fix included.
- #1725: Fix compiler warnings
- #1756: Remove broken printing_in_wx.py example.
- #1094: Feature request - make it simpler to use full OO interface
- #1457: Better object-oriented interface for users
- #1762: Make cbook safe to import while removing duplicate is_string_like;
- #1019: Repeated is_string_like function
• #1761: plot_wireframe does not accept vmin, vmax
• #300: subplot args description confusing
• #1252: Properly passing on horiz-/vertOn to Cursor()
• #1632: Fix build on Ubuntu 12.10
• #1686: Fix lost ticks
• #1640: Fix bugs in legend positioning with loc='best'
• #1687: Update lib/matplotlib/backends/backend_cairo.py
• #1760: Improved the subplot function documentation and fixed the autogeneration from boilerplate.
• #1647: WIP: Deprecation of the cbook module
• #1662: is_string_like existed both in matplotlib and matplotlib.cbook
• #1716: PEP8 fixes on the figure module
• #1643: Clean up code in cbook
• #953: subplot docstring improvement (re #300)
• #1112: Bad kwargs to savefig
• #1755: Update examples/pylab_examples/histogram_demo_extended.py
• #1754: Fixed a typo in histogram example code
• #1490: empty scatter messes up the limits
• #1497: Fix for empty collection check in axes.add_collection
• #1685: Add default savefig directory
• #1698: Fix bug updating WeakKeyDictionary during iteration
• #1743: slight tweak to the documentation of errorbar
• #1748: Typo in “Annotation” docstring.
• #1750: Name mismatch in filetypes.rgba and print_rgb of backend_bases.py
• #1749: Incompatibility with latest stable Numpy build (v1.7)
• #1722: Fix sign of infstr in exceltools.rec2exel
• #1126: Qt4 save dialog not functional on CentOS-5
• #1740: alpha is not set correctly when using eps format
• #1741: pcolormesh memory leak
• #1726: stackplot_test_baseline has different results on 32-bit and 64-bit platforms
• #1577: PEP8 fixes on the line module
• #1728: Macosx backend: tweak to coordinates position
• #1701: dash setting in LineCollection is broken
• #1704: Contour does not pass a list of linestyles to LineCollection
• #1718: Fix set dashes for line collections
• #1721: rcParams.keys() is not Python 3 compatible
• #1723: Re-write stacked histogram (fixes bugs)
• #1706: Fix bugs in stacked histograms
• #1401: RuntimeError: dictionary changed size during iteration from colors.py, 3.3 but not 3.2
• #1699: Enable to switch off the removal of comments in csv2rec.
• #1710: Mixing Arial with mathtext on Windows 8 fails
• #1683: Remove figure from Gcf when it is closed
• #1705: Qt closeevent fixes for v1.2.x
• #1504: markerfacecolor/markeredgecolor alpha issue
• #1671: Feature stack base
• #1075: fix hist limit issue for step+ for both linear and log scale
• #1659: super hacky fix to issue #1310
• #196: Axes.hist(...log=True) mishandles y-axis minimum value
• #1029: Implemented fix to issue 196 on github for log=True and histtype='step'
• #1684: Fix hist for log=True and histtype='step'
• #1707: Docs build failure
• #1708: Fix breaking doc build
• #289: reproducible research: sys.argv[0] in plot footer
• #1633: Add rcParam option for number of scatterplot symbols
• #1113: Bug in ax.arrow()
• #987: angle/rotate keyword for rectangle
• #775: TypeError in Axes.get_legend_handles_labels
• #331: stem function ability to take one argument
• #1644: NF - Left and right side axes titles
• #1666: Fix USE_FONTCONFIG=True mode
• #1697: Fix bug updating WeakKeyDictionary during iteration
• #1691: Fix svg flipping (again)
• #1695: Alpha kwarg fix
• #1696: Fixed doc dependency on numpy_ext.numpydoc
• #1665: MEP10: adding numpydoc and activating autosummary
• #1660: Explain that matplotlib must be built before the HTML documentation
• #1693: saving to *.eps broken on master
• #1694: fixes Issue #1693
• #1689: SVG flip issue
• #1681: Fancy arrow tests are failing
• #1682: Fixed the expected output from test_arrow_patches.test_fancyarrow.
• #1262: Using figure.suptitle puts another suptitle on top of any existing one.
• #1663: Fix suptitle
• #1675: fix “alpha” kwarg in errorbar plot
• #1610: plotting legends none
• #1676: Qt close events don’t cascade properly.
• #1678: added QtGui.QMainWindow.closeEvent() to make sure the close event
• #1673: Images saved as SVG get upside down when svg.image_noscale is True.
• #1674: Fix SVG flip when svg.image_noscale is True
• #1680: Ignore lib/dateutil
• #1677: add changelog for #1626
• #1626: Add framealpha argument for legend
• #1608: Incorrect ylabel placement in twinx
• #1642: remove import new from cbook.py
• #1534: Make rc_context available via pyplot interface
• #1672: Nuke Travis python 3.1 testing
• #1535: Deprecate mpl.py (was Remove mpl.py)
• #1670: Deprecate mpl
• #1517: ENH: Add baseline feature to stackplot.
• #1635: Recompute Wedge path after change of attributes.
• #1488: Continue propagating resize event up the chain
• #1498: use QMainWindow.closeEvent for close events
• #1617: Legend: Also calc the bbox of the legend when the frame is not drawn. (1.2.x)
• #1585: Fix Qt canvas resize_event
• #1629: Update x,y,z values for an existing Line3D object
• #1611: change handling of legend labels which are None
• #1657: Add EventCollection and eventplot
Matplotlib, Release 1.3.0

- #1641: PEP8 fixes on the rcsetup module
- #1650: _png.read_png crashes on Python 3 with urllib.request object
- #1568: removed deprecated methods from the axes module.
- #1571: Y-labels shifted
- #1589: Fix shifted ylabels (Issue #1571)
- #1276: Fix overwriting suptitle
- #1661: Fix travis install failure on py31
- #1634: add scatterpoints to rcParam
- #1654: added explicit ‘zorder’ kwarg to Collection and LineCollection.
- #570: mplot3d reverse axis behavior
- #1653: Fix #570 - Reversing a 3d axis should now work properly.
- #1651: WebAgg: pylab compatibility
- #1638: web_backend is not installed
- #1505: Issue 1504: changed how draw handles alpha in markerfacecolor
- #1655: add get_segments method to collections.LineCollection
- #1649: add get_segments method to collections.LineCollection
- #1593: NameError: global name ‘iterable’ is not defined
- #1652: Ignore kdevelop4 project files
- #665: Mac OSX backend keyboard focus stays in terminal
- #1613: Using a stricter check to see if Python was installed as a framework.
- #1581: Provide an alternative to lena.png for two examples that use it.
- #1599: Ada Lovelace and Grace Murray Hopper images in place of Lena
- #1582: Linear tri interpolator
- #1637: change cbook to relative import
- #1645: add get_segments method to collections.LineCollection - updated
- #1639: Rename web_static to web_backend in setup.py
- #1618: Mplot3d/crashfixes
- #1636: hexbin log scale is broken in matplotlib 1.2.0
- #1624: implemented inverse transform for Mollweide axes
- #1630: A disconnected callback cannot be reconnected
- #1139: Make Axes.stem take at least one argument.
- #1426: WebAgg backend
• #1606: Document the C/C++ code guidelines
• #1622: zorder is not respected by all parts of errorbar
• #1628: Fix errorbar zorder v1.2
• #1625: saving pgf to a stream is not supported
• #1588: Annotations appear in incorrect locations
• #1620: Fix bug in _AnnotationBase
• #1621: Package for python 3.3 on OS X
• #1616: Legend: Also calc the bbox of the legend when the frame is not drawn.
• #1587: Mac OS X 10.5 needs an autoreleasepool here to avoid memory leaks. Newer...
• #1597: new MatplotlibDeprecationWarning class (against master)
• #1596: new MatplotlibDeprecationWarning class (against 1.2.x)
• #1532: CXX/Python2/cxx_extensions.cxx:1320: Assertion ‘ob_refcnt == 0’
• #1601: invalid/misconfigured fonts cause the font manager to fail
• #1604: Make font_manager ignore KeyErrors for bad fonts
• #1605: Change printed -> pretty-printed
• #1553: invert_xaxis() accidentally disables autoscaling
• #1557: inverting an axis shouldn’t affect the autoscaling setting
• #1603: ylim=0.0 is not well handled in polar plots
• #1583: Crash with text.usetex=True and plt.annotate
• #1584: triplot(x, y, simplex) should not modify the simplex array as a side effect.
• #1576: BUG: tri: prevent Triangulation from modifying specified input
• #1602: Fixed typos in docs (squashed version of #1600)
• #1600: Fixed typos in matplotlibrc and docs
• #1592: Fix a syntax error in examples (movie_demo.py)
• #1572: axes_grid demo broken
• #201: Drawing rubberband box outside of view crash backend_macosx
• #1038: osx backend does not allow font changes
• #1590: Positional argument specifiers are required by Python 2.6
• #1579: Updated custom_projection_example.py to work with v1.2 and newer
• #1578: Fixed blitting in Gtk3Agg backend
• #1580: lena.png is indecent and needs to be removed
• #1573: fix issue #1572 caused by PR #1081
• #1562: Mac OS X Backend: Removing clip that is no longer needed
• #1506: DOC: make example cursor show up in the docs
• #1565: new MatplotlibDeprecationWarning class
• #776: ticks based on number of subplots
• #1462: use plt.subplots() in examples as much as possible
• #1407: Sankey5
• #1574: Improvements to Sankey class
• #1536: ENH: add AVConv movie writer for animations
• #1570: PEP8 fixes on the tests of the dates module
• #1465: Undefined elements in axes module
• #1569: FIX Removes code that does work from the axes module
• #1250: Fix Travis tests
• #1566: pylab overwrites user variable(s)
• #1531: fix rendering slowdown with big invisible lines (issue #1256)
• #1398: PEP8 fixes on dates.py
• #1564: PEP8-compliance on axes.py (patch 4 / 4)
• #1559: Workaround for QT cursor bug in dock areas
• #1552: Remove python 2.5 stuff from texmanager.py
• #1560: Remove python2.5 support from texmanager.py
• #1555: Geo projections getting clobbered by 2to3 when used when python3
• #997: Delaunay interpolator: support grid whose width or height is 1
• #1477: alternate fix for issue #997
• #1556: Invert axis autoscale fix
• #1554: Geo projections getting clobbered by 2to3 when used when python3
• #1522: PEP8-compliance on axes.py (patch 3 / 4)
• #1548: Broken i386 + Python 3 build
• #1550: PEP8 fixes on the module texmanager
• #783: mplot3d: scatter (and others) incorrectly auto-scale axes after set_[xyz]lim()
• #1289: Autoscaling and limits in mplot3d.
• #1551: PEP8 fixes on the spines module
• #1537: Fix savefig.extension == “auto”
• #1297: pyplot.plotfile. gridon option added with default from rcParam.
• #1526: Remove unnecessary clip cairo
• #1538: Remove unnecessary clip from Cairo backend; squashed commit
• #1544: str.format() doesn’t work on python 2.6
• #1549: Add citation page to website
• #1514: Fix streamplot when color argument has NaNs
• #1487: MaxNLocator for log-scale
• #1081: Propagate mpl.text.Text instances to the backend and fix documentation
• #1533: ENH: raise a more informative error
• #955: Strange resize behaviour with ImageGrid
• #1003: Fix for issue #955
• #1546: Quiver crashes if given matrices
• #1542: Wrong __version__numpy__
• #1540: Changed mailinglist archive link.
• #1507: python setup.py build (in parallel)
• #1492: MacOSX backend blocks in IPython QtConsole
• #1493: check ret == False in Timer.on_timer
• #1523: DOC: github ribbon does not cover up index link
• #1515: set_cmap should not require an active image
• #1500: comment on http://matplotlib.org/users/pgf.html#pgf-tutorial - minor issue with xits font
• #1489: Documentation update for specgram
• #1527: fix 2 html color names
• #1524: Make README.txt consistent reStructuredText
• #1525: pgf: documentation enhancements
• #1510: pgf: documentation enhancements
• #1512: Reorganize the developer docs
• #1518: PEP8 compliance on the delaunay module
• #1357: PEP8 fixes on text.py
• #1469: PEP8-compliance on axes.py (patch 2 / 4)
• #1470: Add test and test-coverage to Makefile
• #1513: Problems with image sizes
• #1509: pgf: draw_image() doesn’t store path to png files in the pgf source
• #1516: set_xticklabels changes font when text.usetex is enabled
- #1442: Add savefig_kwarg to Animation.save() method
- #1511: Reorganize developer docs
- #1503: DOC: ‘inout’ option for tick_params direction
- #1494: Added sphinx documentation for Triangulation
- #1480: Remove dead code in patches
- #1496: Correct scatter docstring
- #1495: scatter docstring, minor
- #1472: FIX extra comma in Sankey.add
- #1471: Improved checking logic of _check_xyz in contour.py
- #998: fix for issue #997
- #1479: Reintroduce examples.directory rc parameter
- #1491: Reintroduce examples.directory rc parameter
- #1405: Add angle kwarg to patches.Rectangle
- #1278: Make arrow docstring mention data transform
- #1475: make plt.subplot() act as plt.subplot(111)
- #1355: Add sym-log normalization.
- #1474: use an imagemap for the “fork me on github” ribbon
- #632: ENH: More included norms, especially a symlog like norm
- #1466: Too many open files
- #1485: Fix leak of gc’s in gtkagg backend
- #1484: V1.2.x Fix leak of gc’s in gtkagg backend.
- #1374: PEP8 fixes on widgets.py
- #1379: PEP8 fixes on quiver.py
- #1399: PEP8 fixes on patches
- #1478: Reintroduce examples.directory rcParams to customize cbook.get_sample_data() lookup location
- #1468: use an imagemap for the “fork me on github” ribbon
- #1395: PEP8 fixes on contour.py
- #1473: offsets.shape(-1,2)
- #1467: matplotlib 1.2.0 Binary installer for 32-bit Windows for python 2.7 is missing
- #1419: bbox_extra_artists doesn’t work for a table
- #1432: lengend overlaps graph
• #1464: PEP8-compliance on axes.py (patch 1/4)
• #1400: PEP8 fixes on offsetbox.py
• #1463: Document the Gtk3 backends
• #1417: Pep8 on the axes module
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 (eg LGPL) are acceptable in matplotlib toolkits. For a discussion of the motivations behind the licencing choice, see Licenses.

21.1 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.

21.2 License agreement for matplotlib 1.3.0

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.

   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 1.3.0 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, Release 1.3.0

Matplotlib Development Team; All Rights Reserved” are retained in matplotlib 1.3.0 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 1.3.0 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 1.3.0.

4. MDT is making matplotlib 1.3.0 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 1.3.0 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. MDT SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF MATPLOTLIB 1.3.0 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING MATPLOTLIB 1.3.0, 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 1.3.0, Licensee agrees to be bound by the terms and conditions of this License Agreement.

21.3 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 1.3.0 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 1.3.0 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 1.3.0 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 1.3.0.

4. JDH is making matplotlib 1.3.0 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 1.3.0 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. JDH SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF MATPLOTLIB 1.3.0 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING MATPLOTLIB 1.3.0, 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 1.3.0, Licensee agrees to be bound by the terms and conditions of this License Agreement.
matplotlib was written by John Hunter and is now developed and maintained by a number of active developers. The current lead developer of matplotlib is Michael Droettboom.

Special thanks to those who have made valuable contributions (roughly in order of first contribution by date). Any list like this is bound to be incomplete and can’t capture the thousands and thousands of contributions over the years from these and others:

Jeremy O’Donoghue wrote the wx backend

Andrew Straw provided much of the log scaling architecture, the fill command, PIL support for imshow, and provided many examples. He also wrote the support for dropped axis spines and the buildbot unit testing infrastructure which triggers the JPL/James Evans platform specific builds and regression test image comparisons from svn matplotlib across platforms on svn commits.

Charles Twardy provided the impetus code for the legend class and has made countless bug reports and suggestions for improvement.

Gary Ruben made many enhancements to errorbar to support x and y errorbar plots, and added a number of new marker types to plot.

John Gill wrote the table class and examples, helped with support for auto-legend placement, and added support for legending scatter plots.

David Moore wrote the paint backend (no longer used)

Todd Miller supported by STSCI contributed the TkAgg backend and the numerix module, which allows matplotlib to work with either numeric or numarray. He also ported image support to the postscript backend, with much pain and suffering.

Paul Barrett supported by STSCI overhauled font management to provide an improved, free-standing, platform independent font manager with a WC3 compliant font finder and cache mechanism and ported truetype and mathtext to PS.

Perry Greenfield supported by STSCI overhauled and modernized the goals and priorities page, implemented an improved colormap framework, and has provided many suggestions and a lot of insight to the overall design and organization of matplotlib.

Jared Wahlstrand wrote the initial SVG backend.

Steve Chaplin served as the GTK maintainer and wrote the Cairo and GTKCairo backends.

Jim Benson provided the patch to handle vertical mathtext.
Gregory Lielens  provided the FltkAgg backend and several patches for the frontend, including contributions to toolbar2, and support for log ticking with alternate bases and major and minor log ticking.

Darren Dale
did the work to do mathtext exponential labeling for log plots, added improved support for scalar formatting, and did the lions share of the psfrag LaTeX support for postscript. He has made substantial contributions to extending and maintaining the PS and Qt backends, and wrote the site.cfg and matplotlib.conf build and runtime configuration support. He setup the infrastructure for the sphinx documentation that powers the mpl docs.

Paul Mcguire  provided the pyparsing module on which mathtext relies, and made a number of optimizations to the matplotlib mathtext grammar.

Fernando Perez  has provided numerous bug reports and patches for cleaning up backend imports and expanding pylab functionality, and provided matplotlab support in the pylab mode for ipython. He also provided the matshow() command, and wrote TConfig, which is the basis for the experimental traited mpl configuration.

Andrew Dalke  of Dalke Scientific Software contributed the strftime formatting code to handle years earlier than 1900.

Jochen Voss  served as PS backend maintainer and has contributed several bugfixes.

Nadia Dencheva
supported by STSCI provided the contouring and contour labeling code.

Baptiste Carvello  provided the key ideas in a patch for proper shared axes support that underlies ganged plots and multiscale plots.

Jeffrey Whitaker  at NOAA wrote the Basemap (Not distributed with matplotlib) toolkit

Sigve Tjoraand, Ted Drain, James Evans  and colleagues at the JPL collaborated on the QtAgg backend and sponsored development of a number of features including custom unit types, datetime support, scale free ellipses, broken bar plots and more. The JPL team wrote the unit testing image comparison infrastructure for regression test image comparisons.

James Amundson  did the initial work porting the qt backend to qt4

Eric Firing  has contributed significantly to contouring, masked array, pcolor, image and quiver support, in addition to ongoing support and enhancements in performance, design and code quality in most aspects of matplotlib.

Daishi Harada  added support for “Dashed Text”. See dashpointlabel.py and TextWithDash.

Nicolas Young  added support for byte images to imshow, which are more efficient in CPU and memory, and added support for irregularly sampled images.

The brainvisa Orsay team and Fernando Perez  added Qt support to ipython in pylab mode.

Charlie Moad  contributed work to matplotlib’s Cocoa support and has done a lot of work on the OSX and win32 binary releases.

Jouni K. Seppänen  wrote the PDF backend and contributed numerous fixes to the code, to tex support and to the get_sample_data handler
Paul Kienzle  improved the picking infrastructure for interactive plots, and with Alex Mont contributed fast rendering code for quadrilateral meshes.

Michael Droettboom  supported by STSCI wrote the enhanced mathtext support, implementing Knuth’s box layout algorithms, saving to file-like objects across backends, and is responsible for numerous bug-fixes, much better font and unicode support, and feature and performance enhancements across the matplotlib code base. He also rewrote the transformation infrastructure to support custom projections and scales.

John Porter, Jonathon Taylor and Reinier Heeres  John Porter wrote the mplot3d module for basic 3D plotting in matplotlib, and Jonathon Taylor and Reinier Heeres ported it to the refactored transform trunk.

Jae-Joon Lee  Implemented fancy arrows and boxes, rewrote the legend support to handle multiple columns and fancy text boxes, wrote the axes grid toolkit, and has made numerous contributions to the code and documentation

Paul Ivanov  Has worked on getting matplotlib integrated better with other tools, such as Sage and IPython, and getting the test infrastructure faster, lighter and meaner. Listen to his podcast.

Tony Yu  Has been involved in matplotlib since the early days, and recently has contributed stream plotting among many other improvements. He is the author of mpltools.

Michiel de Hoon  Wrote and maintains the macosx backend.

Ian Thomas  Contributed, among other things, the triangulation (tricolor and tripcontour) methods.

Benjamin Root  Has significantly improved the capabilities of the 3D plotting. He has improved matplotlib’s documentation and code quality throughout, and does invaluable triaging of pull requests and bugs.

Phil Elson  Fixed some deep-seated bugs in the transforms framework, and has been laser-focused on improving polish throughout matplotlib, tackling things that have been considered to large and daunting for a long time.

Damon McDougall  Added triangulated 3D surfaces and stack plots to matplotlib.
Part II

The Matplotlib FAQ
23.1 Report a compilation problem

See *Getting help*.

23.2 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 cd into a directory containing a minimal example in a file. Something like simple_plot.py for example:
from pylab import *
plot([1,2,3])
show()

and run it with:

python simple_plot.py --verbose-helpful

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.

### 23.3 How to completely remove matplotlib

Occasionally, problems with matplotlib can be solved with a clean installation of the package.

The process for removing an installation of matplotlib depends on how matplotlib was originally installed on your system. Follow the steps below that goes with your original installation method to cleanly remove matplotlib from your system.

#### 23.3.1 Easy Install

1. Delete the caches from your .matplotlib configuration directory.
2. Run:

   easy_install -m matplotlib

3. Delete any .egg files or directories from your installation directory.

#### 23.3.2 Windows installer

1. Delete the caches from your .matplotlib configuration directory.
2. Use Start → Control Panel to start the Add and Remove Software utility.

#### 23.3.3 Source install

Unfortunately:

python setup.py clean

does not properly clean the build directory, and does nothing to the install directory. To cleanly rebuild:

1. Delete the caches from your .matplotlib configuration directory.
2. Delete the build directory in the source tree.
3. Delete any matplotlib directories or eggs from your *installation directory*.

### 23.4 How to Install

#### 23.4.1 Source install from git

Clone the main source using one of:

```bash
git clone git@github.com:matplotlib/matplotlib.git
```

or:

```bash
git clone git://github.com/matplotlib/matplotlib.git
```

and build and install as usual with:

```bash
> cd matplotlib
> python setup.py install
```

**Note:** If you are on debian/ubuntu, you can get all the dependencies required to build matplotlib with:

```bash
sudo apt-get build-dep python-matplotlib
```

If you are on Fedora/RedHat, you can get all the dependencies required to build matplotlib by first installing `yum-builddep` and then running:

```bash
su -c "yum-builddep python-matplotlib"
```

This does not build matplotlib, but it does get all of the build dependencies, which will make building from source easier.

If you want to be able to follow the development branch as it changes just replace the last step with (make sure you have `setuptools` installed):

```bash
> python setup.py develop
```

This creates links in the right places and installs the command line script to the appropriate places.

**Note:** Mac OSX users please see the *Building on OSX* guide.

Then, if you want to update your matplotlib at any time, just do:

```bash
> 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 the `python setupegg.py develop` command again to compile them.

There is more information on *using git* in the developer docs.
23.5 Linux Notes

Because most Linux distributions use some sort of package manager, we do not provide a pre-built binary for the Linux platform. Instead, we recommend that you use the “Add Software” method for your system to install matplotlib. This will guarantee that everything that is needed for matplotlib will be installed as well.

If, for some reason, you cannot use the package manager, Linux usually comes with at least a basic build system. Follow the *instructions* found above for how to build and install matplotlib.

23.6 OS-X Notes

23.6.1 Which python for OS X?

Apple ships with its own python, and many users have had trouble with it. There are several alternative versions of python that can be used. If it is feasible, we recommend that you use the enthought python distribution EPD for OS X (which comes with matplotlib and much more). Also available is MacPython or the official OS X version from python.org.

*Note:* Before installing any of the binary packages, be sure that all of the packages were compiled for the same version of python. Often, the download site for NumPy and matplotlib will display a supposed ‘current’ version of the package, but you may need to choose a different package from the full list that was built for your combination of python and OSX.

23.6.2 Installing OSX binaries

If you want to install matplotlib from one of the binary installers we build, you have two choices: a mpkg installer, which is a typical Installer.app, or a binary OSX egg, which you can install via setuptools’ easy_install.

The mkpg installer will have a “zip” extension, and will have a name like matplotlib-1.2.0-py2.7-macosx10.5.mpkg.zip. The name of the installer depends on which versions of python, matplotlib, and OSX it was built for. You need to unzip this file using either the “unzip” command, or simply double clicking on the it. Then when you double-click on the resulting mpkd, which will have a name like matplotlib-1.2.0-py2.7-macosx10.5.mpkg, it will run the Installer.app, prompt you for a password if you need system-wide installation privileges, and install to a directory like /Library/Python/2.7/site-packages/ (exact path depends on your python version). This directory may not be in your python ‘path’ variable, so you should test your installation with:

```
> python -c 'import matplotlib; print matplotlib.__version__, matplotlib.__file__'
```

If you get an error like:

```
Traceback (most recent call last):
  File "<string>", line 1, in <module>
ImportError: No module named matplotlib
```

then you will need to set your PYTHONPATH, eg: 

```bash
> export PYTHONPATH=/Library/Python/2.7/site-packages:
```
export PYTHONPATH=/Library/Python/2.7/site-packages:$PYTHONPATH

See also ref: environment-variables.

### 23.6.3 Building and installing from source on OSX with EPD

If you have the EPD installed (*Which python for OS X?*), it might turn out to be rather tricky to install a new version of matplotlib from source on the Mac OS 10.5. Here’s a procedure that seems to work, at least sometimes:

0. Remove the ~/.matplotlib folder (“rm -rf ~/.matplotlib”).

1. Edit the file (make a backup before you start, just in case):
   /Library/Frameworks/Python.framework/Versions/Current/lib/python2.5/config/Makefile, removing all occurrences of the string -arch ppc, changing the line MACOSX_DEPLOYMENT_TARGET=10.3 to MACOSX_DEPLOYMENT_TARGET=10.5 and changing the occurrences of MacOSX10.4u.sdk into MacOSX10.5.sdk

2. In /Library/Frameworks/Python.framework/Versions/Current/lib/pythonX.Y/site-packages/easy-install.pth (where X.Y is the version of Python you are building against) Comment out the line containing the name of the directory in which the previous version of MPL was installed (Looks something like ./matplotlib-0.98.5.2n2-py2.5-macosx-10.3-fat.egg).

3. Save the following as a shell script, for example ./install-matplotlib-epd-osx.sh:

```bash
NAME=matplotlib
VERSION=v1.1.x
PREFIX=$HOME
#branch="release"
branch="master"
git clone git://github.com/matplotlib/matplotlib.git
cd matplotlib
if [ $branch = "release" ]
  then
echo getting the maintenance branch
git checkout -b $VERSION origin/$VERSION
fi
export CFLAGS="-Os -arch i386"
export LDFLAGS="-Os -arch i386"
export PKG_CONFIG_PATH="/usr/x11/lib/pkgconfig"
export ARCHFLAGS="-arch i386"
python setup.py build
# use --prefix if you don't want it installed in the default location:
python setup.py install #--prefix=$PREFIX
cd ..
```

Run this script (for example sh ./install-matplotlib-epd-osx.sh) in the directory in which you want the source code to be placed, or simply type the commands in the terminal command line. This script sets some local variable (CFLAGS, LDFLAGS, PKG_CONFIG_PATH, ARCHFLAGS), removes previous installations, checks out the source from github, builds and installs it. The backend should to be set to MacOSX.
23.7 Windows Notes

23.7.1 Binary installers for Windows

If you have already installed python, you can use one of the matplotlib binary installers for windows – you can get these from the download site. Choose the files that match your version of python (eg py2.7 if you installed Python 2.7) which have the .exe extension. If you haven’t already installed python, you can get the official version from the python web site.

There are also two packaged distributions of python that come preloaded with matplotlib and many other tools like ipython, numpy, scipy, vtk and user interface toolkits. These packages are quite large because they come with so much, but you get everything with a single click installer.

- The Enthought Python Distribution EPD
- python (x, y)
24.1 General Concepts

matplotlib has an extensive codebase that can be daunting to many new users. However, most of matplotlib can be understood with a fairly simple conceptual framework and knowledge of a few important points.

Plotting requires action on a range of levels, from the most general (e.g., ‘contour this 2-D array’) to the most specific (e.g., ‘color this screen pixel red’). The purpose of a plotting package is to assist you in visualizing your data as easily as possible, with all the necessary control – that is, by using relatively high-level commands most of the time, and still have the ability to use the low-level commands when needed.

Therefore, everything in matplotlib is organized in a hierarchy. At the top of the hierarchy is the matplotlib “state-machine environment” which is provided by the matplotlib.pyplot module. At this level, simple functions are used to add plot elements (lines, images, text, etc.) to the current axes in the current figure.

Note: Pyplot’s state-machine environment behaves similarly to MATLAB and should be most familiar to users with MATLAB experience.

The next level down in the hierarchy is the first level of the object-oriented interface, in which pyplot is used only for a few functions such as figure creation, and the user explicitly creates and keeps track of the figure and axes objects. At this level, the user uses pyplot to create figures, and through those figures, one or more axes objects can be created. These axes objects are then used for most plotting actions.
For even more control – which is essential for things like embedding matplotlib plots in GUI applications – the pyplot level may be dropped completely, leaving a purely object-oriented approach.

### 24.2 Matplotlib, pylab, and pyplot: how are they related?

Matplotlib is the whole package; `pylab` is a module in matplotlib that gets installed alongside `matplotlib`; and `matplotlib.pyplot` is a module in matplotlib.

Pyplot provides the state-machine interface to the underlying plotting library in matplotlib. This means that figures and axes are implicitly and automatically created to achieve the desired plot. For example, calling `plot` from pyplot will automatically create the necessary figure and axes to achieve the desired plot. Setting a title will then automatically set that title to the current axes object:

```python
import matplotlib.pyplot as plt
plt.plot(range(10), range(10))
plt.title("Simple Plot")
plt.show()
```

Pylab combines the pyplot functionality (for plotting) with the numpy functionality (for mathematics and for working with arrays) in a single namespace, making that namespace (or environment) even more MATLAB-like. For example, one can call the `sin` and `cos` functions just like you could in MATLAB, as well as having all the features of pyplot.

The pyplot interface is generally preferred for non-interactive plotting (i.e., scripting). The pylab interface is convenient for interactive calculations and plotting, as it minimizes typing. Note that this is what you get if you use the `ipython` shell with the `-pylab` option, which imports everything from pylab and makes plotting fully interactive.

### 24.3 Coding Styles

When viewing this documentation and examples, you will find different coding styles and usage patterns. These styles are perfectly valid and have their pros and cons. Just about all of the examples can be converted into another style and achieve the same results. The only caveat is to avoid mixing the coding styles for your own code.

---

**Note:** Developers for matplotlib have to follow a specific style and guidelines. See *The Matplotlib Developers’ Guide*.

---

Of the different styles, there are two that are officially supported. Therefore, these are the preferred ways to use matplotlib.

For the preferred pyplot style, the imports at the top of your scripts will typically be:

```python
import matplotlib.pyplot as plt
import numpy as np
```
Then one calls, for example, np.arange, np.zeros, np.pi, plt.figure, plt.plot, plt.show, etc. So, a simple example in this style would be:

```python
import matplotlib.pyplot as plt
import numpy as np
x = np.arange(0, 10, 0.2)
y = np.sin(x)
plt.plot(x, y)
plt.show()
```

Note that this example used pyplot’s state-machine to automatically and implicitly create a figure and an axes. For full control of your plots and more advanced usage, use the pyplot interface for creating figures, and then use the object methods for the rest:

```python
import matplotlib.pyplot as plt
import numpy as np
x = np.arange(0, 10, 0.2)
y = np.sin(x)
fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(x, y)
plt.show()
```

Next, the same example using a pure MATLAB-style:

```python
from pylab import *
x = arange(0, 10, 0.2)
y = sin(x)
plot(x, y)
show()
```

So, why all the extra typing as one moves away from the pure MATLAB-style? For very simple things like this example, the only advantage is academic: the wordier styles are more explicit, more clear as to where things come from and what is going on. For more complicated applications, this explicitness and clarity becomes increasingly valuable, and the richer and more complete object-oriented interface will likely make the program easier to write and maintain.

### 24.4 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 embed matplotlib into graphical user interfaces like wxpython or pygtk to build rich applications. Others use matplotlib in batch scripts to generate postscript images from some numerical simulations, and still others in 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, ie 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”).

There are a two primary ways to configure your backend. One is to set the `backend` parameter in your `matplotlibrc` file (see Customizing matplotlib):

```python
backend : WXAgg # use wxpython with antigrain (agg) rendering
```

The other is to use the matplotlib `use()` directive:

```python
import matplotlib
matplotlib.use('PS')  # generate postscript output by default
```

If you use the `use` directive, this must be done before importing `matplotlib.pyplot` or `matplotlib.pylab`.

**Note:** Backend name specifications are not case-sensitive; e.g., ‘GTKAgg’ and ‘gtkagg’ are equivalent.

With a typical installation of matplotlib, such as from a binary installer or a Linux distribution package, a good default backend will already be set, allowing 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 use either of the two methods given above.

If, however, you want to write graphical user interfaces, or a web application server (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. All of the user interfaces except macosx can be used with agg rendering, eg WXAgg, GTKAgg, QT4Agg, TkAgg. In addition, some of the user interfaces support other rendering engines. For example, with GTK, you can also select GDK rendering (backend GTK) or Cairo rendering (backend GTKcairo).

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 backed for each; these are non-interactive backends, capable of writing to a file):

<table>
<thead>
<tr>
<th>Renderer</th>
<th>Filetypes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A&lt;sup&gt;G&lt;/sup&gt;</td>
<td>png</td>
<td>raster graphics – high quality images using the Anti-Grain Geometry engine</td>
</tr>
<tr>
<td>PS</td>
<td>ps eps</td>
<td>vector graphics – Postscript output</td>
</tr>
<tr>
<td>PDF</td>
<td>pdf</td>
<td>vector graphics – Portable Document Format</td>
</tr>
<tr>
<td>SVG</td>
<td>svg</td>
<td>vector graphics – Scalable Vector Graphics</td>
</tr>
<tr>
<td>Cairo</td>
<td>png ps pdf svg ...</td>
<td>vector graphics – Cairo graphics</td>
</tr>
<tr>
<td>GDK</td>
<td>png jpg tiff ...</td>
<td>raster graphics – the Gimp Drawing Kit</td>
</tr>
</tbody>
</table>

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:
Matplotlib, Release 1.3.0

<table>
<thead>
<tr>
<th>Backend</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GTK-Agg</td>
<td>Agg rendering to a GTK 2.x canvas (requires PyGTK)</td>
</tr>
<tr>
<td>GTK3Agg</td>
<td>Agg rendering to a GTK 3.x canvas (requires PyGObject)</td>
</tr>
<tr>
<td>GTK</td>
<td>GDK rendering to a GTK 2.x canvas (not recommended) (requires PyGTK)</td>
</tr>
<tr>
<td>GTK-Cairo</td>
<td>Cairo rendering to a GTK 2.x canvas (requires PyGTK and pycairo)</td>
</tr>
<tr>
<td>GTK3Cairo</td>
<td>Cairo rendering to a GTK 3.x canvas (requires PyGObject and pycairo)</td>
</tr>
<tr>
<td>WXAgg</td>
<td>Agg rendering to a wxWidgets canvas (requires wxPython)</td>
</tr>
<tr>
<td>WX</td>
<td>Native wxWidgets drawing to a wxWidgets Canvas (not recommended) (requires wxPython)</td>
</tr>
<tr>
<td>TkAgg</td>
<td>Agg rendering to a Tk canvas (requires TkInter)</td>
</tr>
<tr>
<td>QtAgg</td>
<td>Agg rendering to a Qt canvas (requires PyQt4)</td>
</tr>
<tr>
<td>macosx</td>
<td>Cocoa rendering in OSX windows (presently lacks blocking show() behavior when matplotlib is in non-interactive mode)</td>
</tr>
</tbody>
</table>

### 24.5 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). 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.

#### 24.5.1 Interactive example

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

```python
import matplotlib.pyplot as plt
plt.ion()
plt.plot([1.6, 2.7])
```
Assuming you are running version 1.0.1 or higher, and you have an interactive backend installed and selected by default, you should see a plot, and your terminal prompt should also be active; you can type additional commands such as:

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

and you will see the plot being updated after each line. This is because you are in interactive mode and you are using pyplot functions. Now try an alternative method of modifying the plot. Get a reference to the `Axes` instance, and call a method of that instance:

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

Nothing changed, because the Axes methods do not include an automatic call to `draw_if_interactive()`; that call is added by the pyplot functions. If you are using methods, then when you want to update the plot on the screen, you need to call `draw()`:

```python
plt.draw()
```

Now you should see the new line added to the plot.

### 24.5.2 Non-interactive example

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

```python
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:

```python
plt.show()
```

Now you see the plot, but your terminal command line is unresponsive; the `show()` command 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 re-drawing 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:

```python
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.

### 24.5.3 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.
Contents

- **How-To**
  - Plotting: howto
    - Find all objects in a figure of a certain type
    - Save transparent figures
    - Save multiple plots to one pdf file
    - Move the edge of an axes to make room for tick labels
    - Automatically make room for tick labels
    - Configure the tick linewidths
    - Align my ylabels across multiple subplots
    - Skip dates where there is no data
    - Test whether a point is inside a polygon
    - Control the depth of plot elements
    - Make the aspect ratio for plots equal
    - Make a movie
    - Multiple y-axis scales
    - Generate images without having a window appear
    - Use `show()`
  - Contributing: howto
    - Submit a patch
    - Contribute to matplotlib documentation
  - Matplotlib in a web application server
    - `matplotlib` with apache
    - `matplotlib` with django
    - `matplotlib` with zope
    - Clickable images for HTML
  - Search examples
  - Cite Matplotlib
25.1 Plotting: howto

25.1.1 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 (eg 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:

```python
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:

```python
import matplotlib.text as text
for o in fig.findobj(text.Text):
    o.set_fontstyle('italic')
```

25.1.2 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, eg 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.:

```python
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.:

```python
ax.plot(x, y, alpha=0.5)
ax.set_xlabel('volts', alpha=0.5)
```

25.1.3 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:

```python
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:

```python
plt.savefig(pp, format='pdf')
```

An easier way is to call `PdfPages.savefig`:

```python
pp.savefig()
```

Finally, the multipage pdf object has to be closed:

```python
pp.close()
```

### 25.1.4 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:

```python
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`. 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 blank space between subplots
- `hspace = 0.2`  the amount of height reserved for white space between subplots

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:

```python
ax = fig.add_axes([left, bottom, width, height])
```

where all values are in fractional (0 to 1) coordinates. See `pylab_examples-axes_demo` for an example of placing axes manually.

### 25.1.5 Automatically make room for tick labels

**Note:** This is now easier to handle than ever before. Calling `tight_layout()` can fix many common
Matplotlib, Release 1.3.0

layout issues. See the *Tight Layout guide.*

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, eg 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

```python
import matplotlib.pyplot as plt
import matplotlib.transforms as mtransforms

fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(range(10))
ax.set_yticks((2, 5, 7))
labels = ax.set_yticklabels(('really, really, really', 'long', 'labels'))

def on_draw(event):
    bboxes = []
    for label in labels:
        bbox = label.get_window_extent()
        # the figure transform goes from relative coords->pixels and we
        # want the inverse of that
        bboxi = bbox.inverse_transformed(fig.transFigure)
        bboxes.append(bboxi)

    # this is the bbox that bounds all the bboxes, again in relative
    # figure coords
    bbox = mtransforms.Bbox.union(bboxes)
    if fig.subplotpars.left < bbox.width:
        # we need to move it over
        fig.subplots_adjust(left=1.1*bbox.width)  # pad a little
        fig.canvas.draw()
    return False

fig.canvas.mpl_connect('draw_event', on_draw)
plt.show()
```
25.1.6 Configure the tick linewidths

In matplotlib, the ticks are markers. All Line2D objects support a line (solid, dashed, etc) and a marker (circle, square, tick). The tick linewidth is controlled by the “markeredgewidth” property:

```python
import matplotlib.pyplot as plt
fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(range(10))
for line in ax.get_xticklines() + ax.get_yticklines():
    line.set_markersize(10)
plt.show()
```

The other properties that control the tick marker, and all markers, are markerfacecolor, markeredgewidth, markersize. For more information on configuring ticks, see Axis containers and Tick containers.

25.1.7 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.

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

box = dict(facecolor='yellow', pad=5, alpha=0.2)

fig = plt.figure()
fig.subplots_adjust(left=0.2, wspace=0.6)

ax1 = fig.add_subplot(221)
ax1.plot(2000*np.random.rand(10))
ax1.set_title('ylabels not aligned')
ax1.set_ylabel('misaligned 1', bbox=box)
ax1.set_ylim(0, 2000)
ax3 = fig.add_subplot(223)
ax3.set_ylabel('misaligned 2',bbox=box)
ax3.plot(np.random.rand(10))

labelx = -0.3  # axes coords

ax2 = fig.add_subplot(222)
ax2.set_title('ylabels aligned')
ax2.plot(2000*np.random.rand(10))
ax2.set_ylabel('aligned 1', bbox=box)
ax2.yaxis.set_label_coords(labelx, 0.5)
ax2.set_ylim(0, 2000)

ax4 = fig.add_subplot(224)
ax4.plot(np.random.rand(10))
ax4.set_ylabel('aligned 2', bbox=box)
ax4.yaxis.set_label_coords(labelx, 0.5)

plt.show()
```
25.1.8 Skip dates where there is no data

When plotting time series, eg financial time series, one often wants to leave out days on which there is no data, eg weekends. By passing in dates on the x-axis, you get large horizontal gaps on periods when there is not data. The solution is to pass in some proxy x-data, eg evenly sampled indices, and then use a custom formatter to format these as dates. The example below shows how to use an ‘index formatter’ to achieve the desired plot:

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.mlab as mlab
import matplotlib.ticker as ticker

r = mlab.csv2rec('../data/aapl.csv')
r.sort()
r = r[-30:]  # get the last 30 days
N = len(r)
ind = np.arange(N)  # the evenly spaced plot indices

def format_date(x, pos=None):
    thisind = np.clip(int(x+0.5), 0, N-1)
    return r.date[thisind].strftime('%Y-%m-%d')

import matplotlib.pyplot as plt

fig, (ax1, ax2) = plt.subplots(2, 1)

ax1.plot(r.date, r['Close'])
ax1.yaxis.set_major_formatter(ticker.FuncFormatter(format_date))
ax1.set_title('ylabels not aligned')

ax2.plot(r.date, r['Close'])
ax2.yaxis.set_major_formatter(ticker.FuncFormatter(format_date))
ax2.set_title('ylabels aligned')

fig, (ax3, ax4) = plt.subplots(2, 1)

ax3.plot(r[['Close', 'Volume']].T, r['Volume'][:, None])
ax3.yaxis.set_major_formatter(ticker.FuncFormatter(format_date))
ax3.set_title('misaligned 1')

ax4.plot(r[['Close', 'Volume']].T, r['Volume'][:, None])
ax4.yaxis.set_major_formatter(ticker.FuncFormatter(format_date))
ax4.set_title('aligned 1')

fig, (ax5, ax6) = plt.subplots(2, 1)

ax5.plot(r[['Close', 'Volume']].T, r['Volume'][:, None])
ax5.yaxis.set_major_formatter(ticker.FuncFormatter(format_date))
ax5.set_title('misaligned 2')

ax6.plot(r[['Close', 'Volume']].T, r['Volume'][:, None])
ax6.yaxis.set_major_formatter(ticker.FuncFormatter(format_date))
ax6.set_title('aligned 2')
```

25.1. Plotting: howto
fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(ind, r.adj_close, 'o-')
ax.xaxis.set_major_formatter(ticker.FuncFormatter(format_date))
fig.autofmt_xdate()
plt.show()

**25.1.9 Test whether a point is inside a polygon**

The *nxutils* provides two high-performance methods: for a single point use *pnpoly()* and for an array of points use *points_inside_poly()* For a discussion of the implementation see *pnpoly*.

**In [25]:** import numpy as np

**In [26]:** import matplotlib.nxutils as nx

**In [27]:** verts = np.array([[0,0], [0, 1], [1, 1], [1,0]], float)

**In [28]:** nx.pnpoly( 0.5, 0.5, verts)
**Out[28]:** 1

**In [29]:** nx.pnpoly( 0.5, 1.5, verts)
**Out[29]:** 0

**In [30]:** points = np.random.rand(10,2)*2

**In [31]:** points
**Out[31]:** array([[ 1.03597426, 0.61029911],
 [ 1.94061056, 0.65233947],
 [ 1.08593748, 1.16010789],
 [ 0.9255139 , 1.79098751],
 [ 1.54564936, 1.15604046],
 [ 1.71514397, 1.26147554],
 [ 1.19133536, 0.56787764],
 [ 0.40939549, 0.35190339],
 [ 1.8944715 , 0.61785408],
 [ 0.03128518, 0.48144145]])

**In [32]:** nx.points_inside_poly(points, verts)
**Out[32]:** array([False, False, False, False, False, False, False, True, False, True], dtype=bool)

**25.1.10 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.

### 25.1.11 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:

```python
ax = fig.add_subplot(111, aspect='equal')
```

### 25.1.12 Make a movie

If you want to take an animated plot and turn it into a movie, the best approach is to save a series of image files (eg PNG) and use an external tool to convert them to a movie. You can use `mencoder`, which is part of the `mplayer` suite for this:

```bash
#fps (frames per second) controls the play speed
mencoder 'mf://*.png' -mf type=png:fps=10 -ovc \l
   lavc -lavcopts vcodec=wmv2 -oac copy -o animation.avi
```

The swiss army knife of image tools, ImageMagick’s `convert` works for this as well.

Here is a simple example script that saves some PNGs, makes them into a movie, and then cleans up:

```python
import os, sys
import matplotlib.pyplot as plt

files = []
fig = plt.figure(figsize=(5,5))
ax = fig.add_subplot(111)
for i in range(50): # 50 frames
    ax.cla()
    ax.imshow(rand(5,5), interpolation='nearest')
    fname = '_tmp%03d.png'%i
    print 'Saving frame', fname
    fig.savefig(fname)
    files.append(fname)

print 'Making movie animation.mpg - this make take a while'
os.system("mencoder 'mf://_tmp*.png' -mf type=png:fps=10 \l
   -ovc lavc -lavcopts vcodec=wmv2 -oac copy -o animation.mpg")
```

### 25.1.13 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.

25.1.14 Generate images without having a window appear

The easiest way to do this is use a non-interactive backend (see What is a backend?) such as Agg (for PNGs), PDF, SVG or PS. In your figure-generating script, just call the `matplotlib.use()` directive before importing `pylab` or `pyplot`:

```python
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
plt.plot([1,2,3])
plt.savefig('myfig')
```

See also:

Matplotlib in a web application server for information about running matplotlib inside of a web application.
25.1.15 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 invokes the destruction of its plotting elements, you should call `savefig()` before calling `show` if you wish to save the figure as well as view it.

New in version v1.0.0: `show` now starts the GUI mainloop only if it isn’t already running. Therefore, multiple calls to `show` are now allowed.

Having `show` block further execution of the script or the python interpreter depends on whether matplotlib is set for interactive mode or not. 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).

**Note:** Support for interactive/non-interactive mode depends upon the backend. Until version 1.0.0 (and subsequent fixes for 1.0.1), the behavior of the interactive mode was not consistent across backends. As of v1.0.1, only the macosx backend differs from other backends because it does not support non-interactive mode.

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:

```python
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 Using matplotlib in a python shell), 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.

Many users are frustrated by `show` because they want it to be a blocking call that raises the figure, pauses the script until they close the figure, and then allow the script to continue running until the next figure is created and the next show is made. Something like this:
# WARNING : illustrating how NOT to use show

```python
for i in range(10):
    # make figure i
    show()
```

This is not what show does and unfortunately, because doing blocking calls across user interfaces can be tricky, is currently unsupported, though we have made significant progress towards supporting blocking events.

New in version v1.0.0: As noted earlier, this restriction has been relaxed to allow multiple calls to `show`. In most backends, you can now expect to be able to create new figures and raise them in a subsequent call to `show` after closing the figures from a previous call to `show`.

## 25.2 Contributing: howto

### 25.2.1 Submit a patch

See *Making patches* for information on how to make a patch with git.

If you are posting a patch to fix a code bug, please explain your patch in words – what was broken before and how you fixed it. Also, even if your patch is particularly simple, just a few lines or a single function replacement, we encourage people to submit git diffs against HEAD of the branch they are patching. It just makes life easier for us, since we (fortunately) get a lot of contributions, and want to receive them in a standard format. If possible, for any non-trivial change, please include a complete, free-standing example that the developers can run unmodified which shows the undesired behavior pre-patch and the desired behavior post-patch, with a clear verbal description of what to look for. A developer may have written the function you are working on years ago, and may no longer be with the project, so it is quite possible you are the world expert on the code you are patching and we want to hear as much detail as you can offer.

When emailing your patch and examples, feel free to paste any code into the text of the message, indeed we encourage it, but also attach the patches and examples since many email clients screw up the formatting of plain text, and we spend lots of needless time trying to reformat the code to make it usable.

You should check out the guide to developing matplotlib to make sure your patch abides by our coding conventions *The Matplotlib Developers’ Guide*.

### 25.2.2 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 (*Search examples*), 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. Bundle matplotlib in a py2exe app? ... 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 (or here for this document).

The sphinx website is a good resource for learning sphinx, but we have put together a cheat-sheet at Documenting matplotlib which shows you how to get started, and outlines the matplotlib conventions and extensions, eg 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 Submit a patch. Looking for something to do? Search for TODO.

25.3 Matplotlib in a web application server

Many users report initial problems trying to use matplotlib in web application servers, because by default matplotlib ships configured to work with a graphical user interface which may require an X11 connection. Since many barebones application servers do not have X11 enabled, you may get errors if you don’t configure matplotlib for use in these environments. Most importantly, you need to decide what kinds of images you want to generate (PNG, PDF, SVG) and configure the appropriate default backend. For 99% of users, this will be the Agg backend, which uses the C++ antigrain rendering engine to make nice PNGs. The Agg backend is also configured to recognize requests to generate other output formats (PDF, PS, EPS, SVG). The easiest way to configure matplotlib to use Agg is to call:

```python
# do this before importing pylab or pyplot
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
```

For more on configuring your backend, see What is a backend?.

Alternatively, you can avoid pylab/pyplot altogether, which will give you a little more control, by calling the API directly as shown in api-agg_oo.

You can either generate hardcopy on the filesystem by calling savefig:

```python
# do this before importing pylab or pyplot
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot([1,2,3])
fig.savefig('test.png')
```

or by saving to a file handle:

```python
fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot([1,2,3])
fig.savefig('test.png')
```
import sys
fig.savefig(sys.stdout)

Here is an example using the Python Imaging Library (PIL). First, the figure is saved to a StringIO object which is then fed to PIL for further processing:

import StringIO, Image
imgdata = StringIO.StringIO()
fig.savefig(imgdata, format='png')
imgdata.seek(0)  # rewind the data
im = Image.open(imgdata)

25.3.1 matplotlib with apache

TODO; see Contribute to matplotlib documentation.

25.3.2 matplotlib with django

TODO; see Contribute to matplotlib documentation.

25.3.3 matplotlib with zope

TODO; see Contribute to matplotlib documentation.

25.3.4 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 and add a SWF backend to support these kind of images. If you are interested in contributing to these efforts that would be great.

25.4 Search examples

The nearly 300 code examples-index included with the matplotlib source distribution are full-text searchable from the search page, but sometimes when you search, you get a lot of results from the The Matplotlib API or other documentation that you may not be interested in if you just want to find a complete, free-standing, working piece of example code. To facilitate example searches, we have tagged every code example page with the keyword codex for code example which shouldn’t appear anywhere else on this site except in the FAQ. So if you want to search for an example that uses an ellipse, search for codex ellipse.

25.5 Cite Matplotlib

If you want to refer to matplotlib in a publication, you can use “Matplotlib: A 2D Graphics Environment” by J. D. Hunter In Computing in Science & Engineering, Vol. 9, No. 3. (2007), pp. 90-95 (see this reference
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.
26.1 Obtaining matplotlib version

To find out your matplotlib version number, import it and print the __version__ attribute:

```python
>>> import matplotlib
>>> matplotlib.__version__
'0.98.0'
```

26.2 matplotlib install location

You can find what directory matplotlib is installed in by importing it and printing the __file__ attribute:

```python
>>> import matplotlib
>>> matplotlib.__file__
'/home/jdhunter/dev/lib64/python2.5/site-packages/matplotlib/__init__.pyc'
```

26.3 .matplotlib directory location

Each user has a matplotlib configuration directory which may contain a `.matplotlibrc` file. To locate your `.matplotlib/` directory, use matplotlib.get_configdir():

```python
>>> import matplotlib
>>> matplotlib.get_configdir()
```

>>> import matplotlib as mpl
>>> mpl.get_configdir()
'/home/darren/.matplotlib'

On unix-like systems, this directory is generally located in your HOME directory. On windows, it is in your documents and settings directory by default:

>>> import matplotlib
>>> mpl.get_configdir()
'C:\\Documents and Settings\\jhunter\\.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 OS-X.

### 26.4 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 or the matplotlib Sourceforge site, or the enthought python distribution EPD).
- any customizations to your `matplotlibrc` file (see Customizing matplotlib).
- 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.
- you can get very helpful debugging output from matplotlib by running your script with a `verbose-helpful` or `--verbose-debug` flags and posting the verbose output the lists:
  ```
  > python simple_plot.py --verbose-helpful > output.txt
  ```

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 – eg, 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.

### 26.5 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 build
    rm -rf /path/to/site-packages/matplotlib*
    git pull
    python setup.py install > build.out
    python examples/pylab_examples/simple_plot.py --verbose-debug > 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
TWENTYSEVEN

ENVIRONMENT VARIABLES

Contents

- Environment Variables
  - Setting environment variables in Linux and OS-X
    * BASH/KSH
    * CSH/TCSH
  - Setting environment variables in windows

HOME
The user's home directory. On linux, ~ is shorthand for HOME.

PATH
The list of directories searched to find executable programs

PYTHONPATH
The list of directories that is added to Python's standard search list when importing packages and modules

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/.matplotlib is used if it is writable. Otherwise, the python standard library tempfile.gettempdir() is used to find a base directory in which the matplotlib subdirectory is created.

27.1 Setting environment variables in Linux and OS-X

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. BASH seems to be the most common, but CSH is also common. You should be able to determine which by running at the command prompt:

```
echo $SHELL
```
27.1.1 BASH/KSH

To create a new environment variable:

```bash
export PYTHONPATH=~/Python
```

To prepend to an existing environment variable:

```bash
export PATH=~/bin:${PATH}
```

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:

```bash
export PATH=${PATH}:~/bin
```

To make your changes available in the future, add the commands to your `~/.bashrc` file.

27.1.2 CSH/TCSH

To create a new environment variable:

```bash
setenv PYTHONPATH ~/Python
```

To prepend to an existing environment variable:

```bash
setenv PATH ~/bin:${PATH}
```

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:

```bash
setenv PATH ${PATH}:~/bin
```

To make your changes available in the future, add the commands to your `~/.cshrc` file.

27.2 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

The Matplotlib Developers’ Guide
28.1 Pull request checklist

This checklist should be consulted when creating pull requests to make sure they are complete before merging. These are not intended to be rigidly followed—it’s just an attempt to list in one place all of the items that are necessary for a good pull request. Of course, some items will not always apply.

28.1.1 Branch selection

- In general, simple bugfixes that are unlikely to introduce new bugs of their own should be merged onto the maintenance branch. New features, or anything that changes the API, should be made against master. The rules are fuzzy here – when in doubt, try to get some consensus.
  - Once changes are merged into the maintenance branch, they should be merged into master.

28.1.2 Style

- Formatting should follow PEP8. Exceptions to these rules are acceptable if it makes the code objectively more readable.
  - You may want to consider installing automatic PEP8 checking in your editor.
- No tabs (only spaces). No trailing whitespace.
  - Configuring your editor to remove these things upon saving will save a lot of trouble.
- Import the following modules using the standard scipy conventions:

  ```python
  import numpy as np
  import numpy.ma as ma
  import matplotlib as mpl
  from matplotlib import pyplot as plt
  import matplotlib.cbook as cbook
  import matplotlib.collections as mcol
  import matplotlib.patches as mpatches
  ```

- See below for additional points about *Keyword argument processing*, if code in your pull request does that.
• Adding a new pyplot function involves generating code. See Writing a new pyplot function for more information.

28.1.3 Documentation

• Every new feature should be documented. If it’s a new module, don’t forget to add it to the API docs.
• Docstrings should be in numpydoc format. Don’t be thrown off by the fact that many of the existing docstrings are not in that format. We are working to standardize on numpydoc.
• Each high-level plotting function should have a simple example in the Example section. This should be as simple as possible to demonstrate the method. More complex examples should go in the examples tree.
• Build the docs and make sure all formatting warnings are addressed.
• See Documenting matplotlib for our documentation style guide.
• If your changes are non-trivial, please make an entry in the CHANGELOG.
• 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 doc/api/api_changes.rst.

28.1.4 Testing

Using the test framework is discussed in detail in the section Testing.
• If the PR is a bugfix, add a test that fails prior to the change and passes with the change. Include any relevant issue numbers in the docstring of the test.
• If this is a new feature, add a test that exercises as much of the new feature as possible. (The --with-coverage option may be useful here).
• Make sure the Travis tests are passing before merging.
  – The Travis tests automatically test on all of the Python versions matplotlib supports whenever a pull request is created or updated. The tox support in matplotlib may be useful for testing locally.

28.1.5 Installation

• If you have added new files or directories, or reorganized existing ones, make sure the new files included in the match patterns in MANIFEST.in, and/or in package_data in setup.py.

28.1.6 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).

• Interfacing with Python may be done either with the raw Python/C API or Cython. Use of PyCXX is discouraged for new code.

• Python/C interface code should be kept separate from the core C/C++ code. The interface code should be named FOO_wrap.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.

28.2 Style guide

28.2.1 Keyword argument processing

Matplotlib makes extensive use of **kwargs for pass-through customizations from one function to another. A typical example is in matplotlib.pylab.text(). The definition of the pylab text function is a simple pass-through to matplotlib.axes.Axes.text():

```python
# 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.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. You can pop the ones to be used locally and pass on the rest. For example, in plot(), scalex and scaley are
local arguments and the rest are passed on as **kwargs keyword arguments:

```python
# in axes.py
def plot(self, *args, **kwargs):
    scalex = kwargs.pop('scalex', True)
    scaley = kwargs.pop('scaley', True)
    if not self._hold: self.cla()
    lines = []
    for line in self._get_lines(*args, **kwargs):
        self.add_line(line)
        lines.append(line)
```

Note: there is a use case when kwargs are meant to be used locally in the function (not passed on), but you still need the **kwargs idiom. That is when you want to use *args to allow variable numbers of non-keyword args. In this case, python will not allow you to use named keyword args after the *args usage, so you will be forced to use **kwargs. An example is matplotlib.contour.ContourLabeler.clabel():

```python
# in contour.py
def clabel(self, *args, **kwargs):
    fontsize = kwargs.get('fontsize', None)
    inline = kwargs.get('inline', 1)
    self.fmt = kwargs.get('fmt', '%$1.3f$')
    colors = kwargs.get('colors', None)
    if len(args) == 0:
        levels = self.levels
        indices = range(len(self.levels))
    elif len(args) == 1:
        ...etc...
```

### 28.3 Hints

This section describes how to add certain kinds of new features to matplotlib.

#### 28.3.1 Developing a new backend

If you are working on a custom backend, the backend setting in matplotlibrc supports an external backend via the module directive. If my_backend.py is a matplotlib backend in your PYTHONPATH, you can set use it on one of several ways:

- in matplotlibrc:
  ```
  backend : module://my_backend
  ```

- with the use directive in your script:
  ```
  import matplotlib
  matplotlib.use('module://my_backend')
  ```

- from the command shell with the -d flag:
> python simple_plot.py -d module://my_backend

### 28.3.2 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 both in the examples and gallery sections 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:

```python
import matplotlib.cbook as cbook
fh = cbook.get_sample_data('mydata.dat')
```

### 28.3.3 Writing a new `pyplot` function

A large portion of the `pyplot` interface is automatically generated by the `boilerplate.py` script (in the root of the source tree). To add or remove a plotting method from `pyplot`, edit the appropriate list in `boilerplate.py` and then run the script which will update the content in `lib/matplotlib/pyplot.py`. Both the changes in `boilerplate.py` and `lib/matplotlib/pyplot.py` should be checked into the repository.
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 (eg basemap), but make sure you clearly state the licenses you are using.

29.1 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
THIRTY

WORKING WITH MATPLOTLIB SOURCE CODE

Contents:

30.1 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.

30.2 Install git

30.2.1 Overview

| Debian / Ubuntu | sudo apt-get install git-core |
| Fedora          | sudo yum install git-core    |
| Windows         | Download and install msysGit |
| OS X            | Use the git-osx-installer   |

30.2.2 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: http://book.git-scm.com/2_installing_git.html
30.3 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 git repository from github
- update local copy from time to time

30.3.1 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.

30.3.2 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.

30.4 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.

30.4.1 Making patches

Overview
# tell git who you are
```bash
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
```bash
git clone git://github.com/matplotlib/matplotlib.git
```
# make a branch for your patching
```bash
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
```bash
git add somewhere/tests/test_my_bug.py
```
# commit work in progress as you go
```bash
git commit -am 'BF - added tests for Funny bug'
```
# hack hack, hack
```bash
git commit -am 'BF - added fix for Funny bug'
```
# make the patch files
```bash
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:
   ```bash
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:
   ```bash
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:
   ```bash
git branch the-fix-im-thinking-of
git checkout the-fix-im-thinking-of
```

4. Do some edits, and commit them as you go:
   ```bash
   # 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

### 30.4.2 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:

```bash
# checkout and refresh master branch from main repo
git checkout 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.

### 30.5 Git for development

Contents:

#### 30.5.1 Making your own copy (fork) of matplotlib

You need to do this only once. The instructions here are very similar to the instructions at http://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:

   ![Fork button](image)

   ![Issues (0) Downloads (0) Wiki (1) Graphs](image)

Now, after a short pause and some ‘Hardcore forking action’, you should find yourself at the home page for your own forked copy of matplotlib.

30.5.2 Set up your fork

First you follow the instructions for Making your own copy (fork) of matplotlib.

Overview

git clone git@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 git@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 git@. The git:// URL is read only. This means we 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.

Note this command needs to be run on every clone of the repository that you make. It is not tracked in your personal repository on github.

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 git@github.com:your-user-name/matplotlib.git (fetch)
origin git@github.com:your-user-name/matplotlib.git (push)

30.5.3 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 -a
 stat = status -a
 br = branch
 wdiff = diff --color-words

[core]
You can edit this 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
```

Of course 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:

```bash
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:

```ini
[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):

```ini
[merge]
  log = true
```

Or from the command line:

```
git config --global merge.log true
```

### 30.5.4 Development workflow

You already have your own forked copy of the matplotlib repository, by following *Making your own copy (fork) of matplotlib, Set up your fork*, and you have configured `git` by following *Configure git*.

**Workflow summary**

- Keep your master branch clean of edits that have not been merged to the main matplotlib development repo. Your master then will follow the main matplotlib repository.
• Start a new feature branch for each set of edits that you do.
• If you can avoid it, try not to merge other branches into your feature branch while you are working.
• Ask for review!

This way of working really helps to keep work well organized, and in keeping history as clear as possible. See — for example — linux git workflow.

**Making a new feature branch**

```bash
git checkout -b my-new-feature master
```

This will create and immediately check out a feature branch based on master. To create a feature branch based on a maintenance branch, use:

```bash
git fetch origin

git checkout -b my-new-feature origin/v1.0.x
```

Generally, you will want to keep this also 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:

```bash
git push origin my-new-feature
```

You will need to use this exact command, rather than simply `git push` every time you want to push changes on your feature branch to your github repo. However, in git >1.7 you can set up a link by using the `--set-upstream` option:

```bash
git push --set-upstream origin my-new-feature
```

and then next time you need to push changes to your branch a simple `git push` will suffice. Note that `git push` pushes out all branches that are linked to a remote branch.

**The editing workflow**

**Overview**

```bash
# 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 my-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`).

**Asking for code review**

1. Go to your repo URL — e.g., `http://github.com/your-user-name/matplotlib`.

2. Click on the `Branch list` button:

3. Click on the `Compare` button for your feature branch — here `my-new-feature`:

4. If asked, select the `base` and `comparison` branch names you want to compare. Usually these will be `master` and `my-new-feature` (where that is your feature branch name).

5. At this point you should get a nice summary of the changes. Copy the URL for this, and post it to the `matplotlib mailing list`, asking for review. The URL will look something like: `http://github.com/your-user-name/matplotlib/compare/master...my-new-feature`.  

Chapter 30. Working with `matplotlib` source code

The generated comparison, is between your feature branch my-new-feature, and the place in master from which you branched my-new-feature. In other words, you can keep updating master without interfering with the output from the comparison. More detail? Note the three dots in the URL above (master...my-new-feature) and see dot2-dot3.

It’s a good idea to consult the Pull request checklist to make sure your pull request is ready for merging.

**Asking for your changes to be merged into the main repo**

When you are ready to ask for the merge of your code:

1. Go to the URL of your forked repo, say http://github.com/your-user-name/matplotlib.git.
2. Click on the ‘Pull request’ button:

   ![Pull Request Button](image)

   Enter a message; we suggest you select only matplotlib as the recipient. The message will go to the matplotlib mailing list. Please feel free to add others from the list as you like.

3. If the branch is to be merged into a maintenance branch on the main repo, make sure the “base branch” indicates the maintenance branch and not master. Github can not automatically determine the branch to merge into.

**Staying up to date with changes in the central repository**

This updates your working copy from the upstream matplotlib github repo.

**Overview**

```bash
# go to your master branch
git checkout master
# pull changes from github
git fetch upstream
# merge from upstream
git merge --ff-only upstream/master
```
We suggest that you do this only for your master branch, and leave your ‘feature’ branches unmerged, to keep their history as clean as possible. This makes code review easier:

```
git checkout master
```

Make sure you have done *Linking your repository to the upstream repo.*

Merge the upstream code into your current development by first pulling the upstream repo to a copy on your local machine:

```
git fetch upstream
```

then merging into your current branch:

```
git merge --ff-only upstream/master
```

The `--ff-only` option guarantees that if you have mistakenly committed code on your master branch, the merge fails at this point. If you were to merge `upstream/master` to your `master`, you would start to diverge from the upstream. If this command fails, see the section on *accidents.*

The letters ‘ff’ in `--ff-only` mean ‘fast forward’, which is a special case of merge where git can simply update your branch to point to the other branch and not do any actual merging of files. For master and other integration branches this is exactly what you want.

### Other integration branches

Some people like to keep separate local branches corresponding to the maintenance branches on github. At the time of this writing, `v1.0.x` is the active maintenance branch. If you have such a local branch, treat it just as `master`: don’t commit on it, and before starting new branches off of it, update it from upstream:

```
git checkout v1.0.x
```

```
git fetch upstream
```

```
git merge --ff-only upstream/v1.0.x
```

But you don’t necessarily have to have such a branch. Instead, if you are preparing a bugfix that applies to the maintenance branch, fetch from upstream and base your bugfix on the remote branch:

```
git fetch upstream
```

```
git checkout -b my-bug-fix upstream/v1.0.x
```

### Recovering from accidental commits on master

If you have accidentally committed changes on master and `git merge --ff-only` fails, don’t panic! First find out how much you have diverged:

```
git diff upstream/master...master
```

If you find that you want simply to get rid of the changes, reset your master branch to the upstream version:
git reset --hard upstream/master

As you might surmise from the words ‘reset’ and ‘hard’, this command actually causes your changes to the current branch to be lost, so think twice.

If, on the other hand, you find that you want to preserve the changes, create a feature branch for them:

git checkout -b my-important-changes

Now my-important-changes points to the branch that has your changes, and you can safely reset master as above — but make sure to reset the correct branch:

git checkout master
git reset --hard upstream/master

Deleting 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 test-branch. See also: http://github.com/guides/remove-a-remote-branch)

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 http://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 git@github.com:your-user-name/matplotlib.git

Remember that links starting with git@ use the ssh protocol and are read-write; links starting with git:// are read-only.

Your collaborators can then commit directly into that repo with the usual:

30.5. Git for development
git commit -am 'ENH - much better code'
git push origin master # pushes directly into your repo

Exploring 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.

30.6 git resources

30.6.1 Tutorials and summaries

- github help has an excellent series of how-to guides.
- learn.github has an excellent series of tutorials
- 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 casts — video snippets giving git how-tos.
- git magic — extended introduction with intermediate detail
- The git parable is an easy read explaining the concepts behind git.
- Our own 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
30.6.2 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.

30.6.3 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
Matplotlib has a testing infrastructure based on nose, making it easy to write new tests. The tests are in `matplotlib.tests`, and customizations to the nose testing infrastructure are in `matplotlib.testing`. (There is other old testing cruft around, please ignore it while we consolidate our testing to these locations.)

### 31.1 Requirements

The following software is required to run the tests:

- **nose**, version 1.0 or later
- **Ghostscript** (to render PDF files)
- **Inkscape** (to render SVG files)

### 31.2 Running the tests

Running the tests is simple. Make sure you have nose installed and run the script `tests.py` in the root directory of the distribution. The script can take any of the usual `nose test` arguments, such as

```
-v increase verbosity
-d detailed error messages
--with-coverage enable collecting coverage information
```

To run a single test from the command line, you can provide a dot-separated path to the module followed by the function separated by a colon, e.g., (this is assuming the test is installed):

```
python tests.py matplotlib.tests.test_simplification:test_clipping
```

An alternative implementation that does not look at command line arguments works from within Python:

```
import matplotlib
matplotlib.test()
```

Running tests by any means other than `matplotlib.test()` does not load the nose “knownfailureif” (Known failing tests) plugin, causing known-failing tests to fail for real.
31.3 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:

```python
from nose.tools import assert_equal

def test_simple():
    """
    very simple example test
    """
    assert_equal(1+1,2)
```

Nose determines which functions are tests by searching for functions beginning with “test” in their name.

If the test has side effects that need to be cleaned up, such as creating figures using the pyplot interface, use the `@cleanup` decorator:

```python
from matplotlib.testing.decorators import cleanup

def test_create_figure():
    """
    very simple example test that creates a figure using pyplot.
    """
    fig = figure()
    ...
```

31.4 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:

```python
import numpy as np
import matplotlib
from matplotlib.testing.decorators import image_comparison
import matplotlib.pyplot as plt

@image_comparison(baseline_images=['spines_axes_positions'])
def test_spines_axes_positions():
    # SF bug 2852168
    fig = plt.figure()
    x = np.linspace(0,2*np.pi,100)
    y = 2*np.sin(x)
    ax = fig.add_subplot(1,1,1)
    ax.set_title('centered spines')
    ax.plot(x,y)
    ax.spines['right'].set_position(('axes',0.1))
    ax.yaxis.set_ticks_position('right')
    ax.spines['top'].set_position(('axes',0.25))
```
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_category/spines_axes_positions.*`) to the correct subdirectory of `baseline_images` tree in the source directory (in this case `lib/matplotlib/tests/baseline_images/test_category`). Note carefully the `*` at the end: this will copy only the images we need to include in the git repository. The files ending in `_pdf.png` and `_svg.png` are converted from the pdf and svg originals on the fly and do not need to be in the repository. Put these new files under source code revision control (with `git add`). When rerunning the tests, they should now pass.

There are two optional keyword arguments to the `image_comparison` decorator:

- `extensions`: If you only wish to test some of the image formats (rather than the default png, svg and pdf formats), pass a list of the extensions to test.
- `tol`: This is the image matching tolerance, the default `1e-3`. If some variation is expected in the image between runs, this value may be adjusted.

### 31.5 Known failing tests

If you’re writing a test, you may mark it as a known failing test with the `knownfailureif()` 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:

```python
from nose.tools import assert_equal
from matplotlib.testing.decorators import knownfailureif

@knownfailureif(True)
def test_simple_fail():
    '''very simple example test that should fail'''
    assert_equal(1+1,3)
```

Note that the first argument to the `knownfailureif()` decorator is a fail condition, which can be a value such as True, False, or ‘indeterminate’, or may be a dynamically evaluated expression.

### 31.6 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`.

Let’s say you’ve added a new module named `whizbang.py` and you want to add tests for it in `matplotlib.tests.test_whizbang`. To add this module to the list of default tests, append its name to `default_test_modules` in `lib/matplotlib/__init__.py`.
31.7 Using tox

Tox is a tool for running tests against multiple Python environments, including multiple versions of Python (e.g., 2.6, 2.7, 3.2, etc.) and even different Python implementations altogether (e.g., CPython, PyPy, Jython, etc.)

Testing all 4 versions of Python (2.6, 2.7, 3.1, and 3.2) requires having four versions of Python installed on your system and on the PATH. Depending on your operating system, you may want to use your package manager (such as apt-get, yum or MacPorts) to do this, or use pythonbrew.

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 py26,py27

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.

31.8 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 http://travis-ci.org/#!your_GitHub_user_name/matplotlib – here’s an example.
32.1 Getting started

The documentation for matplotlib is generated from ReStructured Text using the Sphinx documentation generation tool. Sphinx-1.0 or later and numpydoc 0.4 or later is required.

The documentation sources are found in the doc/ directory in the trunk. To build the users guide in html format, cd into doc/ and do:

```python
make.py html
```

or:

```bash
./make.py html
```

you can also pass a latex flag to make.py to build a pdf, or pass no arguments to build everything.

The output produced by Sphinx can be configured by editing the conf.py file located in the doc/.

32.2 Organization of matplotlib’s documentation

The actual ReStructured Text files are kept in doc/users, doc/devel, doc/api and doc/faq. The main entry point is doc/index.rst, which pulls in the index.rst file for the users guide, developers guide, api reference, and faqs. The documentation suite is built as a single document in order to make the most effective use of cross referencing, we want to make navigating the Matplotlib documentation as easy as possible.

Additional files can be added to the various guides by including their base file name (the .rst extension is not necessary) in the table of contents. It is also possible to include other documents through the use of an include statement, such as:

```
.. include:: ../../TODO
```

32.2.1 docstrings

In addition to the “narrative” documentation described above, matplotlib also defines its API reference documentation in docstrings. For the most part, these are standard Python docstrings, but matplotlib also
includes some features to better support documenting getters and setters.

Matplotlib uses artist introspection of docstrings to support properties. All properties that you want to support through `setp` and `getp` should have a `set_property` and `get_property` method in the `Artist` class. Yes, this is not ideal given python properties or enthought traits, but it is a historical legacy for now. The setter methods use the docstring with the `ACCEPTS` token to indicate the type of argument the method accepts. e.g., in `matplotlib.lines.Line2D`:

```python
# in lines.py
def set linestyle(self, linestyle):
    
    Set the linestyle of the line

    ACCEPTS: ['-' | '--' | '-' | ':' | 'steps' | 'None' | ' ' | '] [']
```

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 automagically.

The functions `matplotlib.artist.kwdocd` and `matplotlib.artist.kwdoc()` to facilitate this. They combine python string interpolation in the docstring with the matplotlib artist introspection facility that underlies `setp` and `getp`. The `kwdocd` is a single dictionary that maps class name to a docstring of `kwargs`. Here is an example from `matplotlib.lines`:

```python
# in lines.py
artist.kwdocd['Line2D'] = artist.kwdoc(Line2D)
```

Then in any function accepting `Line2D` pass-through `kwargs`, e.g., `matplotlib.axes.Axes.plot()`:

```python
# in axes.py
def plot(self, *args, **kwargs):
    
    Some stuff omitted

    The kwargs are Line2D properties:
    %(Line2D)s

    kwars 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
    
    pass
plot.__doc__ = cbook.dedent(plot.__doc__) % artist.kwdocd
```

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 manual hacks in this case, violating the “single entry point” requirement above – see the `artist.kwdocd['Patch']` setting in `matplotlib.patches`.

### 32.3 Formatting

The Sphinx website contains plenty of documentation concerning ReST markup and working with Sphinx in general. Here are a few additional things to keep in mind:

- Please familiarize yourself with the Sphinx directives for inline markup. Matplotlib’s documentation makes heavy use of cross-referencing and other semantic markup. For example, when referring to external files, use the `:file:` directive.

- Function arguments and keywords 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*

  Please do not use the `default` role:

  Please do not describe ‘argument’ like this.

  nor the `literal` role:

  Please do not describe ‘‘argument’’ like this.

- Sphinx does not support tables with column- or row-spanning cells for latex output. Such tables cannot be used when documenting matplotlib.

- Mathematical expressions can be rendered as png images in html, and in the usual way by latex. For example:

  :math:`\sin(x_n^2)` yields: \( \sin(x_n^2) \), and:

  .. math::

  \int_{-\infty}^{\infty} \frac{e^{i\phi}}{1+x^2} \frac{e^{i\phi}}{1+x^2} \tag{32.1}

  yields:

  \[ \int_{-\infty}^{\infty} \frac{e^{i\phi}}{1+x^2} \frac{e^{i\phi}}{1+x^2} \]

- Interactive IPython sessions can be illustrated in the documentation using the following directive:

  .. sourcecode:: ipython

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

  which would yield:

  In [69]: lines = plot([1,2,3])
• Footnotes \footnote{1} can be added using \[\text{[#]}\], followed later by:

.. rubric:: Footnotes

.. [#]

• Use the note and warning directives, sparingly, to draw attention to important comments:

.. note::
   Here is a note

yields:

**Note:** here is a note

also:

**Warning:** here is a warning

• Use the deprecated directive when appropriate:

.. deprecated:: 0.98
   This feature is obsolete, use something else.

yields:

Deprecated since version 0.98: This feature is obsolete, use something else.

• Use the versionadded and versionchanged directives, which have similar syntax to the deprecated role:

.. versionadded:: 0.98
   The transforms have been completely revamped.

New in version 0.98: The transforms have been completely revamped.

• Use the seealso directive, for example:

.. seealso::

   Using ReST :ref:`emacs-helpers`:
      One example

   A bit about :ref:`referring-to-mpl-docs`:
      One more

yields:

**See also:**

Using ReST *Emacs helpers*: One example

A bit about *Referring to mpl documents*: One more

\footnote{1}{For example.}
• Please keep the *Glossary* in mind when writing documentation. You can create a references to a term in the glossary with the `:term:` role.

• The autodoc extension will handle index entries for the API, but additional entries in the index need to be explicitly added.

• Please limit the text width of docstrings to 70 characters.

• Keyword arguments should be described using a definition list.

**Note:** matplotlib makes extensive use of keyword arguments as pass-through arguments, there are many cases where a table is used in place of a definition list for autogenerated sections of docstrings.

### 32.4 Figures

#### 32.4.1 Dynamically generated figures

Figures can be automatically generated from scripts and included in the docs. It is not necessary to explicitly save the figure in the script, this will be done automatically at build time to ensure that the code that is included runs and produces the advertised figure.

The path should be relative to the *doc* directory. Any plots specific to the documentation should be added to the *doc/pyplots* directory and committed to git. Plots from the *examples* directory may be referenced through the symlink *mpl_examples* in the *doc* directory. e.g.:

```bash
.. plot:: mpl_examples/pylab_examples/simple_plot.py
```

The :scale: directive rescales the image to some percentage of the original size, though we don’t recommend using this in most cases since it is probably better to choose the correct figure size and dpi in mpl and let it handle the scaling.

**Plot directive documentation**

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:

   ```bash
   .. 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:

   ```bash
   .. plot:: path/to/plot.py
   
   This is the caption for the plot
   ```
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’}] Specify 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] 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.
- **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_pre_code** Code that should be executed before each plot.
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.

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 behaviour 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.

32.4.2 Static figures

Any figures that rely on optional system configurations need to be handled a little differently. These figures are not to be generated during the documentation build, in order to keep the prerequisites to the documentation effort as low as possible. Please run the doc/pyplots/make.py script when adding such figures, and commit the script and the images to git. Please also add a line to the README in doc/pyplots for any additional requirements necessary to generate a new figure. Once these steps have been taken, these figures can be included in the usual way:

.. plot:: pyplots/tex_unicode_demo.py
 :include-source:

32.4.3 Examples

The source of the files in the examples directory are automatically included in the HTML docs. An image is generated and included for all examples in the api and pylab_examples directories. To exclude the example from having an image rendered, insert the following special comment anywhere in the script:

# -*- noplot -*-
32.4.4 Animations

We have a matplotlib google/gmail account with username mplgithub which we 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:

```html
.. raw:: html
   :file: embed_video.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

```python
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.

32.5 Referring to mpl documents

In the documentation, you may want to include to a document in the matplotlib src, e.g., a license file or an image file from mpl-data, refer to it via a relative path from the document where the rst file resides, eg, in `users/navigation_toolbar.rst`, we refer to the image icons with:

```rst
.. image:: ../../lib/matplotlib/mpl-data/images/subplots.png
```

In the `users` subdirectory, if I want to refer to a file in the mpl-data directory, I use the symlink directory. For example, from `customizing.rst`:

```rst
.. literalinclude:: ../../lib/matplotlib/mpl-data/matplotlibrc
```

One exception to this is when referring to the examples dir. Relative paths are extremely confusing in the sphinx plot extensions, so without getting into the dirty details, it is easier to simply include a symlink to the files at the top doc level directory. This way, API documents like `matplotlib.pyplot.plot()` can refer to the examples in a known location.

In the top level doc directory we have symlinks pointing to the mpl examples:

```bash
home:~/mpl/doc> ls -l mpl_*
ml_examples  ->  ../examples
```

So we can include plots from the examples dir using the symlink:
We used to use a symlink for mpl-data too, but the distro becomes very large on platforms that do not support links (eg the font files are duplicated and large)

32.6 Internal section references

To maximize internal consistency in section labeling and references, use hyphen separated, descriptive labels for section references, eg:

.. _howto-webapp:

and refer to it using the standard reference syntax:

See :ref:`howto-webapp`

Keep in mind that we may want to reorganize the contents later, so let’s 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, let’s prefer hyphens to separate words.

32.7 Section names, etc

For everything but top level chapters, please use Upper lower for section titles, eg Possible Hangups rather than Possible Hangups

32.8 Inheritance diagrams

Class inheritance diagrams can be generated with the inheritance-diagram directive. To use it, you 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:
32.9 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

C-c C-r rst-shift-region-right

   Shift region to the right
DOING A MATPLOTLIB RELEASE

A guide for developers who are doing a matplotlib release.

- Edit __init__.py and bump the version number

33.1 Testing

- Run all of the regression tests by running the tests.py script at the root of the source tree.
- Run unit/memleak_hawaii3.py and make sure there are no memory leaks
- try some GUI examples, eg simple_plot.py with GTKAgg, TkAgg, etc...
- remove font cache and tex cache from .matplotlib and test with and without cache on some example script
- Optionally, make sure examples/tests/backend_driver.py runs without errors and check the output of the PNG, PDF, PS and SVG backends

33.2 Branching

Once all the tests are passing and you are ready to do a release, you need to create a release branch. These only need to be created when the second part of the version number changes:

git checkout -b v1.1.x

On the branch, do any additional testing you want to do, and then build binaries and source distributions for testing as release candidates.

For each release candidate as well as for the final release version, please git tag the commit you will use for packaging like so:

git tag -a v1.1.0rc1

The -a flag will allow you to write a message about the tag, and affiliate your name with it. A reasonable tag message would be something like v1.1.0 Release Candidate 1 (September 24, 2011). To tag a release after the fact, just track down the commit hash, and:
33.3 Packaging

- Make sure the MANIFEST.in is up to date and remove MANIFEST so it will be rebuilt by MANIFEST.in
- run git clean in the mpl git directory before building the sdist
- unpack the sdist and make sure you can build from that directory
- Use setup.cfg to set the default backends. For windows and OSX, the default backend should be TkAgg. You should also turn on or off any platform specific build options you need. Importantly, you also need to make sure that you delete the build dir after any changes to setup.cfg before rebuilding since cruft in the build dir can get carried along.
- On windows, unix2dos the rc file.
- We have a Makefile for the OS X builds in the mpl source dir release/osx, so use this to prepare the OS X releases.
- We have a Makefile for the win32 mingw builds in the mpl source dir release/win32 which you can use this to prepare the windows releases.

33.4 Posting files

Our current method is for the release manager to collect all of the binaries from the platform builders and post the files online on Sourceforge. It is also possible that those building the binaries could upload to directly to Sourceforge.

There are many ways to upload files to Sourceforge (scp, rsync, sftp, and a web interface) described in Sourceforge Release File System documentation. Below, we will use sftp.

1. Create a directory containing all of the release files and cd to it.
2. sftp to Sourceforge:
   
   sftp USERNAME@frs.sourceforge.net:/home/frs/project/matplotlib/matplotlib

3. Make a new directory for the release and move to it:

   mkdir matplotlib-1.1.0rc1
   cd matplotlib-1.1.0rc1

4. Upload all of the files in the current directory on your local machine:
If this release is a final release, the default download for the matplotlib project should also be updated. Login to Sourceforge and visit the matplotlib files page. Navigate to the tarball of the release you just updated, click on “Details” icon (it looks like a lower case i), and make it the default download for all platforms.

There is a list of direct links to downloads on matplotlib’s main website. This needs to be manually generated and updated every time new files are posted.

1. Clone the matplotlib documentation repository and cd into it:
   ```bash
   git clone git@github.com:matplotlib/matplotlib.github.com.git
cd matplotlib.github.com
   ```

2. Update the list of downloads that you want to display by editing the downloads.txt file. Generally, this should contain the last two final releases and any active release candidates.

3. Update the downloads webpage by running the update_downloads.py script. This script requires paramiko (for sftp support) and jinja2 for templating. Both of these dependencies can be installed using pip:
   ```bash
   pip install paramiko
   pip install jinja2
   ```
   Then update the download page:
   ```bash
   ./update_downloads.py
   ```
   You will be prompted for your Sourceforge username and password.

4. Commit the changes and push them up to github:
   ```bash
   git commit -m "Updating download list"
git push
   ```

### 33.5 Update PyPI

Once the tarball has been posted on Sourceforge, you can register a link to the new release on PyPI. This should only be done with final (non-release-candidate) releases, since doing so will hide any available stable releases.

You may need to set up your pypirc file as described in the [distutils register command documentation](https://docs.python.org/3/distutils/command/register.html).

Then updating the record on PyPI is as simple as:

```bash
python setup.py register
```

This will hide any previous releases automatically.
33.6 Documentation updates

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 as well as the bleeding edge release version called dev (usually based on what’s on master in the github repository, but it may also temporarily be a staging area for proposed changes). There is also a symlink directory with the name of the most recently released version that points to the root. With each new release, these directories may need to be reorganized accordingly. Any time these version directories are added or removed, the versions.html file (which contains a list of the available documentation versions for the user) must also be updated.

To make sure everyone’s hard work gets credited, regenerate the github stats. cd into the tools directory and run:

```python
python github_stats.py $TAG > ../doc/users/github_stats.rst
```

where $TAG is the tag of the last major release. This will generate stats for all work done since that release.

In the matplotlib source repository, build the documentation:

```bash
cd doc
python make.py html
python make.py latex
```

Then copy the build products into your local checkout of the matplotlib.github.com repository (assuming here to be checked out in com):

```bash
cp -r build/html/* ~/matplotlib.github.com
cp build/latex/Matplotlib.pdf ~/matplotlib.github.com
```

Then, from the matplotlib.github.com directory, commit and push the changes upstream:

```bash
git commit -m "Updating for v1.0.1"
git push upstream master
```

33.7 Announcing

Announce the release on matplotlib-announce, matplotlib-users, and matplotlib-devel. Final (non-release-candidate) versions should also be announced on python-announce. Include a summary of highlights from the CHANGELOG and/or post the whole CHANGELOG since the last release.
34.1 matplotlib.transforms

matplotlib includes a framework for arbitrary geometric transformations that is used to 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 to 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:

\[ \text{full transform} = \text{non-affine part} + \text{affine part} \]

The backends are not expected to handle non-affine transformations themselves.
class matplotlib.transforms.TransformNode(shorthand_name=None)
    Bases: object

    TransformNode is 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.

    Creates a new TransformNode.

    shorthand_name - a string representing the “name” of this transform. The name carries no sig-
    nificance other than to improve the readability of str(transform) when DEBUG=True.

    frozen()
    Returns a frozen copy of this transform node. The frozen copy will not update when its children
    change. Useful for storing a previously known state of a transform where copy.deepcopy() might normally be used.

    invalidate()
    Invalidate this TransformNode and triggers an invalidation of its ancestors. Should be called
    any time the transform changes.

    pass_through = False
    If pass_through is True, all ancestors will always be invalidated, even if ‘self’ is already invalid.

    set_children(*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.BboxBase(shorthand_name=None)
    Bases: matplotlib.transforms.TransformNode

    This is the base class of all bounding boxes, and provides read-only access to its data. A mutable
    bounding box is provided by the Bbox class.

    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.

    Creates a new TransformNode.

    shorthand_name - a string representing the “name” of this transform. The name carries no sig-
    nificance other than to improve the readability of str(transform) when DEBUG=True.

    anchored(c, container=None)
    Return a copy of the Bbox, shifted to position c within a container.

    c: 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.

    Optional argument container is the box within which the Bbox is positioned; it defaults to the
    initial Bbox.
bounds
  (property) Returns \((x_0, y_0, \text{width}, \text{height})\).

contains\((x, y)\)
  Returns \textit{True} if \((x, y)\) is a coordinate inside the bounding box or on its edge.

containsx\((x)\)
  Returns \textit{True} if \(x\) is between or equal to \(x_0\) and \(x_1\).

containsy\((y)\)
  Returns \textit{True} if \(y\) is between or equal to \(y_0\) and \(y_1\).

corners()
  Return an array of points which are the four corners of this rectangle. For example, if this \textit{Bbox} is defined by the points \((a, b)\) and \((c, d)\), \textit{corners()} returns \((a, b), (a, d), (c, b)\) and \((c, d)\).

count_contains\((\text{vertices})\)
  Count the number of vertices contained in the \textit{Bbox}.
  \textit{vertices} is a \(\text{Nx2 Numpy array}\).

count_overlaps\((\text{bboxes})\)
  Count the number of bounding boxes that overlap this one.
  \textit{bboxes} is a sequence of \textit{BboxBase} objects

expanded\((sw, sh)\)
  Return a new \textit{Bbox} which is this \textit{Bbox} expanded around its center by the given factors \(sw\) and \(sh\).

extents
  (property) Returns \((x_0, y_0, x_1, y_1)\).

frozen()
  \textit{TransformNode} is 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.

fullyContains\((x, y)\)
  Returns \textit{True} if \((x, y)\) is a coordinate inside the bounding box, but not on its edge.

fullyContainsx\((x)\)
  Returns \textit{True} if \(x\) is between but not equal to \(x_0\) and \(x_1\).

fullyContainsy\((y)\)
  Returns \textit{True} if \(y\) is between but not equal to \(y_0\) and \(y_1\).

fully_overlaps\((\text{other})\)
  Returns \textit{True} if this bounding box overlaps with the given bounding box \textit{other}, but not on its edge alone.

height
  (property) The height of the bounding box. It may be negative if \(y_1 < y_0\).
static intersection(bbox1, bbox2)
    Return the intersection of the two bboxes or None if they do not intersect.

    Implements the algorithm described at:
    http://www.tekpool.com/node/2687

intervalx
    (property) intervalx is the pair of x coordinates that define the bounding box. It is not guar-
    anteed to be sorted from left to right.

intervaly
    (property) intervaly is the pair of y coordinates that define the bounding box. It is not guar-
    anteed to be sorted from bottom to top.

inverse_transformed(transform)
    Return a new Bbox object, statically transformed by the inverse of the given transform.

is_unit()
    Returns True if the Bbox is the unit bounding box from (0, 0) to (1, 1).

max
    (property) max is the top-right corner of the bounding box.

min
    (property) min is the bottom-left corner of the bounding box.

overlaps(other)
    Returns True if this bounding box overlaps with the given bounding box other.

p0
    (property) p0 is the first pair of (x, y) coordinates that define the bounding box. It is not guaran-
    teed to be the bottom-left corner. For that, use min.

p1
    (property) p1 is the second pair of (x, y) coordinates that define the bounding box. It is not guaran-
    teed to be the top-right corner. For that, use max.

padded(p)
    Return a new Bbox that is padded on all four sides by the given value.

rotated(radians)
    Return a new bounding box that bounds a rotated version of this bounding box by the given
    radians. The new bounding box is still aligned with the axes, of course.

shrunken(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.

shrunken_to_aspect(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—that is, 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.
size
   (property) The width and height of the bounding box. May be negative, in the same way as width and height.

splitx(*args)
e.g., bbox.splitx(f1, f2, ...)
   Returns a list of new Bbox objects formed by splitting the original one with vertical lines at fractional positions f1, f2, ...

splity(*args)
e.g., bbox.splitx(f1, f2, ...)
   Returns a list of new Bbox objects formed by splitting the original one with horizontal lines at fractional positions f1, f2, ...

transformed(transform)
   Return a new Bbox object, statically transformed by the given transform.

translated(tx, ty)
   Return a copy of the Bbox, statically translated by tx and ty.

static union(bboxes)
   Return a Bbox that contains all of the given bboxes.

width
   (property) The width of the bounding box. It may be negative if x1 < x0.

x0
   (property) x0 is the first of the pair of x coordinates that define the bounding box. x0 is not guaranteed to be less than x1. If you require that, use xmin.

x1
   (property) x1 is the second of the pair of x coordinates that define the bounding box. x1 is not guaranteed to be greater than x0. If you require that, use xmax.

xmax
   (property) xmax is the right edge of the bounding box.

xmin
   (property) xmin is the left edge of the bounding box.

y0
   (property) y0 is the first of the pair of y coordinates that define the bounding box. y0 is not guaranteed to be less than y1. If you require that, use ymin.

y1
   (property) y1 is the second of the pair of y coordinates that define the bounding box. y1 is not guaranteed to be greater than y0. If you require that, use ymax.

ymax
   (property) ymax is the top edge of the bounding box.

ymin
   (property) ymin is the bottom edge of the bounding box.
class matplotlib.transforms.Bbox(points, **kwargs)
    Bases: matplotlib.transforms.BboxBase

    A mutable bounding box.

    points: a 2x2 numpy array of the form [[x0, y0], [x1, y1]]

    If you need to create a Bbox object from another form of data, consider the static methods unit(), from_bounds() and from_extents().

    static from_bounds(x0, y0, width, height)
        (staticmethod) Create a new Bbox from x0, y0, width and height.
        width and height may be negative.

    static from_extents(*args)
        (staticmethod) Create a new Bbox from left, bottom, right and top.
        The y-axis increases upwards.

    get_points()
        Get the points of the bounding box directly as a numpy array of the form: [[x0, y0], [x1, y1]].

    ignore(value)
        Set whether the existing bounds of the box should be ignored by subsequent calls to update_from_data() or update_from_data_xy().
        value:

        • When True, subsequent calls to update_from_data() will ignore the existing bounds of the Bbox.

        • When False, subsequent calls to update_from_data() will include the existing bounds of the Bbox.

    mutated()
        return whether the bbox has changed since init

    mutatedx()
        return whether the x-limits have changed since init

    mutatedy()
        return whether the y-limits have changed since init

    static null()
        (staticmethod) Create a new null Bbox from (inf, inf) to (-inf, -inf).

    set(other)
        Set this bounding box from the “frozen” bounds of another Bbox.

    set_points(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()
        (staticmethod) Create a new unit Bbox from (0, 0) to (1, 1).
update_from_data\( (x, y, \text{ignore=\text{None}}) \)
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.

\(x\): a numpy array of \(x\)-values
\(y\): a numpy array of \(y\)-values

\(\text{ignore}\):

- 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 \text{ignore}().

update_from_data_xy\( (xy, \text{ignore=\text{None}}, \text{updatex=True, updatey=True}) \)
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.

\(xy\): a numpy array of 2D points

\(\text{ignore}\):

- 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 \text{ignore}().

\text{updatex}: when True, update the x values
\text{updatey}: when True, update the y values

update_from_path\( (\text{path, ignore=\text{None}}, \text{updatex=True, updatey=True}) \)
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.

\(\text{path}\): a Path instance

\(\text{ignore}\):

- 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 \text{ignore}().

\text{updatex}: when True, update the x values
\text{updatey}: when True, update the y values

\text{class \text{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.

\(\text{bbox}\): a child Bbox
\text{transform}: a 2D Transform
get_points()
Get the points of the bounding box directly as a numpy array of the form: [[x0, y0], [x1, y1]].

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()
- is_separable
- has_inverse
- inverted() (if has_inverse is True)

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()

Creates a new TransformNode.

shorthand_name - a string representing the “name” of this transform. The name carries no significance other than to improve the readability of str(transform) when DEBUG=True.

contains_branch(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(other_transform)
Returns 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)

depth
Returns 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()
Get the affine part of this transform.

get_matrix()
Get the Affine transformation array 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()
Return the corresponding inverse transformation.
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.

x == self.inverted().transform(self.transform(x))

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(values)
Performs the transformation on the given array of values.
Accepts a numpy array of shape (N x input_dims) and returns a numpy array of shape (N x output_dims).

transform_affine(values)
Performs 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).

Accepts a numpy array of shape (N x input_dims) and returns a numpy array of shape (N x output_dims).

transform_angles(angles, pts, radians=True, pushoff=1e-05)
Performs transformation on a set of angles anchored at specific locations.
The angles must be a column vector (i.e., numpy array).
The pts must be a two-column numpy array of x,y positions (angle transforms currently only work in 2D). This array must have the same number of rows as angles.

radians indicates whether or not input angles are given in radians (True) or degrees (False; the default).
pushoff is the distance to move away from pts for determining transformed angles (see discussion of method below).

The transformed angles are returned in an array with the same size as angles.

The generic version of this method uses a very generic algorithm that transforms pts, as well as locations very close to pts, to find the angle in the transformed system.

**transform_non_affine(values)**

Performs only the non-affine part of the 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.

Accepts a numpy array of shape (N x input_dims) and returns a numpy array of shape (N x output_dims).

**transform_path(path)**

Returns a transformed path.

*path: a Path instance.*

In some cases, this transform may insert curves into the path that began as line segments.

**transform_path_affine(path)**

Returns a path, transformed only by the affine part of this transform.

*path: a Path instance.*

transform_path(path) is equivalent to transform_path_affine(transform_path_non_affine(values)).

**transform_path_non_affine(path)**

Returns a path, transformed only by the non-affine part of this transform.

*path: a Path instance.*

transform_path(path) is equivalent to transform_path_affine(transform_path_non_affine(values)).

**transform_point(point)**

A convenience function that returns the transformed copy of 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.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.

Note that 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 class:Transform instance. This child may later be replaced with set().
frozen()
Returns a frozen copy of this transform node. The frozen copy will not update when its children change. Useful for storing a previously known state of a transform where copy.deepcopy() might normally be used.

set(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.AffineBase(*args, **kwargs)
Bases: matplotlib.transforms.Transform
The base class of all affine transformations of any number of dimensions.

get_affine()
Get the affine part of this transform.

transform(values)
Performs the transformation on the given array of values.
Accepts a numpy array of shape (N x input_dims) and returns a numpy array of shape (N x output_dims).

transform_affine(values)
Performs 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).
Accepts a numpy array of shape (N x input_dims) and returns a numpy array of shape (N x output_dims).

transform_non_affine(points)
Performs only the non-affine part of the 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.
Accepts a numpy array of shape (N x input_dims) and returns a numpy array of shape (N x output_dims).

transform_path(path)
Returns a transformed path.
path: a Path instance.
In some cases, this transform may insert curves into the path that began as line segments.

transform_path_affine(path)
Returns a path, transformed only by the affine part of this transform.
path: a Path instance.
transform_path(path) is equivalent to transform_path_affine(transform_path_non_affine(values))

**transform_path_non_affine(path)**

Returns a path, transformed only by the non-affine part of this transform.

*path*: a Path instance.

transform_path(path) is equivalent to transform_path_affine(transform_path_non_affine(values)).

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.

**frozen()**

    Returns a frozen copy of this transform node. The frozen copy will not update when its children change. Useful for storing a previously known state of a transform where copy.deepcopy() might normally be used.

**inverted()**

    Return the corresponding inverse transformation.

    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.

    \[ x == \text{self.inverted().transform(self.transform(x))} \]

**static matrix_from_values(a, b, c, d, e, f)**

    (staticmethod) Create a new transformation matrix as a 3x3 numpy array of the form:

    \[
    \begin{pmatrix}
    a & c & e \\
    b & d & f \\
    0 & 0 & 1
    \end{pmatrix}
    \]

    **to_values()**

    Return the values of the matrix as a sequence (a,b,c,d,e,f)

**transform_affine(points)**

    Performs 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).

    Accepts a numpy array of shape (N x input_dims) and returns a numpy array of shape (N x output_dims).
**transform_point(point)**

A convenience function that returns the transformed copy of 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`.

```python
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.

**clear()**

Reset the underlying matrix to the identity transform.

**static from_values(a, b, c, d, e, f)**

(staticmethod) Create a new Affine2D instance from the given values:

```
   a  c  e
   b  d  f
   0  0  1
```

**get_matrix()**

Get the underlying transformation matrix as a 3x3 numpy array:

```
   a  c  e
   b  d  f
   0  0  1
```

**static identity()**

(staticmethod) 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(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(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()`.

Chapter 34. Working with transformations
rotate_deg(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(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(sx, sy=None)
Adds 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(other)
Set this transformation from the frozen copy of another Affine2DBase object.

set_matrix(mtx)
Set the underlying transformation matrix from a 3x3 numpy array:

\[
\begin{pmatrix}
a & c & e \\
b & d & f \\
0 & 0 & 1
\end{pmatrix}
\]

translate(tx, ty)
Adds 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.IdentityTransform(*args, **kwargs)
Bases: matplotlib.transforms.Affine2DBase

A special class that does on thing, the identity transform, in a fast way.

frozen()
Returns a frozen copy of this transform node. The frozen copy will not update when its children change. Useful for storing a previously known state of a transform where copy.deepcopy() might normally be used.

get_affine()
Return the corresponding inverse transformation.

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.

x == self.inverted().transform(self.transform(x))
get_matrix()
Get the Affine transformation array for the affine part of this transform.

inverted()
Return the corresponding inverse transformation.

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.

\[ x \equiv \text{self.inverted().transform(self.transform(x))} \]

transform(points)
Performs only the non-affine part of the transformation.

\[ \text{transform(values)} \text{ is always equivalent to transform_affine(transform_non_affine(values))}. \]

In non-affine transformations, this is generally equivalent to \( \text{transform(values)} \). In affine transformations, this is always a no-op.

Accepts a numpy array of shape \((N \times \text{input_dims})\) and returns a numpy array of shape \((N \times \text{output_dims})\).

transform_affine(points)
Performs only the non-affine part of the transformation.

\[ \text{transform(values)} \text{ is always equivalent to transform_affine(transform_non_affine(values))}. \]

In non-affine transformations, this is generally equivalent to \( \text{transform(values)} \). In affine transformations, this is always a no-op.

Accepts a numpy array of shape \((N \times \text{input_dims})\) and returns a numpy array of shape \((N \times \text{output_dims})\).

transform_non_affine(points)
Performs only the non-affine part of the transformation.

\[ \text{transform(values)} \text{ is always equivalent to transform_affine(transform_non_affine(values))}. \]

In non-affine transformations, this is generally equivalent to \( \text{transform(values)} \). In affine transformations, this is always a no-op.

Accepts a numpy array of shape \((N \times \text{input_dims})\) and returns a numpy array of shape \((N \times \text{output_dims})\).

transform_path(path)
Returns a path, transformed only by the non-affine part of this transform.

path: a Path instance.

\[ \text{transform_path(path)} \text{ is equivalent to transform_path_affine(transform_path_non_affine(values))}. \]

transform_path_affine(path)
Returns a path, transformed only by the non-affine part of this transform.

path: a Path instance.

\[ \text{transform_path(path)} \text{ is equivalent to transform_path_affine(transform_path_non_affine(values))}. \]
transform_path_non_affine(path)
Returns a path, transformed only by the non-affine part of this transform.

path: a Path instance.

transform_path(path) is equivalent to transform_path_affine(transform_path_non_affine(values)).

class matplotlib.transforms.BlendedGenericTransform(x_transform, y_transform, **kwargs)
Bases: 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.

frozen()
Returns a frozen copy of this transform node. The frozen copy will not update when its children change. Useful for storing a previously known state of a transform where copy.deepcopy() might normally be used.

get_affine()
Get the affine part of this transform.

inverted()
Return the corresponding inverse transformation.

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.

x == self.inverted().transform(self.transform(x))

transform_non_affine(points)
Performs only the non-affine part of the 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.

Accepts a numpy array of shape (N x input_dims) and returns a numpy array of shape (N x output_dims).

class matplotlib.transforms.BlendedAffine2D(x_transform, y_transform, **kwargs)
Bases: 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.

```python
def get_matrix()
    Get the Affine transformation array for the affine part of this transform.
```

```python
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.
```

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 use the `composite_transform_factory()` function instead, which can automatically choose the best kind of composite transform instance to create.

```python
def frozen()
    Returns a frozen copy of this transform node. The frozen copy will not update when its children change. Useful for storing a previously known state of a transform where `copy.deepcopy()` might normally be used.
```

def get_affine()
    Get the affine part of this transform.
```

def inverted()
    Return the corresponding inverse transformation.

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.

```python
x === self.inverted().transform(self.transform(x))
```

def transform_affine(points)
    Performs 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).

Accepts a numpy array of shape (N x input_dims) and returns a numpy array of shape (N x output_dims).
**transform_non_affine(points)**

Performs only the non-affine part of the 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.

Accepts a numpy array of shape (N x input_dims) and returns a numpy array of shape (N x output_dims).

**transform_path_non_affine(path)**

Returns a path, transformed only by the non-affine part of this transform.

path: a Path instance.

transform_path(path) is equivalent to transform_path_affine(transform_path_non_affine(values)).

### 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 transform a then transform b.

Both a and b must be instances of Affine2DBase.

You will generally not call this constructor directly but use the composite_transform_factory() function instead, which can automatically choose the best kind of composite transform instance to create.

**get_matrix()**

Get the Affine transformation array for the affine part of this transform.

### class matplotlib.transforms.BboxTransform(boxin, boxout, **kwargs)

Bases: matplotlib.transforms.Affine2DBase

BboxTransform linearly transforms points from one Bbox to another Bbox.

Create a new BboxTransform that linearly transforms points from boxin to boxout.

**get_matrix()**

Get the Affine transformation array for the affine part of this transform.

### 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.

get_matrix()
Get the Affine transformation array for the affine part of this transform.

class matplotlib.transforms.BboxTransformFrom(boxin, **kwargs)
Bases: matplotlib.transforms.Affine2DBase

BboxTransformFrom linearly transforms points from a given Bbox to the unit bounding box.

get_matrix()
Get the Affine transformation array for the affine part of this transform.

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 transformad by the given transform scale_trans.

get_matrix()
Get the Affine transformation array for the affine part of this transform.

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.

Create a new TransformedPath from the given Path and Transform.

get_fully_transformed_path()
Return a fully-transformed copy of the child path.

get_transformed_path_and_affine()
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()
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.nonsingular(vmin, vmax, expander=0.001, tiny=1e-15, increasing=True)
Modify the endpoints of a range as needed to avoid singularities.

vmin, vmax the initial endpoints.
tiny 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.

expander fractional amount by which vmin and vmax are expanded if the original interval is too small, based on tiny.

increasing: [True | False] If True (default), swap vmin, vmax if vmin > vmax

Returns vmin, vmax, expanded and/or swapped if necessary.

If either input is inf or NaN, or if both inputs are 0, returns -expander, expander.
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 set_xscale() and set_xscale(). Projections can be chosen using the projection keyword argument to the plot() or subplot() functions, e.g.:

```
plot(x, y, 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.

### 35.1 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 `examples/api/custom_scale_example.py`. There are also some classes in `matplotlib.scale` that may be used as starting points.
35.2 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:
  ```python
  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:
  ```python
  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 `examples/api/custom_projection_example.py`. The polar plot functionality in `matplotlib.projections.polar` may also be of interest.
35.3 API documentation

35.3.1 matplotlib.scale

class matplotlib.scale.LinearScale(axis, **kwargs)
   Bases: matplotlib.scale.ScaleBase

   The default linear scale.

get_transform()
   The transform for linear scaling is just the IdentityTransform.

set_default_locators_and_formatters(axis)
   Set the locators and formatters to reasonable defaults for linear scaling.

class matplotlib.scale.LogScale(axis, **kwargs)
   Bases: matplotlib.scale.ScaleBase

   A standard logarithmic scale. Care is taken so non-positive values are not plotted.

   For computational efficiency (to push as much as possible to Numpy C code in the common cases),
   this scale provides different transforms depending on the base of the logarithm:

   • base 10 (Log10Transform)
   • base 2 (Log2Transform)
   • base e (NaturalLogTransform)
   • arbitrary base (LogTransform)

   basex/basey: The base of the logarithm

   nonposx/nonposy: ['mask' | 'clip'] non-positive values in x or y can be masked as invalid, or clipped
                   to a very small positive number

   subsx/subsy: Where to place the subticks between each major tick. Should be a sequence of integers.
               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.

   get_transform()
      Return a Transform instance appropriate for the given logarithm base.

   limit_range_for_scale(vmin, vmax, minpos)
      Limit the domain to positive values.

   set_default_locators_and_formatters(axis)
      Set the locators and formatters to specialized versions for log scaling.

class matplotlib.scale.ScaleBase
   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()

And optionally:

- set_default_locators_and_formatters()
- limit_range_for_scale()

get_transform()
Return the Transform object associated with this scale.

limit_range_for_scale(vmin, vmax, minpos)
Returns the range vmin, vmax, possibly limited to the domain supported by this scale.

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(axis)
Set the Locator and Formatter objects on the given axis to match 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).

basex/basey: The base of the logarithm

linthreshx/linthreshy: The range (-x, x) within which the plot is linear (to avoid having the plot go to infinity around zero).

subsx/subsy: Where to place the subticks between each major tick. Should be a sequence of integers.
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.

linscalex/linscaley: This allows the linear range (-linthresh to 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.

get_transform()
Return a SymmetricalLogTransform instance.

set_default_locators_and_formatters(axis)
Set the locators and formatters to specialized versions for symmetrical log scaling.

matplotlib.scale.get_scale_docs()
Helper function for generating docstrings related to scales.
matplotlib.scale.register_scale(scale_class)
Register a new kind of scale.

*scale_class* must be a subclass of *ScaleBase*.

matplotlib.scale.scale_factory(scale, axis, **kwargs)
Return a scale class by name.

Accepts: [linear | log | symlog]

### 35.3.2 matplotlib.projections

class matplotlib.projections.ProjectionRegistry
Bases: object

Manages the set of projections available to the system.

get_projection_class(name)
Get a projection class from its name.

get_projection_names()
Get a list of the names of all projections currently registered.

register(*projections)
Register a new set of projection(s).

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()
Get a list of acceptable projection names.

matplotlib.projections.process_projection_requirements(figure, *args, **kwargs)
Handle the args/kwargs to for add_axes/add_subplot(gca, returning:

(axes_proj_class, proj_class_kwargs, proj_stack_key)
Which can be used for new axes initialization/identification.

Note: *kwargs* is modified in place.

matplotlib.projections.projection_factory(projection, figure, rect, **kwargs)
Get a new projection instance.

*projection* is a projection name.

*figure* is a figure to add the axes to.

*rect* is a Bbox object specifying the location of the axes within the figure.

Any other kwargs are passed along to the specific projection constructor being used.
Matplotlib, Release 1.3.0

Deprecated since version 1.3: This routine is deprecated in favour of getting the projection class directly with `get_projection_class()` and initialising it directly. Will be removed in version 1.3.

**matplotlib.projections.polar**

**class matplotlib.projections.polar.InvertedPolarTransform(axis=None, use_rmin=True)**

Bases: `matplotlib.transforms.Transform`

The inverse of the polar transform, mapping Cartesian coordinate space \( x \) and \( y \) back to \( \theta \) and \( r \).

**inverted()**

Return the corresponding inverse transformation.

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.

\[ x = self.inverted().transform(self.transform(x)) \]

**transform_non_affine(xy)**

Performs only the non-affine part of the transformation.

\[ \text{transform(values)} \text{ is always equivalent to transform_affine(transform_non_affine(values))}. \]

In non-affine transformations, this is generally equivalent to `transform(values)`. In affine transformations, this is always a no-op.

Accepts a numpy array of shape \((N \times \text{input_dims})\) and returns a numpy array of shape \((N \times \text{output_dims})\).

**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.

\( \text{limits} \) is the view limit of the data. The only part of its bounds that is used is ymax (for the radius maximum). The theta range is always fixed to \((0, 2\pi)\).

**get_matrix()**

Get the Affine transformation array for the affine part of this transform.

**class matplotlib.projections.polar.PolarAxes(*args, **kwargs)**

Bases: `matplotlib.axes.Axes`

A polar graph projection, where the input dimensions are \( \theta \), \( r \).

Theta starts pointing east and goes anti-clockwise.

**class InvertedPolarTransform(axis=None, use_rmin=True)**

Bases: `matplotlib.transforms.Transform`

The inverse of the polar transform, mapping Cartesian coordinate space \( x \) and \( y \) back to \( \theta \) and \( r \).
inverted()
Return the corresponding inverse transformation.

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.

\[ x === \text{self.inverted().transform(self.transform(x))} \]

transform_non_affine(xy)
Performs only the non-affine part of the 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.

Accepts a numpy array of shape (N x input_dims) and returns a numpy array of shape (N x output_dims).

class PolarAxes.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 ymax (for the radius maximum). The theta range is always fixed to (0, 2pi).

get_matrix()
Get the affine transformation array for the affine part of this transform.

class PolarAxes.PolarTransform(axis=None, use_rmin=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.

inverted()
Return the corresponding inverse transformation.

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.

\[ x === \text{self.inverted().transform(self.transform(x))} \]

transform_non_affine(tr)
Performs only the non-affine part of the 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.

Accepts a numpy array of shape (N x input_dims) and returns a numpy array of shape (N x output_dims).

transform_path_non_affine(path)
Returns a path, transformed only by the non-affine part of this transform.
path: a Path instance.

transform_path(path) is equivalent to transform_path_affine(transform_path_non_affine(values))

class PolarAxes.RadialLocator(base)
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).

class PolarAxes.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.

PolarAxes.can_pan()
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.

PolarAxes.can_zoom()
Return True if this axes supports the zoom box button functionality.

Polar axes do not support zoom boxes.

PolarAxes.format_coord(theta, r)
Return a format string formatting the coordinate using Unicode characters.

PolarAxes.get_data_ratio()
Return the aspect ratio of the data itself. For a polar plot, this should always be 1.0

PolarAxes.get_theta_direction()
Get the direction in which theta increases.

-1: Theta increases in the clockwise direction
1: Theta increases in the counterclockwise direction

PolarAxes.get_theta_offset()
Get the offset for the location of 0 in radians.

PolarAxes.set_rgrids(radii, labels=None, angle=None, fmt=None, **kwargs)
Set the radial locations and labels of the r grids.

The labels will appear at radial distances radii at the given angle in degrees.

labels, if not None, is a len(radii) list of strings of the labels to use at each radius.

If labels is None, the built-in formatter will be used.

Return value is a list of tuples (line, label), where line is Line2D instances and the label is Text instances.

kwargs are optional text properties for the labels:
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>backgroundcolor</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>bbox</td>
<td>rectangle prop dict</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>family or fontfamily or fontname or name</td>
<td>[FONTNAME</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fontproperties or font_properties</td>
<td>a matplotlib.font_manager.FontProperties instance</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>horizontalalignment or ha</td>
<td>[ ‘center’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linespacing</td>
<td>float (multiple of font size)</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>multialignment</td>
<td>[‘left’</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>position</td>
<td>(x,y)</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>rotation</td>
<td>[ angle in degrees</td>
</tr>
<tr>
<td>rotation_mode</td>
<td>unknown</td>
</tr>
<tr>
<td>size or fontsize</td>
<td>[size in points</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>stretch or fontstretch</td>
<td>[a numeric value in range 0-1000</td>
</tr>
<tr>
<td>style or fontstyle</td>
<td>[ ‘normal’</td>
</tr>
<tr>
<td>text</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>variant or fontvariant</td>
<td>[ ‘normal’</td>
</tr>
<tr>
<td>verticalalignment or va or ma</td>
<td>[ ‘center’</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>x</td>
<td>float</td>
</tr>
<tr>
<td>y</td>
<td>float</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

ACCEPTS: sequence of floats
PolarAxes.set_theta_direction(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

PolarAxes.set_theta_offset(offset)
Set the offset for the location of 0 in radians.

PolarAxes.set_theta_zero_location(loc)
Sets the location of theta’s zero. (Calls set_theta_offset with the correct value in radians under the hood.)

May be one of “N”, “NW”, “W”, “SW”, “S”, “SE”, “E”, or “NE”.

PolarAxes.set_thetagrids(angles, labels=None, frac=None, fmt=None, **kwargs)
Set the angles at which to place the theta grids (these gridlines are equal along the theta dimension). angles is in degrees.

labels, if not None, is a len(angles) list of strings of the labels to use at each angle.

If labels is None, the labels will be fmt % angle

frac is the fraction of the polar axes radius at which to place the label (1 is the edge). e.g., 1.05 is outside the axes and 0.95 is inside the axes.

Return value is a list of tuples (line, label), where line is Line2D instances and the label is Text instances.

kwargs are optional text properties for the labels:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>backgroundcolor</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>bbox</td>
<td>rectangle prop dict</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>family or fontfamily or fontname or name</td>
<td>[FONTNAME</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fontproperties or font_properties</td>
<td>a matplotlib.font_manager.FontProperties instance</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>horizontalalignment or ha</td>
<td>[ ‘center’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ’%s’ conversion.</td>
</tr>
<tr>
<td>linespacing</td>
<td>float (multiple of font size)</td>
</tr>
</tbody>
</table>
Table 35.2 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>multialignement</td>
<td>['left’</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>position</td>
<td>(x,y)</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>rotation</td>
<td>[ angle in degrees</td>
</tr>
<tr>
<td>rotation_mode</td>
<td>unknown</td>
</tr>
<tr>
<td>size or fontsize</td>
<td>[size in points</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>stretch or fontstretch</td>
<td>[a numeric value in range 0-1000</td>
</tr>
<tr>
<td>style or fontstyle</td>
<td>[ ‘normal’</td>
</tr>
<tr>
<td>text</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>variant or fontvariant</td>
<td>[ ‘normal’</td>
</tr>
<tr>
<td>verticalalignment or va or ma</td>
<td>[ ‘center’</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>x</td>
<td>float</td>
</tr>
<tr>
<td>y</td>
<td>float</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

ACCEPTS: sequence of floats

class matplotlib.projections.polar.PolarTransform(axis=None, use_rmin=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.

    inverted()

Return the corresponding inverse transformation.

        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.

        x === self.inverted().transform(self.transform(x))

    transform_non_affine(tr)

Performs only the non-affine part of the 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.
Accepts a numpy array of shape (N x input_dims) and returns a numpy array of shape (N x output_dims).

**transform_path_non_affine(path)**

Returns a path, transformed only by the non-affine part of this transform.

*path*: a *Path* instance.

transform_path(path) is equivalent to transform_path_affine(transform_path_non_affine(values))

class matplotlib.projections.polar.RadialLocator(base)
    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).

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.
Part IV

Matplotlib AxesGrid Toolkit
The matplotlib AxesGrid 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, AxesGrid toolkit provides a framework to adjust the position of multiple axes according to their aspects.

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 (since svn r8226), 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.

Warning: axes_grid and axisartist (but not axes_grid1) uses a custom Axes class (derived from the mpl's original Axes class). As a side effect, some commands (mostly tick-related) do not work. Use axes_grid1 to avoid this, or see how things are different in axes_grid and axisartist (LINK needed)
36.1 What is AxesGrid toolkit?

The matplotlib AxesGrid toolkit is a collection of helper classes, mainly to ease displaying (multiple) images in matplotlib.

**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 (since svn r8226), 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.

**Warning:** `axes_grid` and `axisartist` (but not `axes_grid1`) uses a custom Axes class (derived from the mpl’s original Axes class). As a side effect, some commands (mostly tick-related) do not work. Use `axes_grid1` to avoid this, or see how things are different in `axes_grid` and `axisartist` (LINK needed).

AxesGrid toolkit has two namespaces (`axes_grid1` and `axisartist`). `axisartist` contains custom Axes class that is meant to support for curvilinear grids (e.g., the world coordinate system in astronomy). Unlike mpl’s original Axes class which uses Axes.xaxis and Axes.yaxis to draw ticks, ticklines and etc., Axes in axisartist uses special artist (AxisArtist) which can handle tick, ticklines and etc. for curved coordinate systems.
Since it uses a special artists, some mpl commands that work on Axes.xaxis and Axes.yaxis may not work. See LINK for more detail.

`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 you to have a colorbar whose height always matches that of the image. `ImageGrid`, `RGB Axes` and `AxesDivider` are helper classes that deals 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.
36.2 AXES_GRID1

36.2.1 ImageGrid

A class that creates a grid of Axes. In matplotlib, the axes location (and size) is specified in the normalized figure coordinates. This may not be ideal for images that needs to be displayed with a given aspect ratio. For example, displaying images of a same size with some fixed padding between them cannot be easily done in matplotlib. ImageGrid is used in such case.

```python
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import ImageGrid
import numpy as np

im = np.arange(100)
im.shape = 10, 10

fig = plt.figure(1, (4., 4.))
grid = ImageGrid(fig, 111,  # similar to subplot(111)
nrows_ncols = (2, 2),  # creates 2x2 grid of axes
axes_pad=0.1,  # pad between axes in inch.
)

for i in range(4):
    grid[i].imshow(im)  # The AxesGrid object work as a list of axes.

plt.show()
```
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).

```python
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import ImageGrid

def get_demo_image():
    import numpy as np
    from matplotlib.cbook import get_sample_data
    f = get_sample_data("axes_grid/bivariate_normal.npy", asfileobj=False)
    z = np.load(f)
    # z is a numpy array of 15x15
    return z, (-3,4,-4,3)

F = plt.figure(1, (5.5, 3.5))
grid = ImageGrid(F, 111,  # similar to subplot(111)
                 nrows_ncols = (1, 3),
                 axes_pad = 0.1,
                 add_all=True,
                 label_mode = "L",
                 )

Z, extent = get_demo_image()  # demo image
```
im1 = Z
im2 = Z[:, :10]
im3 = Z[:, 10:]
vmin, vmax = Z.min(), Z.max()
for i, im in enumerate([im1, im2, im3]):
    ax = grid[i]
    ax.imshow(im, origin="lower", vmin=vmin, vmax=vmax, interpolation="nearest")

plt.draw()
plt.show()

- 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.

When initialized, ImageGrid creates given number (ngrids or ncols * nrows if ngrids is None) of Axes
instances. A sequence-like interface is provided to access the individual Axes instances (e.g., grid[0] is the
first Axes in the grid. See below for the order of axes).

AxesGrid takes following arguments,
<table>
<thead>
<tr>
<th>Name</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fig</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rect</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nrows_ncols</td>
<td></td>
<td>number of rows and cols. e.g., (2,2)</td>
</tr>
<tr>
<td>ngrids</td>
<td>None</td>
<td>number of grids. nrows x ncols if None</td>
</tr>
<tr>
<td>direction</td>
<td>“row”</td>
<td>increasing direction of axes number. [row</td>
</tr>
<tr>
<td>axes_pad</td>
<td>0.02</td>
<td>pad between axes in inches</td>
</tr>
<tr>
<td>add_all</td>
<td>True</td>
<td>Add axes to figures if True</td>
</tr>
<tr>
<td>share_all</td>
<td>False</td>
<td>xaxis &amp; yaxis of all axes are shared if True</td>
</tr>
<tr>
<td>aspect</td>
<td>True</td>
<td>aspect of axes</td>
</tr>
<tr>
<td>label_mode</td>
<td>“L”</td>
<td>location of tick labels thaw will be displayed. “L” (only the lower left axes), “L” (left most and bottom most axes), or “all”.</td>
</tr>
<tr>
<td>cbar_mode</td>
<td>None</td>
<td>[None</td>
</tr>
<tr>
<td>cbar_location</td>
<td>right</td>
<td>[right</td>
</tr>
<tr>
<td>cbar_pad</td>
<td>None</td>
<td>pad between image axes and colorbar axes</td>
</tr>
<tr>
<td>cbar_size</td>
<td>“5%”</td>
<td>size of the colorbar</td>
</tr>
<tr>
<td>axes_class</td>
<td>None</td>
<td></td>
</tr>
</tbody>
</table>

**rect** specifies the location of the grid. You can either specify coordinates of the rectangle to be used (e.g., (0.1, 0.1, 0.8, 0.8) as in the Axes), or the subplot-like position (e.g., “121”).

**direction** means the increasing direction of the axes number.

**aspect** By default (False), widths and heights of axes in the grid are scaled independently. If True, they are scaled according to their data limits (similar to aspect parameter in mpl).

**share_all** if True, xaxis and yaxis of all axes are shared.

**direction** direction of increasing axes number. For “row”,

```
  grid[0]  grid[1]  
```

For “column”,

```
  grid[0]  grid[2]  
```

You can also create a colorbar (or colorbars). You can have colorbar for each axes (cbar_mode=”each”), or you can have a single colorbar for the grid (cbar_mode=”single”). The colorbar can be placed on your right, or top. The axes for each colorbar is stored as a cbar_axes attribute.

The examples below show what you can do with AxesGrid.
36.2.2 AxesDivider

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. While a more about the AxesDivider is (will be) explained in (yet to be written) AxesDividerGuide, 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.

```python
ax = subplot(1,1,1)
divider = make_axes_locatable(ax)
```

`make_axes_locatable` returns an instance of the AxesLocator class, derived from the Locator. It provides `append_axes` method that creates a new axes on the given side of (“top”, “right”, “bottom” and “left”) of the original axes.

36.2.3 colorbar whose height (or width) in sync with the master axes

```python
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import make_axes_locatable
import numpy as np

ax = plt.subplot(111)
im = ax.imshow(np.arange(100).reshape((10,10)))

# create an axes on the right side of ax. The width of cax will be 5% # of ax and the padding between cax and ax will be fixed at 0.05 inch.
divider = make_axes_locatable(ax)
cax = divider.append_axes("right", size="5%", pad=0.05)

plt.colorbar(im, cax=cax)
```
scatter_hist.py with AxesDivider

The “scatter_hist.py” example in mpl can be rewritten using `make_axes_locatable`.

```python
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 scatter_hist using the AxesDivider has some advantage over the original scatter_hist.py in mpl. For example, you can set the aspect ratio of the scatter plot, even with the x-axis or y-axis is shared accordingly.

### 36.2.4 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

```python
from mpl_toolkits.axes_grid1 import host_subplot
import matplotlib.pyplot as plt

host = host_subplot(111)
par = host.twinx()

host.set_xlabel("Distance")
host.set_ylabel("Density")
par.set_ylabel("Temperature")

p1, = host.plot([0, 1, 2], [0, 1, 2], label="Density")
p2, = par.plot([0, 1, 2], [0, 3, 2], label="Temperature")

leg = plt.legend()

host.yaxis.get_label().set_color(p1.get_color())
leg.texts[0].set_color(p1.get_color())

par.yaxis.get_label().set_color(p2.get_color())
leg.texts[1].set_color(p2.get_color())

plt.show()
```

Example 2. twin

twin without a transform argument treat the parasite axes to have a 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.

```python
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])
ax2.set_xticklabels(["0", r"\frac{1}{2}\pi", r"\pi", r"\frac{3}{2}\pi", r"2\pi"])
```

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.
### 36.2.5 AnchoredArtists

It’s a collection of artists whose location is anchored to the (axes) bbox, like the legend. It is derived from OffsetBox in mpl, 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.

```python
import matplotlib.pyplot as plt

def draw_text(ax):
    from mpl_toolkits.axes_grid1.anchored_artists import AnchoredText
    at = AnchoredText("Figure 1a", loc=2, prop=dict(size=8), frameon=True,
        at.patch.set_boxstyle("round,pad=0.,rounding_size=0.2")
    ax.add_artist(at)

    at2 = AnchoredText("Figure 1(b)", loc=3, prop=dict(size=8), frameon=True,
        bbox_to_anchor=(0., 1.),
        bbox_transform=ax.transAxes
    )
    at2.patch.set_boxstyle("round,pad=0.,rounding_size=0.2")
```
def draw_circle(ax):
    from mpl_toolkits.axes_grid1.anchored_artists import AnchoredDrawingArea
    from matplotlib.patches import Circle
    ada = AnchoredDrawingArea(20, 20, 0, 0,
                               loc=1, pad=0., frameon=False)
    p = Circle((10, 10), 10)
    ada.da.add_artist(p)
    ax.add_artist(ada)

def draw_ellipse(ax):
    from mpl_toolkits.axes_grid1.anchored_artists import AnchoredEllipse
    # draw an ellipse of width=0.1, height=0.15 in the data coordinate
    ae = AnchoredEllipse(ax.transData, width=0.1, height=0.15, angle=0.,
                         loc=3, pad=0.5, borderpad=0.4, frameon=True)
    ax.add_artist(ae)

def draw_sizebar(ax):
    from mpl_toolkits.axes_grid1.anchored_artists import AnchoredSizeBar
    # draw a horizontal bar with length of 0.1 in Data coordinate
    # (ax.transData) with a label underneath.
    asb = AnchoredSizeBar(ax.transData,
                          0.1,
                          r"1'$\prime$" ,
                          loc=8, 
                          pad=0.1, borderpad=0.5, sep=5, 
                          frameon=False)
    ax.add_artist(asb)

if 1:
    ax = plt.gca()
    ax.set_aspect(1.)
    
draw_text(ax)
draw_circle(ax)
draw_ellipse(ax)
draw_sizebar(ax)

plt.show()
36.2.6 InsetLocator

mpl_toolkits.axes_grid.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_grid.inset_locator.inset_axes(), you can have inset axes whose size is either fixed, or a fixed proportion of the parent axes. For example:

```
inset_axes = inset_axes(parent_axes,
                      width="30%",  # width = 30% of parent_bbox
                      height=1.,  # height : 1 inch
                      loc=3)
```

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=1)
```

creates an inset axes whose data scale is half of the parent axes. Here is complete examples.
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 `mpl_toolkits/axes_grid/inset_locator` provides a helper function `mark_inset()` to mark the location of the area represented by the inset axes.

```python
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1.inset_locator import zoomed_inset_axes
from mpl_toolkits.axes_grid1.inset_locator import mark_inset
import numpy as np

def get_demo_image():
    from matplotlib.cbook import get_sample_data
    import numpy as np
    f = get_sample_data("axes_grid/bivariate_normal.npy", asfileobj=False)
    z = np.load(f)
    # z is a numpy array of 15x15
    return z, (-3,4,-4,3)

fig, ax = plt.subplots(figsize=[5,4])

# prepare the demo image
Z, extent = get_demo_image()
Z2 = np.zeros([150, 150], dtype="d")
ny, nx = Z.shape
Z2[30:30+ny, 30:30+nx] = Z

# extent = [-3, 4, -4, 3]
ax.imshow(Z2, extent=extent, interpolation="nearest", origin="lower")

axins = zoomed_inset_axes(ax, 6, loc=1) # zoom = 6
axins.imshow(Z2, extent=extent, interpolation="nearest", origin="lower")
```
# sub region of the original image
x1, x2, y1, y2 = -1.5, -0.9, -2.5, -1.9
axins.set_xlim(x1, x2)
axins.set_ylim(y1, y2)
plt.xticks(visible=False)
plt.yticks(visible=False)

# draw a bbox of the region of the inset axes in the parent axes and
# connecting lines between the bbox and the inset axes area
mark_inset(ax, axins, loc1=2, loc2=4, fc="none", ec="0.5")

plt.draw()
plt.show()

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(1)
ax = RGBAxes(fig, [0.1, 0.1, 0.8, 0.8])
r, g, b = get_rgb()  # r,g,b are 2-d images
36.3 AXISARTIST

36.3.1 AxisArtist

AxisArtist module provides a custom (and very experimental) Axes class, where each axis (left, right, top and bottom) have a separate artist associated which is responsible to draw axis-line, ticks, ticklabels, label. Also, you can 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, have its xaxis and yaxis invisible, and has 4 additional artists which are responsible to draw axis 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 methods and it behaves as an original Axes.axis method in mpl).

To create an axes,

```python
import mpl_toolkits.axisartist as AA
fig = plt.figure(1)
ax = AA.Axes(fig, [0.1, 0.1, 0.8, 0.8])
fig.add_axes(ax)
```

```python
ax.imshow_rgb(r, g, b,
              origin="lower", interpolation="nearest")
```
or to create a subplot

```python
ax = AA.Subplot(fig, 111)
fig.add_subplot(ax)
```

For example, you can hide the right, and top axis by

```python
ax.axis["right"].set_visible(False)
ax.axis["top"].set_visible(False)
```

It is also possible to add an extra axis. For example, you may have an horizontal axis at \(y=0\) (in data coordinate).

```python
ax.axis["y=0"] = ax.new_floating_axis(nth_coord=0, value=0)
```

```python
import matplotlib.pyplot as plt
import mpl_toolkits.axisartist as AA

fig = plt.figure(1)
fig.subplots_adjust(right=0.85)
ax = AA.Subplot(fig, 1, 1, 1)
fig.add_subplot(ax)

# make some axis invisible
ax.axis["bottom", "top", "right"].set_visible(False)

# make an new axis along the first axis axis (x-axis) which pass # throught \(y=0\).
ax.axis["y=0"] = ax.new_floating_axis(nth_coord=0, value=0,
axis_direction="bottom")

ax.axis["y=0"].toggle(all=True)
ax.axis["y=0"].label.set_text("y = 0")
```
ax.set_ylim(-2, 4)
plt.show()

Or a fixed axis with some offset

```python
# 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 a axes_class keyword argument, and the commands creates an axes of the given class. For example, to create a host subplot with axisartist.Axes,

```python
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.
36.3.2 Curvilinear Grid

The motivation behind the AxisArtist module is to support curvilinear grid and ticks.
See AXISARTIST namespace for more details.

### 36.3.3 Floating Axes

This also support a Floating Axes whose outer axis are defined as floating axis.
37.1 AxesDivider

The axes_divider module provide helper classes to adjust the axes positions of set of images in the drawing time.

- **axes_size** provides a classes of units that the size of each axes will be determined. For example, you can specify a fixed size
- **Divider** this is the class that is used calculates the axes position. It divides the given rectangular area into several areas. You initialize the divider by setting the horizontal and vertical list of sizes that the division will be based on. You then use the new_locator method, whose return value is a callable object that can be used to set the axes_locator of the axes.

You first initialize the divider by specifying its grids, i.e., horizontal and vertical.

for example,:;

```
rect = [0.2, 0.2, 0.6, 0.6]
horiz=[h0, h1, h2, h3]
vert=[v0, v1, v2]
divider = Divider(fig, rect, horiz, vert)
```

where, rect is a bounds of the box that will be divided and h0...h3, v0...v2 need to be an instance of classes in the **axes_size**. They have *get_size* method that returns a tuple of two floats. The first float is the relative size, and the second float is the absolute size. Consider a following grid.

```
v0   
v1   
h0,v2 h1 h2 h3
```

- v0 => 0, 2
- v1 => 2, 0
- v2 => 3, 0
The height of the bottom row is always 2 (axes_divider internally assumes that the unit is inch). The first and the second rows with height ratio of 2:3. For example, if the total height of the grid 6, then the first and second row will each occupy \( \frac{2}{2+3} \) and \( \frac{3}{2+3} \) of (6-1) inches. The widths of columns (horiz) will be similarly determined. When aspect ratio is set, the total height (or width) will be adjusted accordingly.

The `mpl_toolkits.axes_grid.axes_size` contains several classes that can be used to set the horizontal and vertical configurations. For example, for the vertical configuration above will be:

```python
from mpl_toolkits.axes_grid.axes_size import Fixed, Scaled
vert = [Fixed(2), Scaled(2), Scaled(3)]
```

After you set up the divider object, then you create a locator instance which will be given to the axes:

```python
locator = divider.new_locator(nx=0, ny=1)
ax.set_axes_locator(locator)
```

The return value of the new_locator method is a instance of the AxesLocator class. It is a callable object that returns the location and size of the cell at the first column and the second row. You may create a locator that spans over multiple cells:

```python
locator = divider.new_locator(nx=0, nx=2, ny=1)
```

The above locator, when called, will return the position and size of the cells spanning the first and second column and the first row. You may consider it as \([0:2, 1]\).

See the example,

```python
import mpl_toolkits.axes_grid.axes_size as Size
from mpl_toolkits.axes_grid import Divider
import matplotlib.pyplot as plt
fig1 = plt.figure(1, (5.5, 4.))
# the rect parameter will be ignore as we will set axes_locator
rect = (0.1, 0.1, 0.8, 0.8)
ax = [fig1.add_axes(rect, label="%d") for i in range(4)]
horiz = [Size.Scaled(1.5), Size.Fixed(.5), Size.Scaled(1.), Size.Scaled(.5)]
vert = [Size.Scaled(1.), Size.Fixed(.5), Size.Scaled(1.5)]

# divide the axes rectangle into grid whose size is specified by horiz * vert
divider = Divider(fig1, rect, horiz, vert, aspect=False)
ax[0].set_axes_locator(divider.new_locator(nx=0, ny=0))
ax[1].set_axes_locator(divider.new_locator(nx=0, ny=2))
ax[2].set_axes_locator(divider.new_locator(nx=2, ny=2))
ax[3].set_axes_locator(divider.new_locator(nx=2, nx1=4, ny=0))

for ax1 in ax:
    plt.setp(ax1.get_xticklabels()+ax1.get_yticklabels(),
             visible=False)
```

```
You can adjust the size of the each axes according to their x or y data limits (AxesX and AxesY), similar to the axes aspect parameter.

```python
import mpl_toolkits.axes_grid.axes_size as Size
from mpl_toolkits.axes_grid import Divider
import matplotlib.pyplot as plt

fig1 = plt.figure(1, (5.5, 4))
# the rect parameter will be ignore as we will set axes_locator
rect = (0.1, 0.1, 0.8, 0.8)
ax = [fig1.add_axes(rect, label="%d") for i in range(4)]

horiz = [Size.AxesX(ax[0]), Size.Fixed(.5), Size.AxesX(ax[1])]
vert = [Size.AxesY(ax[0]), Size.Fixed(.5), Size.AxesY(ax[2])]

# divide the axes rectangle into grid whose size is specified by horiz * vert
divider = Divider(fig1, rect, horiz, vert, aspect=False)

ax[0].set_axes_locator(divider.new_locator(nx=0, ny=0))
ax[1].set_axes_locator(divider.new_locator(nx=2, ny=0))
ax[2].set_axes_locator(divider.new_locator(nx=0, ny=2))
```

37.1. AxesDivider
ax[3].set_axes_locator(divider.new_locator(nx=2, ny=2))

ax[0].set_xlim(0, 2)
ax[1].set_xlim(0, 1)

ax[0].set_ylim(0, 1)
ax[2].set_ylim(0, 2)

divider.set_aspect(1.)

for ax1 in ax:
    plt.setp(ax1.get_xticklabels()+ax1.get_yticklabels(),
             visible=False)

plt.draw()
plt.show()
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

```python
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 mpl may have no effect. In addition to AxisArtist instances, the mpl_toolkits.axisartist.Axes class 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 Grid-
HelperRectlinear class is a wrapper around the $xaxis$ and $yaxis$ of mpl’s original Axes, and it was meant to work as the way how mpl’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.

### 37.2.1 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:

```python
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

```python
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.

### 37.2.2 HowTo

1. Changing tick locations and label.

   Same as the original mpl’s axes:

   ```python
   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:

   ```python
   ax.axis["left"].major_ticklabels.set_color("r")
   ```

3. To change the attributes of multiple axis:
or to change the attributes of all axis:

```python
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.

### 37.2.3 Rotation and Alignment of TickLabels

This is also quite different from the original mpl and can be confusing. When you want to rotate the ticklabels, first consider using “set_axis_direction” method.

```python
ax1.axis["left"].major_ticklabels.set_axis_direction("top")
ax1.axis["right"].label.set_axis_direction("left")
```

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.
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 (-).

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.

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.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>left</th>
<th>bottom</th>
<th>right</th>
<th>top</th>
</tr>
</thead>
<tbody>
<tr>
<td>axislabel direction</td>
<td><code>-</code></td>
<td>`'-'</td>
<td>`'+'</td>
<td><code>'</code></td>
<td></td>
</tr>
<tr>
<td>axislabel rotation</td>
<td>180</td>
<td>0</td>
<td>0</td>
<td>180</td>
<td></td>
</tr>
<tr>
<td>axislabel va</td>
<td>center</td>
<td>top</td>
<td>center</td>
<td>bottom</td>
<td></td>
</tr>
<tr>
<td>axislabel ha</td>
<td>right</td>
<td>center</td>
<td>right</td>
<td>center</td>
<td></td>
</tr>
<tr>
<td>ticklabel direction</td>
<td><code>-</code></td>
<td>`'+'</td>
<td>`'+'</td>
<td><code>'</code></td>
<td></td>
</tr>
<tr>
<td>ticklabels rotation</td>
<td>90</td>
<td>0</td>
<td>-90</td>
<td>180</td>
<td></td>
</tr>
<tr>
<td>ticklabel ha</td>
<td>right</td>
<td>center</td>
<td>right</td>
<td>center</td>
<td></td>
</tr>
<tr>
<td>ticklabel va</td>
<td>center</td>
<td>baseline</td>
<td>center</td>
<td>baseline</td>
<td></td>
</tr>
</tbody>
</table>

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.
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.

```python
ax1.axis["left"].set_axis_direction("top")
```
changes axis_direction of all the associated artist with the “left” axis, while

```python
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.

```python
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).

```python
ax.axis[:].major_ticks.set_tick_out(True)
```

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
Matplotlib, Release 1.3.0

- 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

![Diagram](image)

**Adjusting pad**

To change the pad between ticks and ticklabels

```python
ax.axis["left"][].major_ticklabels.set_pad(10)
```

Or ticklabels and axis-label
ax.axis["left"].label.set_pad(10)

default ticklabels.set_pad(10) label.set_pad(20) ticks.set_tick_out(True)

37.2.4 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 rectilinear coordinate.
def tr(x, y):
    x, y = np.asarray(x), np.asarray(y)
    return x, y-x

# from rectilinear 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)

tick_formatter1 = angle_helper.FormatterDMS()

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)
37.2.5 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.

37.2.6 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
38.1 mpl_toolkits.axes_grid.axes_size

class mpl_toolkits.axes_grid.axes_size.Fixed(fixed_size)
   Simple fixed size with absolute part = fixed_size and relative part = 0

class mpl_toolkits.axes_grid.axes_size.Scaled(scalable_size)
   Simple scaled (?) size with absolute part = 0 and relative part = scalable_size

class mpl_toolkits.axes_grid.axes_size.AxesX(axes, aspect=1.0)
   Scaled size whose relative part corresponds to the data width of the axes multiplied by the aspect.

class mpl_toolkits.axes_grid.axes_size.AxesY(axes, aspect=1.0)
   Scaled size whose relative part corresponds to the data height of the axes multiplied by the aspect.

class mpl_toolkits.axes_grid.axes_size.MaxWidth(artist_list)
   Size whose absolute part is the largest width of the given artist_list.

class mpl_toolkits.axes_grid.axes_size.MaxHeight(artist_list)
   Size whose absolute part is the largest height of the given artist_list.

class mpl_toolkits.axes_grid.axes_size.Fraction(fraction, ref_size)
   An instance whose size is a fraction of the ref_size.

>>> s = Fraction(0.3, AxesX(ax))

class mpl_toolkits.axes_grid.axes_size.Padded(size, pad)
   Return a instance where the absolute part of size is increase by the amount of pad.

mpl_toolkits.axes_grid.axes_size.from_any(size, fraction_ref=None)
   Creates 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.:
38.2 mpl_toolkits.axes_grid.axes_divider

**class mpl_toolkits.axes_grid.axes_divider.Divider(fig, pos, horizontal, vertical, aspect=None, anchor='C')**

This is the class that is used to calculate the axes position. It divides the given rectangular area into several sub-rectangles. You initialize the divider by setting the horizontal and vertical lists of sizes (mpl_toolkits.axes_grid.axes_size) that the division will be based on. You then use the `new_locator` method to create a callable object that can be used to as the `axes_locator` of the axes.

**Parameters**

- **fig** – matplotlib figure
- **pos** – position (tuple of 4 floats) of the rectangle that will be divided.
- **horizontal** – list of sizes (axes_size) for horizontal division
- **vertical** – list of sizes (axes_size) for vertical division
- **aspect** – if True, the overall rectangular area is reduced so that the relative part of the horizontal and vertical scales have the same scale.
- **anchor** – Determine how the reduced rectangle is placed when aspect is True.

**add_auto_adjustable_area**(use_axes, pad=0.1, adjust_dirs=['left', 'right', 'bottom', 'top'])

**append_size**(position, size)

**get_anchor()**

return the anchor

**get_aspect()**

return aspect

**get_horizontal()**

return horizontal sizes

**get_horizontal_sizes**(renderer)

**get_locator()**

**get_position()**

return the position of the rectangle.

**get_position_runtime**(ax, renderer)

**get_vertical()**

return vertical sizes

**get_vertical_sizes**(renderer)
get_vsize_hsize()

locate(nx, ny, nx1=None, ny1=None, axes=None, renderer=None)

Parameters

• nx1 (nx,) – 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.

• ny1 (ny,) – same as nx and nx1, but for row positions.

new_locator(nx, ny, nx1=None, ny1=None)
returns a new locator (mpl_toolkits.axes_grid.axes_divider.AxesLocator) for specified cell.

Parameters

• nx1 (nx,) – 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.

• ny1 (ny,) – same as nx and nx1, but for row positions.

set_anchor(anchor)

Parameters anchor – anchor position

<table>
<thead>
<tr>
<th>value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘C’</td>
<td>Center</td>
</tr>
<tr>
<td>‘SW’</td>
<td>bottom left</td>
</tr>
<tr>
<td>‘S’</td>
<td>bottom</td>
</tr>
<tr>
<td>‘SE’</td>
<td>bottom right</td>
</tr>
<tr>
<td>‘E’</td>
<td>right</td>
</tr>
<tr>
<td>‘NE’</td>
<td>top right</td>
</tr>
<tr>
<td>‘N’</td>
<td>top</td>
</tr>
<tr>
<td>‘NW’</td>
<td>top left</td>
</tr>
<tr>
<td>‘W’</td>
<td>left</td>
</tr>
</tbody>
</table>

set_aspect(aspect=False)

Parameters anchor – True or False

set_horizontal(h)

Parameters horizontal – list of sizes (axes_size) for horizontal division

set_locator(_locator)
set_position(pos)
    set the position of the rectangle.

    Parameters pos – position (tuple of 4 floats) of the rectangle that will be divided.

set_vertical(v)

    Parameters horizontal – list of sizes (axes_size) for horizontal division

class mpl_toolkits.axes_grid.axes_divider.AxesLocator(axes_divider, nx, ny, nx1=None, ny1=None)
A simple callable object, initialized with AxesDivider class, returns the position and size of the given cell.

Parameters

- axes_divider – An instance of AxesDivider class.
- nx1 (nx,) – 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 is specified.
- ny1 (ny,) – same as nx and nx1, but for row positions.

get_subplotspec()

class mpl_toolkits.axes_grid.axes_divider.SubplotDivider(fig, *args, **kwargs)
The Divider class whose rectangle area is specified as a subplot geometry.

    fig is a matplotlib.figure.Figure instance.

    args is the tuple (numRows, numCols, plotNum), where the array of subplots in the figure has dimensions numRows, numCols, and where plotNum is the number of the subplot being created. plotNum starts at 1 in the upper left corner and increases to the right.

    If numRows <= numCols <= plotNum < 10, args can be the decimal integer numRows * 100 + numCols * 10 + plotNum.

change_geometry(numrows, numcols, num)
    change subplot geometry, e.g., from 1,1,1 to 2,2,3

generate()
    get the subplot geometry, eg 2,2,3

generate()
    return the bounds of the subplot box

generate()
    get the SubplotSpec instance

generate() (subplotspec)
    set the SubplotSpec instance
update_params()
update the subplot position from fig.subplotpars

class mpl_toolkits.axes_grid.axes_divider.AxesDivider(axes, xref=None, yref=None)
Divider based on the pre-existing axes.

Parameters
axes – axes

append_axes(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.

new_horizontal(size, pad=None, pack_start=False, **kwargs)
Add a new axes on the right (or left) side of the main axes.

Parameters
• size – A width of the axes. A axes_size instance or if float or string is given,
  from_any function is used to create one, with ref_size set to AxesX instance of
  the current axes.
• pad – pad between the axes. It takes same argument as size.
• pack_start – 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.

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(size, pad=None, pack_start=False, **kwargs)
Add a new axes on the top (or bottom) side of the main axes.

Parameters
• size – A height of the axes. A axes_size instance or if float or string is given,
  from_any function is used to create one, with ref_size set to AxesX instance of
  the current axes.
• pad – pad between the axes. It takes same argument as size.
• pack_start – If False, the new axes is appended at the end of the list, i.e., it
  became the top-most axes. If True, it is inserted at the start of the list, and
  becomes the bottom-most axes.

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.
class mpl_toolkits.axes_grid.axes_grid.Grid(fig, rect, nrows_ncols, ngrids=None, direction='row', axes_pad=0.02, add_all=True, share_all=False, share_x=True, share_y=True, label_mode='L', axes_class=None)

Build an Grid instance with a grid nrows*ncols Axes in Figure fig with rect=[left, bottom, width, height] (in Figure coordinates) or the subplot position code (e.g., “121”).

Optional keyword arguments:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>direction</td>
<td>“row”</td>
<td>[“row”</td>
</tr>
<tr>
<td>axes_pad</td>
<td>0.02</td>
<td>float pad between axes given in inches</td>
</tr>
<tr>
<td>add_all</td>
<td>True</td>
<td>[ True</td>
</tr>
<tr>
<td>share_all</td>
<td>False</td>
<td>[ True</td>
</tr>
<tr>
<td>share_x</td>
<td>True</td>
<td>[ True</td>
</tr>
<tr>
<td>share_y</td>
<td>True</td>
<td>[ True</td>
</tr>
<tr>
<td>label_mode</td>
<td>“L”</td>
<td>[“L”</td>
</tr>
<tr>
<td>axes_class</td>
<td>None</td>
<td>a type object which must be a subclass of Axes</td>
</tr>
</tbody>
</table>

class mpl_toolkits.axes_grid.axes_grid.ImageGrid(fig, rect, nrows_ncols, ngrids=None, direction='row', axes_pad=0.02, add_all=True, 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)

Build an ImageGrid instance with a grid nrows*ncols Axes in Figure fig with rect=[left, bottom, width, height] (in Figure coordinates) or the subplot position code (e.g., “121”).

Optional keyword arguments:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>direction</td>
<td>“row”</td>
<td>[“row”</td>
</tr>
<tr>
<td>axes_pad</td>
<td>0.02</td>
<td>float pad between axes given in inches</td>
</tr>
<tr>
<td>add_all</td>
<td>True</td>
<td>[ True</td>
</tr>
<tr>
<td>share_all</td>
<td>False</td>
<td>[ True</td>
</tr>
<tr>
<td>aspect</td>
<td>True</td>
<td>[ True</td>
</tr>
<tr>
<td>label_mode</td>
<td>“L”</td>
<td>[“L”</td>
</tr>
<tr>
<td>cbar_mode</td>
<td>None</td>
<td>[“each”</td>
</tr>
<tr>
<td>cbar_location</td>
<td>“right”</td>
<td>[“left”</td>
</tr>
<tr>
<td>cbar_pad</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>cbar_size</td>
<td>“5%”</td>
<td></td>
</tr>
<tr>
<td>cbar_set_cax</td>
<td>True</td>
<td>[ True</td>
</tr>
<tr>
<td>axes_class</td>
<td>None</td>
<td>a type object which must be a subclass of Axes</td>
</tr>
</tbody>
</table>
Matplotlib, Release 1.3.0

```
cbar_set_cax [if True, each axes in the grid has a cax] attribute that is bind to associated cbar_axes.

38.4 mpl_toolkits.axes_grid.axis_artist

class mpl_toolkits.axes_grid.axis_artist.AxisArtist(axes, helper, offset=None, axis_direction='bottom', **kw)

An artist which draws axis (a line along which the n-th axes coord is constant) line, ticks, ticklabels, and axis label.

axes : axes helper : an AxisArtistHelper instance.

LABELPAD

ZORDER = 2.5

draw(artist, renderer, *args, **kwargs)

Draw the axis lines, tick lines and labels

get_axisline_style()

return the current axisline style.

get_helper()

Return axis artist helper instance.

get_tightbbox(renderer)

get_transform()

invert_ticklabel_direction()

set_axis_direction(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].

<table>
<thead>
<tr>
<th>property</th>
<th>left</th>
<th>bottom</th>
<th>right</th>
<th>top</th>
</tr>
</thead>
<tbody>
<tr>
<td>ticklabels location</td>
<td>“-”</td>
<td>“-”</td>
<td>“-”</td>
<td>“-”</td>
</tr>
<tr>
<td>axislabel location</td>
<td>“-”</td>
<td>“+”</td>
<td>“+”</td>
<td>“-”</td>
</tr>
<tr>
<td>ticklabels angle</td>
<td>90</td>
<td>0</td>
<td>-90</td>
<td>180</td>
</tr>
<tr>
<td>ticklabel va</td>
<td>center</td>
<td>baseline</td>
<td>center</td>
<td>baseline</td>
</tr>
<tr>
<td>ticklabel ha</td>
<td>right</td>
<td>center</td>
<td>right</td>
<td>center</td>
</tr>
<tr>
<td>axislabel angle</td>
<td>180</td>
<td>0</td>
<td>0</td>
<td>180</td>
</tr>
<tr>
<td>axislabel va</td>
<td>center</td>
<td>top</td>
<td>center</td>
<td>bottom</td>
</tr>
<tr>
<td>axislabel ha</td>
<td>right</td>
<td>center</td>
<td>right</td>
<td>center</td>
</tr>
</tbody>
</table>
```
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**(label_direction)
Adjust the direction of the axislabel.

ACCEPES: [ “+” | “-” ]

Note that the label_direction ‘+’ and ‘-’ are relative to the direction of the increasing coordinate.

**set_axisline_style**(axisline_style=None, **kw)
Set the axisline style.

*axisline_style can be a string with axisline style name with optional comma-separated attributes. Alternatively, the attrs can be provided as keywords.*

```
set_arrowstyle(“->,size=1.5”) set_arrowstyle(“->”, size=1.5)
```

Old attrs simply are forgotten.

Without argument (or with arrowstyle=None), return available styles as a list of strings.

**set_label**(s)

**set_ticklabel_direction**(tick_direction)
Adjust the direction of the ticklabel.

ACCEPES: [ “+” | “-” ]

Note that the label_direction ‘+’ and ‘-’ are relative to the direction of the increasing coordinate.

**toggle**(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))
```

**class mpl_toolkits.axes_grid.axis_artist.Ticks**(ticksize, tick_out=False, **kwargs)
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).

**get_tick_out**()
Return True if the tick will be rotated by 180 degree.
get_ticksize()
Return length of the ticks in points.

set_tick_out(b)
set True if tick need to be rotated by 180 degree.

set_ticksize(ticksize)
set length of the ticks in points.

class mpl_toolkits.axes_grid.axis_artist.AxisLabel(*kl, **kwargs)
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.

get_pad()
return pad in points. See set_pad for more details.

set_axis_direction(d)
Adjust the text angle and text alignment of axis label according to the matplotlib convention.

<table>
<thead>
<tr>
<th>property</th>
<th>left</th>
<th>bottom</th>
<th>right</th>
<th>top</th>
</tr>
</thead>
<tbody>
<tr>
<td>axislabel angle</td>
<td>180</td>
<td>0</td>
<td>0</td>
<td>180</td>
</tr>
<tr>
<td>axislabel va</td>
<td>center</td>
<td>top</td>
<td>center</td>
<td>bottom</td>
</tr>
<tr>
<td>axislabel ha</td>
<td>right</td>
<td>center</td>
<td>right</td>
<td>center</td>
</tr>
</tbody>
</table>

Note that the text angles are actually relative to (90 + angle of the direction to the ticklabel), which gives 0 for bottom axis.

set_pad(pad)
Set the pad in points. Note that the actual pad will be the sum of the internal pad and the external pad (that are set automatically by the AxisArtist), and it only set the internal pad

class mpl_toolkits.axes_grid.axis_artist.TickLabels(**kwargs)
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.

get_texts_widths_heights_descents(renderer)
return a list of width, height, descent for ticklabels.

set_axis_direction(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].

<table>
<thead>
<tr>
<th>property</th>
<th>left</th>
<th>bottom</th>
<th>right</th>
<th>top</th>
</tr>
</thead>
<tbody>
<tr>
<td>ticklabels angle</td>
<td>90</td>
<td>0</td>
<td>-90</td>
<td>180</td>
</tr>
<tr>
<td>ticklabel va</td>
<td>center</td>
<td>baseline</td>
<td>center</td>
<td>baseline</td>
</tr>
<tr>
<td>ticklabel ha</td>
<td>right</td>
<td>center</td>
<td>right</td>
<td>center</td>
</tr>
</tbody>
</table>
Note that the text angles are actually relative to (90 + angle of the direction to the ticklabel), which gives 0 for bottom axis.
Part V

mplot3d
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.

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.

39.1 mplot3d tutorial
39.1.1 Getting started

An Axes3D object is created just like any other axes using the projection='3d' keyword. Create a new matplotlib.pyplot.Figure and add a new axes to it of type Axes3D:

```python
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
```

New in version 1.0.0: This approach is the preferred method of creating a 3D axes.

**Note:** Prior to version 1.0.0, the method of creating a 3D axes was different. For those using older versions of matplotlib, change `ax = fig.add_subplot(111, projection='3d')` to `ax = Axes3D(fig)`.

39.1.2 Line plots

Axes3D.plot(xs, ys, *args, **kwargs)
Plot 2D or 3D data.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xs, ys</td>
<td>X, y coordinates of vertices</td>
</tr>
<tr>
<td>zs</td>
<td>z value(s), either one for all points or one for each point.</td>
</tr>
<tr>
<td>zdir</td>
<td>Which direction to use as z ('x', 'y' or 'z') when plotting a 2D set.</td>
</tr>
</tbody>
</table>

Other arguments are passed on to plot()
### 39.1.3 Scatter plots

**Axes3D.scatter**\((xs, ys, zs=0, zdir='z', s=20, c='b', *args, **kwargs)\)

Create a scatter plot.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>xs, ys</strong></td>
<td>Positions of data points.</td>
</tr>
<tr>
<td><strong>zs</strong></td>
<td>Either an array of the same length as <strong>xs</strong> and <strong>ys</strong> or a single value to place all points in the same plane. Default is 0.</td>
</tr>
<tr>
<td><strong>zdir</strong></td>
<td>Which direction to use as z ('x', 'y' or 'z') when plotting a 2D set.</td>
</tr>
<tr>
<td><strong>s</strong></td>
<td>size in points^2. It is a scalar or an array of the same length as <strong>x</strong> and <strong>y</strong>.</td>
</tr>
<tr>
<td><strong>c</strong></td>
<td>a color. <strong>c</strong> can be a single color format string, or a sequence of color specifications of length <strong>N</strong>, or a sequence of <strong>N</strong> numbers to be mapped to colors using the <strong>cmap</strong> and <strong>norm</strong> specified via <strong>kwargs</strong> (see below). Note that <strong>c</strong> should not be a single numeric RGB or RGBA sequence because that is indistinguishable from an array of values to be colormapped. <strong>c</strong> can be a 2-D array in which the rows are RGB or RGBA, however.</td>
</tr>
</tbody>
</table>

Keyword arguments are passed on to **scatter()**.

Returns a **Patch3DCollection**.
39.1.4 Wireframe plots

Axes3D.plot_wireframe(X, Y, Z, *args, **kwargs)

Plot a 3D wireframe.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X, Y, Z</td>
<td>Data values as 2D arrays</td>
</tr>
<tr>
<td>rstride</td>
<td>Array row stride (step size)</td>
</tr>
<tr>
<td>cstride</td>
<td>Array column stride (step size)</td>
</tr>
</tbody>
</table>

Keyword arguments are passed on to LineCollection.

Returns a Line3DCollection
39.1.5 Surface plots

Axes3D.plot_surface(X, Y, Z, *args, **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.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X, Y, Z</td>
<td>Data values as 2D arrays</td>
</tr>
<tr>
<td>rstride</td>
<td>Array row stride (step size)</td>
</tr>
<tr>
<td>cstride</td>
<td>Array column stride (step size)</td>
</tr>
<tr>
<td>color</td>
<td>Color of the surface patches</td>
</tr>
<tr>
<td>cmap</td>
<td>A colormap for the surface patches.</td>
</tr>
<tr>
<td>facecolors</td>
<td>Face colors for the individual patches</td>
</tr>
<tr>
<td>norm</td>
<td>An instance of Normalize to map values to colors</td>
</tr>
<tr>
<td>vmin</td>
<td>Minimum value to map</td>
</tr>
<tr>
<td>vmax</td>
<td>Maximum value to map</td>
</tr>
<tr>
<td>shade</td>
<td>Whether to shade the facecolors</td>
</tr>
</tbody>
</table>

Other arguments are passed on to Poly3DCollection
Chapter 39. Matplotlib mplot3d toolkit
### 39.1.6 Tri-Surface plots

Axes3D.plot_trisurf(*args, **kwargs)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X, Y, Z</td>
<td>Data values as 1D arrays</td>
</tr>
<tr>
<td>color</td>
<td>Color of the surface patches</td>
</tr>
<tr>
<td>cmap</td>
<td>A colormap for the surface patches.</td>
</tr>
<tr>
<td>norm</td>
<td>An instance of Normalize to map values to colors</td>
</tr>
<tr>
<td>vmin</td>
<td>Minimum value to map</td>
</tr>
<tr>
<td>vmax</td>
<td>Maximum value to map</td>
</tr>
<tr>
<td>shade</td>
<td>Whether to shade the facecolors</td>
</tr>
</tbody>
</table>

The (optional) triangulation can be specified in one of two ways; either:

```python
plot_trisurf(triangulation, ...)
```

where triangulation is a `Triangulation` object, or:

```python
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:

\texttt{plot_trisurf(..., z)}

where \( z \) is the array of values to contour, one per point in the triangulation.

Other arguments are passed on to \texttt{Poly3DCollection}

\textbf{Examples:}
New in version 1.2.0: This plotting function was added for the v1.2.0 release.
39.1.7 Contour plots

Axes3D.contour(X, Y, Z, *args, **kwargs)
Create a 3D contour plot.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X, Y, Z</td>
<td>Data values as numpy.arrays</td>
</tr>
<tr>
<td>extend3d</td>
<td>Whether to extend contour in 3D (default: False)</td>
</tr>
<tr>
<td>stride</td>
<td>Stride (step size) for extending contour</td>
</tr>
<tr>
<td>zdir</td>
<td>The direction to use: x, y or z (default)</td>
</tr>
<tr>
<td>offset</td>
<td>If specified plot a projection of the contour lines on this position in plane normal to zdir</td>
</tr>
</tbody>
</table>

The positional and other keyword arguments are passed on to contour().

Returns a contour
Chapter 39. Matplotlib mplot3d toolkit
39.1.8 Filled contour plots

Axes3D.contourf(X, Y, Z, *args, **kwargs)

Create a 3D contourf plot.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X, Y, Z</td>
<td>Data values as numpy.arrays</td>
</tr>
<tr>
<td>zdir</td>
<td>The direction to use: x, y or z (default)</td>
</tr>
<tr>
<td>offset</td>
<td>If specified plot a projection of the filled contour on this position in plane normal to zdir</td>
</tr>
</tbody>
</table>

The positional and keyword arguments are passed on to contourf()

Returns a contourf

Changed in version 1.1.0: The zdir and offset kwargs were added.
Chapter 39. Matplotlib mplot3d toolkit
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.

### 39.1.9 Polygon plots

**Axes3D.add_collection3d(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
39.1.10 Bar plots

Axes3D.bar(left, height, zs=0, zdir='z', *args, **kwargs)

Add 2D bar(s).

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>left</td>
<td>The x coordinates of the left sides of the bars.</td>
</tr>
<tr>
<td>height</td>
<td>The height of the bars.</td>
</tr>
<tr>
<td>zs</td>
<td>Z coordinate of bars, if one value is specified they will all be placed at the same z.</td>
</tr>
<tr>
<td>zdir</td>
<td>Which direction to use as z (‘x’, ‘y’ or ‘z’) when plotting a 2D set.</td>
</tr>
</tbody>
</table>

Keyword arguments are passed onto bar().

Returns a Patch3DCollection
39.1.11 2D plots in 3D

39.1.12 Text

Axes3D.text(x, y, z, 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.
39.1.13 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.
Chapter 39. Matplotlib mplot3d toolkit
A tale of 2 subplots

39.2 mplot3d API
39.2.1 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!

Module containing Axes3D, an object which can plot 3D objects on a 2D matplotlib figure.

class mpl_toolkits.mplot3d.axes3d.Axes3D(fig, rect=None, *args, **kwargs)
    Bases: matplotlib.axes.Axes

    3D axes object.

    add_collection3d(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(cset, extend3d=False, stride=5, zdir='z', offset=None)

    add_contourf_set(cset, zdir='z', offset=None)

    auto_scale_xyz(X, Y, Z=None, had_data=None)

    autoscale(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: This function was added, but not tested. Please report any bugs.
**autoscale_view**(*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**(*left, height, zs=0, zdir='z', *args, **kwargs*)

Add 2D bar(s).

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>left</td>
<td>The x coordinates of the left sides of the bars.</td>
</tr>
<tr>
<td>height</td>
<td>The height of the bars.</td>
</tr>
<tr>
<td>zs</td>
<td>Z coordinate of bars, if one value is specified they will all be placed at the same z.</td>
</tr>
<tr>
<td>zdir</td>
<td>Which direction to use as z ('x', ‘y’ or ‘z’) when plotting a 2D set.</td>
</tr>
</tbody>
</table>

Keyword arguments are passed onto `bar()`.

Returns a `Patch3DCollection`

**bar3d**(*x, y, z, dx, dy, dz, color='b', zsort='average', *args, **kwargs*)

Generate a 3D bar, or multiple bars.

When generating multiple bars, x, y, z have to be arrays. dx, dy, dz can be arrays or scalars.

`color` can be:

- A single color value, 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

Keyword arguments are passed onto `Poly3DCollection()`

**can_pan**()

Return *True* if this axes supports the pan/zoom button functionality.

3D axes objects do not use the pan/zoom button.
**can** _**zoom**_ ()
Return *True* if this axes supports the zoom box button functionality.

3D axes objects do not use the zoom box button.

**cla**()
Clear axes

**clabel** (*args, **kwargs)
This function is currently not implemented for 3D axes. Returns *None*.

**contour**(*X, Y, Z, *args, **kwargs*)
Create a 3D contour plot.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>X, Y, Z</em></td>
<td>Data values as numpy.arrays</td>
</tr>
<tr>
<td><em>extend3d</em></td>
<td>Whether to extend contour in 3D (default: False)</td>
</tr>
<tr>
<td><em>stride</em></td>
<td>Stride (step size) for extending contour</td>
</tr>
<tr>
<td><em>zdir</em></td>
<td>The direction to use: x, y or z (default)</td>
</tr>
<tr>
<td><em>offset</em></td>
<td>If specified plot a projection of the contour lines on this position in plane normal to <em>zdir</em></td>
</tr>
</tbody>
</table>

The positional and other keyword arguments are passed on to *contour*()

Returns a **contour**

**contour3D**(*X, Y, Z, *args, **kwargs*)
Create a 3D contour plot.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>X, Y, Z</em></td>
<td>Data values as numpy.arrays</td>
</tr>
<tr>
<td><em>extend3d</em></td>
<td>Whether to extend contour in 3D (default: False)</td>
</tr>
<tr>
<td><em>stride</em></td>
<td>Stride (step size) for extending contour</td>
</tr>
<tr>
<td><em>zdir</em></td>
<td>The direction to use: x, y or z (default)</td>
</tr>
<tr>
<td><em>offset</em></td>
<td>If specified plot a projection of the contour lines on this position in plane normal to <em>zdir</em></td>
</tr>
</tbody>
</table>

The positional and other keyword arguments are passed on to *contour*()

Returns a **contour**

**contourf**(*X, Y, Z, *args, **kwargs*)
Create a 3D contourf plot.
<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X, Y, Z</td>
<td>Data values as numpy.arrays</td>
</tr>
<tr>
<td>zdir</td>
<td>The direction to use: x, y or z (default)</td>
</tr>
<tr>
<td>offset</td>
<td>If specified plot a projection of the filled contour on this position in plane normal to zdir</td>
</tr>
</tbody>
</table>

The positional and keyword arguments are passed on to contourf()

Returns a contourf

Changed in version 1.1.0: The zdir and offset kwargs were added.

**contourf3D(X, Y, Z, *args, **kwargs)**
Create a 3D contourf plot.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X, Y, Z</td>
<td>Data values as numpy.arrays</td>
</tr>
<tr>
<td>zdir</td>
<td>The direction to use: x, y or z (default)</td>
</tr>
<tr>
<td>offset</td>
<td>If specified plot a projection of the filled contour on this position in plane normal to zdir</td>
</tr>
</tbody>
</table>

The positional and keyword arguments are passed on to contourf()

Returns a contourf

Changed in version 1.1.0: The zdir and offset kwargs were added.

**convert_zunits(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()**
Disable mouse button callbacks.

**draw(renderer)**

**format_coord(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(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()**
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()
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()

get_axisbelow()
Get whether axis below is true or not.
For axes3d objects, this will always be True
New in version 1.1.0: This function was added for completeness.

get_children()

get_frame_on()
Get whether the 3D axes panels are drawn
New in version 1.1.0.

get_proj()
Create the projection matrix from the current viewing position.
elev stores the elevation angle in the z plane azim stores the azimuth angle in the x,y plane
dist is the distance of the eye viewing point from the object point.

get_w_lims()
Get 3D world limits.

get_xlim()
Get the x-axis range \([left, right]\)
Changed in version 1.1.0: This function now correctly refers to the 3D x-limits

get_xlim3d()
Get the x-axis range \([left, right]\)
Changed in version 1.1.0: This function now correctly refers to the 3D x-limits

get_ylim()
Get the y-axis range \([bottom, top]\)
Changed in version 1.1.0: This function now correctly refers to the 3D y-limits.

get_ylim3d()
Get the y-axis range \([bottom, top]\)
Changed in version 1.1.0: This function now correctly refers to the 3D y-limits.

get_zbound()
Returns the z-axis numerical bounds where:
lowerBound < upperBound
New in version 1.1.0: This function was added, but not tested. Please report any bugs.
get_zlabel()
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()
Get 3D z limits.

get_zlim3d()
Get 3D z limits.

get_zmajorticklabels()
Get the ztick labels as a list of Text instances

New in version 1.1.0.

get_zminorticklabels()
Get the ztick labels as a list of Text instances

Note: Minor ticks are not supported. This function was added only for completeness.

New in version 1.1.0.

get_zscale()

get_zticklabels(minor=False)
Get ztick labels as a list of Text instances. See matplotlib.axes.Axes.get_yticklabels() for more details.

Note: Minor ticks are not supported.

New in version 1.1.0.

get_zticklines()
Get ztick lines as a list of Line2D instances. Note that this function is provided merely for completeness. These lines are re-calculated as the display changes.

New in version 1.1.0.

get_zticks(minor=False)
Return the z ticks as a list of locations. See matplotlib.axes.Axes.get_yticks() for more details.

Note: Minor ticks are not supported.

New in version 1.1.0.

grid(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.
have_units()

Return True if units are set on the x, y, or z axes

invert_zaxis()

Invert the z-axis.

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

locator_params(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(*args, **kw)

Convenience method to set or retrieve autoscaling margins.

signatures:: margins()

returns xmargin, ymargin, zmargin

margins(margin)

margins(xmargin, ymargin, zmargin)

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 argument specifies xmargin, ymargin and zmargin. 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: This function was added, but not tested. Please report any bugs.

mouse_init(rotate_btn=1, zoom_btn=3)

Initializes mouse button callbacks to enable 3D rotation of the axes. Also optionally sets the mouse buttons for 3D rotation and zooming.
### Argument Description

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rotate_btn</td>
<td>The integer or list of integers specifying which mouse button or buttons to use for 3D rotation of the axes. Default = 1.</td>
</tr>
<tr>
<td>zoom_btn</td>
<td>The integer or list of integers specifying which mouse button or buttons to use to zoom the 3D axes. Default = 3.</td>
</tr>
</tbody>
</table>

**name** = ‘3d’

**plot**(*xs, ys, *args, **kwargs*)

Plot 2D or 3D data.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xs, ys</td>
<td>X, y coordinates of vertices</td>
</tr>
<tr>
<td>zs</td>
<td>z value(s), either one for all points or one for each point.</td>
</tr>
<tr>
<td>zdir</td>
<td>Which direction to use as z (‘x’, ‘y’ or ‘z’) when plotting a 2D set.</td>
</tr>
</tbody>
</table>

Other arguments are passed on to plot()

**plot3D**(*xs, ys, *args, **kwargs*)

Plot 2D or 3D data.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xs, ys</td>
<td>X, y coordinates of vertices</td>
</tr>
<tr>
<td>zs</td>
<td>z value(s), either one for all points or one for each point.</td>
</tr>
<tr>
<td>zdir</td>
<td>Which direction to use as z (‘x’, ‘y’ or ‘z’) when plotting a 2D set.</td>
</tr>
</tbody>
</table>

Other arguments are passed on to plot()

**plot_surface**(*X, Y, Z, *args, **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.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X, Y, Z</td>
<td>Data values as 2D arrays</td>
</tr>
<tr>
<td>rstride</td>
<td>Array row stride (step size)</td>
</tr>
<tr>
<td>cstride</td>
<td>Array column stride (step size)</td>
</tr>
<tr>
<td>color</td>
<td>Color of the surface patches</td>
</tr>
<tr>
<td>cmap</td>
<td>A colormap for the surface patches.</td>
</tr>
<tr>
<td>facecolors</td>
<td>Face colors for the individual patches</td>
</tr>
<tr>
<td>norm</td>
<td>An instance of Normalize to map values to colors</td>
</tr>
<tr>
<td>vmin</td>
<td>Minimum value to map</td>
</tr>
<tr>
<td>vmax</td>
<td>Maximum value to map</td>
</tr>
<tr>
<td>shade</td>
<td>Whether to shade the facecolors</td>
</tr>
</tbody>
</table>

Other arguments are passed on to Poly3DCollection

**plot_trisurf**(*args, **kwargs*)
<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X, Y, Z</td>
<td>Data values as 1D arrays</td>
</tr>
<tr>
<td>color</td>
<td>Color of the surface patches</td>
</tr>
<tr>
<td>cmap</td>
<td>A colormap for the surface patches.</td>
</tr>
<tr>
<td>norm</td>
<td>An instance of Normalize to map values to colors</td>
</tr>
<tr>
<td>vmin</td>
<td>Minimum value to map</td>
</tr>
<tr>
<td>vmax</td>
<td>Maximum value to map</td>
</tr>
<tr>
<td>shade</td>
<td>Whether to shade the facecolors</td>
</tr>
</tbody>
</table>

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 an 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.

Other arguments are passed on to `Poly3DCollection`.

**Examples:**
New in version 1.2.0: This plotting function was added for the v1.2.0 release.

**plot_wireframe**(X, Y, Z, *args, **kwargs)

Plot a 3D wireframe.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X, Y, Z</td>
<td>Data values as 2D arrays</td>
</tr>
<tr>
<td>rstride</td>
<td>Array row stride (step size)</td>
</tr>
<tr>
<td>cstride</td>
<td>Array column stride (step size)</td>
</tr>
</tbody>
</table>

Keyword arguments are passed on to LineCollection.

Returns a Line3DCollection.

**scatter**(xs, ys, zs=0, zdir='z', s=20, c='b', *args, **kwargs)

Create a scatter plot.
### Argument Description

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$xs, ys$</td>
<td>Positions of data points.</td>
</tr>
<tr>
<td>$zs$</td>
<td>Either an array of the same length as $xs$ and $ys$ or a single value to place all points in the same plane. Default is 0.</td>
</tr>
<tr>
<td>$zdir$</td>
<td>Which direction to use as $z$ (‘x’, ‘y’ or ‘z’) when plotting a 2D set.</td>
</tr>
<tr>
<td>$s$</td>
<td>Size in points$^2$. It is a scalar or an array of the same length as $x$ and $y$.</td>
</tr>
<tr>
<td>$c$</td>
<td>A color. $c$ can be a single color format string, or a sequence of color specifications of length $N$, or a sequence of $N$ numbers to be mapped to colors using the $cmap$ and $norm$ specified via kwargs (see below). 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. $c$ can be a 2-D array in which the rows are RGB or RGBA, however.</td>
</tr>
</tbody>
</table>

Keyword arguments are passed on to `scatter()`. Returns a `Patch3DCollection`.

**scatter3D($xs$, $ys$, $zs=0$, $zdir='z'$, $s=20$, $c='b'$, *args, **kwargs)**

Create a scatter plot.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$xs, ys$</td>
<td>Positions of data points.</td>
</tr>
<tr>
<td>$zs$</td>
<td>Either an array of the same length as $xs$ and $ys$ or a single value to place all points in the same plane. Default is 0.</td>
</tr>
<tr>
<td>$zdir$</td>
<td>Which direction to use as $z$ (‘x’, ‘y’ or ‘z’) when plotting a 2D set.</td>
</tr>
<tr>
<td>$s$</td>
<td>Size in points$^2$. It is a scalar or an array of the same length as $x$ and $y$.</td>
</tr>
<tr>
<td>$c$</td>
<td>A color. $c$ can be a single color format string, or a sequence of color specifications of length $N$, or a sequence of $N$ numbers to be mapped to colors using the $cmap$ and $norm$ specified via kwargs (see below). 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. $c$ can be a 2-D array in which the rows are RGB or RGBA, however.</td>
</tr>
</tbody>
</table>

Keyword arguments are passed on to `scatter()`. Returns a `Patch3DCollection`.

**set_autoscale_on($b$)**

Set whether autoscaling is applied on plot commands

accepts: [ $True$ | $False$ ]

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

**set_autoscalez_on($b$)**

Set whether autoscaling for the z-axis is applied on plot commands

accepts: [ $True$ | $False$ ]
New in version 1.1.0: This function was added, but not tested. Please report any bugs.

```python
set_axis_off()
```

```python
set_axis_on()
```

```python
set_axisbelow(b)
```
Set whether the axis ticks and gridlines are above or below most artists

For axes3d objects, this will ignore any settings and just use `True`

ACCEPTS: `[ True | False ]`

New in version 1.1.0: This function was added for completeness.

```python
set_frame_on(b)
```
Set whether the 3D axes panels are drawn

ACCEPTS: `[ True | False ]`

New in version 1.1.0.

```python
set_title(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:

```python
{ 'fontsize': rcParams['axes.titlesize'],
  'verticalalignment': 'baseline',
  'horizontalalignment': loc }
```

`loc` : `{ ‘center’, ‘left’, ‘right’ },` str, optional

Which title to set, defaults to ‘center’

**Returns**

`text` : Text

The matplotlib text instance representing the title

**Other Parameters**

Other keyword arguments are text properties, see `:class:~matplotlib.text.Text` for a list of valid text properties.

```python
set_top_view()
```
**set_xlim**(*left=None, right=None, emit=True, auto=False, **kw*)
Set 3D x limits.
See `matplotlib.axes.Axes.set_xlim()` for full documentation.

**set_xlim3d**(*left=None, right=None, emit=True, auto=False, **kw*)
Set 3D x limits.
See `matplotlib.axes.Axes.set_xlim()` for full documentation.

**set_xscale**(*value, **kwargs*)
Call signature:

```
set_xscale(value)
```
Set the scaling of the x-axis: ‘linear’ | ‘log’ | ‘symlog’

**`ACCEPTS`**:['linear' | ‘log’ | ‘symlog’]

Different kwargs are accepted, depending on the scale: ‘linear’

- ‘log’
  - `basex/basey`: The base of the logarithm
  - `nonposx/nonposy`: ['mask' | ‘clip’] non-positive values in x or y can be masked as invalid, or clipped to a very small positive number
  - `subsx/subsy`: Where to place the subticks between each major tick. Should be a sequence of integers. 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.

- ‘symlog’
  - `basex/basey`: The base of the logarithm
  - `linthreshx/linthreshy`: The range (-x, x) within which the plot is linear (to avoid having the plot go to infinity around zero).
  - `subsx/subsy`: Where to place the subticks between each major tick. Should be a sequence of integers. 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.
  - `linscalex/linscaley`: This allows the linear range (\(-linsthresh\) to \(linsthresh\)) 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.

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

**set_ylim**(*bottom=None, top=None, emit=True, auto=False, **kw*)
Set 3D y limits.
See `matplotlib.axes.Axes.set_ylim()` for full documentation.

**set_ylim3d**(*bottom=None, top=None, emit=True, auto=False, **kw*)
Set 3D y limits.
See `matplotlib.axes.Axes.set_yscale()` for full documentation.

**set_yscale**(value, **kwargs)

Call signature:

```python
set_yscale(value)
```

Set the scaling of the y-axis: ‘linear’ | ‘log’ | ‘symlog’

ACCEPTS: ['linear' | ‘log’ | ‘symlog’]

Different kwargs are accepted, depending on the scale: ‘linear’

- ‘log’
  - `basex`/`basey`: The base of the logarithm
  - `nonposx`/`nonposy`: ['mask' | 'clip'] non-positive values in x or y can be masked as invalid, or clipped to a very small positive number
  - `subsx`/`subsy`: Where to place the subticks between each major tick. Should be a sequence of integers. 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.

- ‘symlog’
  - `basex`/`basey`: The base of the logarithm
  - `linthreshx`/`linthreshy`: The range (-x, x) within which the plot is linear (to avoid having the plot go to infinity around zero).
  - `subsx`/`subsy`: Where to place the subticks between each major tick. Should be a sequence of integers. 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.

- `linscalex`/`linscaley`: This allows the linear range (-`linthresh` to `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.

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

**set_zbound**(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 `_autoscaleZon` attribute.

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

**set_zlabel**(zlabel, fontdict=None, labelpad=None, **kwargs)

Set zlabel. See doc for `set_ylabel()` for description.

---

**Note**: Currently, `labelpad` does not have an effect on the labels.

**set_zlim**(bottom=None, top=None, emit=True, auto=False, **kw)

Set 3D z limits.
**set_zlim3d**(bottom=None, top=None, emit=True, auto=False, **kw)

Set 3D z limits.

See `matplotlib.axes.Axes.set_ylim()` for full documentation

**set_zmargin**(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: This function was added, but not tested. Please report any bugs.

**set_zscale**(value, **kwargs)

call signature:

```
set_zscale(value)
```

Set the scaling of the z-axis: ‘linear’ | ‘log’ | ‘symlog’

ACCEPTS: ['linear' | 'log' | 'symlog']

Different kwargs are accepted, depending on the scale: ‘linear’

‘log’

- **base**: The base of the logarithm
- **nonposx**/**nonposy**: [‘mask’ | ‘clip’] non-positive values in x or y can be masked as invalid, or clipped to a very small positive number
- **subsx**/**subsy**: Where to place the subticks between each major tick. Should be a sequence of integers. 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.

‘symlog’

- **base**: The base of the logarithm
- **linthreshx**/**linthreshy**: The range (-x, x) within which the plot is linear (to avoid having the plot go to infinity around zero).
- **subsx**/**subsy**: Where to place the subticks between each major tick. Should be a sequence of integers. 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.

**linscalex**/**linscaley**: This allows the linear range (-linthresh to 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.

**Note:** Currently, Axes3D objects only supports linear scales. Other scales may or may not work, and support for these is improving with each release.

New in version 1.1.0: This function was added, but not tested. Please report any bugs.
set_zticklabels(*args, **kwargs)
Set z-axis tick labels. See matplotlib.axes.Axes.set_yticklabels() for more details.

**Note:** Minor ticks are not supported by Axes3D objects.

New in version 1.1.0.

set_zticks(*args, **kwargs)
Set z-axis tick locations. See matplotlib.axes.Axes.set_yticks() for more details.

**Note:** Minor ticks are not supported.

New in version 1.1.0.

text(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(x, y, s, fontdict=None, withdash=False, **kwargs)
Add text to the axes.

Add text in string s to axis at location x, y, data coordinates.

**Parameters**

- **s** : string
text
- **x, y** : scalars
data coordinates
- **fontdict** : dictionary, optional, default: None
A dictionary to override the default text properties. If fontdict is None, the defaults are determined by your rc parameters.
- **withdash** : boolean, optional, default: False
Creates a TextWithDash instance instead of a Text instance.

**Other Parameters**

- **kwargs** : Text properties.
  Other miscellaneous text parameters.

**Examples**

Individual keyword arguments can be used to override any given parameter:

```python
>>> 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:

```python
>>> 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:

```python
>>> text(x, y, s, bbox=dict(facecolor='red', alpha=0.5))
```

**text3D** *(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** *(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:** While this function is currently implemented, the core part of the `Axes3D` object may ignore some of these settings. Future releases will fix this. Priority will be given to those who file bugs.

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

**ticklabel_format** (**kwargs)**

Convenience method for manipulating the ScalarFormatter used by default for linear axes in `Axes3D` objects.

See `matplotlib.axes.Axes.ticklabel_format()` for full documentation. Note that this version applies to all three axes of the `Axes3D` object. Therefore, the `axis` argument will also accept a value of ‘z’ and the value of ‘both’ will apply to all three axes.

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

**tricontour** *(X, Y, Z, *args, **kwargs)*

Create a 3D contour plot.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X, Y, Z</td>
<td>Data values as numpy.arrays</td>
</tr>
<tr>
<td>extend3d</td>
<td>Whether to extend contour in 3D (default: False)</td>
</tr>
<tr>
<td>stride</td>
<td>Stride (step size) for extending contour</td>
</tr>
<tr>
<td>zdir</td>
<td>The direction to use: x, y or z (default)</td>
</tr>
<tr>
<td>offset</td>
<td>If specified plot a projection of the contour lines on this position in plane normal to zdir</td>
</tr>
</tbody>
</table>

Other keyword arguments are passed on to `tricontour()`
Returns a `contour`  
New in version 1.1.0.

**tricontourf**(*X, Y, Z, offset=*, zdir='z', *args, **kwargs*)
Create a 3D contourf plot.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X, Y, Z</td>
<td>Data values as numpy.arrays</td>
</tr>
<tr>
<td>zdir</td>
<td>The direction to use: x, y or z (default)</td>
</tr>
<tr>
<td>offset</td>
<td>If specified plot a projection of the contour lines on this position in plane normal to zdir</td>
</tr>
</tbody>
</table>

Other keyword arguments are passed on to `tricontour()`

Returns a `contour`  
New in version 1.1.0.

**tunit_cube**(vals=*, M=*)

**tunit_edges**(vals=*, M=*)

**unit_cube**(vals=*)

**update_datalim**(xys, **kwargs)

**view_init**(elev=*, azim=*)
Set the elevation and azimuth of the axes.
This can be used to rotate the axes programatically.
‘elev’ stores the elevation angle in the z plane. ‘azim’ stores the azimuth angle in the x,y plane.
if elev or azim are None (default), then the initial value is used which was specified in the Axes3D constructor.

**zaxis_date**(tz=)
Sets up z-axis ticks and labels that treat the z data as dates.
tz is a timezone string or tzinfo instance. Defaults to rc value.

**zaxis_inverted**()
Returns True if the z-axis is inverted.

New in version 1.1.0: This function was added, but not tested. Please report any bugs.
New in version 1.1.0: This function was added, but not tested. Please report any bugs.

mpl_toolkits.mplot3d.axes3d.get_test_data(delta=0.05)
Return a tuple X, Y, Z with a test data set.

mpl_toolkits.mplot3d.axes3d.unit_bbox()

39.2.2 axis3d

Note: Historically, axis3d has suffered from having hard-coded constants controlling the look and feel of the 3D plot. This precluded user level adjustments such as label spacing, font colors and panel colors. For version 1.1.0, these constants have been consolidated into a single private member dictionary, self._axinfo, for the axis object. This is intended only as a stop-gap measure to allow user-level customization, but it is not intended to be permanent.

class mpl_toolkits.mplot3d.axis3d.Axis(adir, v_intervalx, d_intervalx, axes, *args, **kwargs)

Bases: matplotlib.axis.XAxis

draw(renderer)

draw_pane(renderer)

get_major_ticks(numticks=None)

get_rotate_label(text)

get_tick_positions()

get_tightbbox(renderer)

get_view_interval()
    return the Interval instance for this 3d axis view limits

init3d()

set_pane_color(color)
    Set pane color to a RGBA tuple

set_pane_pos(xys)

set_rotate_label(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.
set_view_interval(vmin, vmax, ignore=False)

class mpl_toolkits.mplot3d.axis3d.XAxis(adir, v_intervalx, d_intervalx, axes, *args, **kwargs)

get_data_interval()

Bases: mpl_toolkits.mplot3d.axis3d.Axis

class mpl_toolkits.mplot3d.axis3d.YAxis(adir, v_intervalx, d_intervalx, axes, *args, **kwargs)

get_data_interval()

Bases: mpl_toolkits.mplot3d.axis3d.Axis

class mpl_toolkits.mplot3d.axis3d.ZAxis(adir, v_intervalx, d_intervalx, axes, *args, **kwargs)

get_data_interval()

Bases: mpl_toolkits.mplot3d.axis3d.Axis

mpl_toolkits.mplot3d.axis3d.get_flip_min_max(coord, index, mins, maxs)

mpl_toolkits.mplot3d.axis3d.move_from_center(coord, centers, deltas, axmask=(True, True, True))

Return a coordinate that is moved by “deltas” away from the center.

mpl_toolkits.mplot3d.axis3d.tick_update_position(tick, tickxs, tickys, labelpos)

Update tick line and label position and style.

39.2.3 art3d

Module containing 3D artist code and functions to convert 2D artists into 3D versions which can be added to an Axes3D.

class mpl_toolkits.mplot3d.art3d.Line3D(xs, ys, zs, *args, **kwargs)

Bases: matplotlib.lines.Line2D

3D line object.

Keyword arguments are passed onto Line2D().

draw(renderer)

set_3d_properties(zs=0, zdir='z')

class mpl_toolkits.mplot3d.art3d.Line3DCollection(segments, *args, **kwargs)

Bases: matplotlib.collections.LineCollection

A collection of 3D lines.
Keyword arguments are passed onto `LineCollection()`.

```python
do_3d_projection(renderer)
   Project the points according to renderer matrix.

draw(renderer, project=False)
```

```python
set_segments(segments)
   Set 3D segments

set_sort_zpos(val)
   Set the position to use for z-sorting.
```

```python
class mpl_toolkits.mplot3d.art3d.Patch3D(*args, **kwargs)
   Bases: matplotlib.patches.Patch

3D patch object.

do_3d_projection(renderer)

draw(renderer)

get_facecolor()

get_path()

set_3d_properties(verts, zs=0, zdir='z')
```

```python
class mpl_toolkits.mplot3d.art3d.Patch3DCollection(*args, **kwargs)
   Bases: matplotlib.collections.PatchCollection

A collection of 3D patches.

do_3d_projection(renderer)

draw(renderer)

set_3d_properties(zs, zdir)

set_sort_zpos(val)
   Set the position to use for z-sorting.
```

```python
class mpl_toolkits.mplot3d.art3d.PathPatch3D(path, **kwargs)
   Bases: mpl_toolkits.mplot3d.art3d.Path3D

3D PathPatch object.

do_3d_projection(renderer)
```
```python
set_3d_properties(path, zs=0, zdir='z')
```

class mpl_toolkits.mplot3d.art3d.Poly3DCollection(verts, *args, **kwargs):
    Bases: matplotlib.collections.PolyCollection

    A collection of 3D polygons.

    Create a Poly3DCollection.

    verts should contain 3D coordinates.

    Keyword arguments: zsort, see set_zsort for options.

    Note that this class does a bit of magic with the _facecolors and _edgecolors properties.

    do_3d_projection(renderer)
        Perform the 3D projection for this object.

    draw(renderer)

    get_edgecolor()

    get_edgecolors()

    get_facecolor()

    get_facecolors()

    get_vector(segments3d)
        Optimize points for projection

    set_3d_properties()

    set_edgecolor(colors)

    set_edgecolors(colors)

    set_facecolor(colors)

    set_facecolors(colors)

    set_sort_zpos(val)
        Set the position to use for z-sorting.

    set_verts(verts, closed=True)
        Set 3D vertices.
set_zsort(zsort)

Set z-sorting behaviour: boolean: if True use default ‘average’ string: ‘average’, ‘min’ or ‘max’

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 (in the future) direction.
x, y, z Position of text text Text string to display zdir Direction of text
Keyword arguments are passed onto Text().

draw(renderer)

set_3d_properties(z=0, zdir='z')

mpl_toolkits.mplot3d.art3d.get_colors(c, num)
Stretch the color argument to provide the required number num

mpl_toolkits.mplot3d.art3d.get_dir_vector(zdir)

mpl_toolkits.mplot3d.art3d.get_patch_verts(patch)
Return a list of vertices for the path of a patch.

mpl_toolkits.mplot3d.art3d.iscolor(c)

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.

mpl_toolkits.mplot3d.art3d.line_2d_to_3d(line, zs=0, zdir='z')
Convert a 2D line to 3D.

mpl_toolkits.mplot3d.art3d.line_collection_2d_to_3d(col, zs=0, zdir='z')
Convert a LineCollection to a Line3DCollection object.

mpl_toolkits.mplot3d.art3d.norm_angle(a)
Return angle between -180 and +180

mpl_toolkits.mplot3d.art3d.norm_text_angle(a)
Return angle between -90 and +90

mpl_toolkits.mplot3d.art3d.patch_2d_to_3d(patch, z=0, zdir='z')
Convert a Patch to a Patch3D object.

mpl_toolkits.mplot3d.art3d.patch_collection_2d_to_3d(col, zs=0, zdir='z')
Convert a PatchCollection to a Patch3DCollection object.

mpl_toolkits.mplot3d.art3d.path_to_3d_segment(path, zs=0, zdir='z')
Convert a path to a 3D segment.
mpl_toolkits.mplot3d.art3d.pathpatch_2d_to_3d(pathpatch, z=0, zdir='z')
Convert a PathPatch to a PathPatch3D object.

mpl_toolkits.mplot3d.art3d.paths_to_3d_segments(paths, zs=0, zdir='z')
Convert paths from a collection object to 3D segments.

mpl_toolkits.mplot3d.art3d.poly_collection_2d_to_3d(col, zs=0, zdir='z')
Convert a PolyCollection to a Poly3DCollection object.

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

mpl_toolkits.mplot3d.art3d.text_2d_to_3d(obj, z=0, zdir='z')
Convert a Text to a Text3D object.

mpl_toolkits.mplot3d.art3d.zalpha(colors, zs)
Modify the alphas of the color list according to depth

39.2.4 proj3d

Various transforms used for by the 3D code

mpl_toolkits.mplot3d.proj3d.inv_transform(xs, ys, zs, M)

mpl_toolkits.mplot3d.proj3d.line2d(p0, p1)
Return 2D equation of line in the form ax+by+c = 0

mpl_toolkits.mplot3d.proj3d.line2d_dist(l, p)
Distance from line to point line is a tuple of coefficients a,b,c

mpl_toolkits.mplot3d.proj3d.line2d_seg_dist(p1, p2, p0)
distance(s) from line defined by p1 - p2 to point(s) p0
p0[0] = x(s) p0[1] = y(s)
intersection point p = p1 + u*(p2-p1) and intersection point lies within segment if u is between 0 and 1

mpl_toolkits.mplot3d.proj3d.mod(v)
3d vector length

mpl_toolkits.mplot3d.proj3d.persp_transformation(zfront, zback)

mpl_toolkits.mplot3d.proj3d.proj_points(points, M)

mpl_toolkits.mplot3d.proj3d.proj_trans_clip_points(points, M)

mpl_toolkits.mplot3d.proj3d.proj_trans_points(points, M)
mpl_toolkits.mplot3d.proj3d.proj_transform(xs, ys, zs, M)
    Transform the points by the projection matrix

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

mpl_toolkits.mplot3d.proj3d.proj_transform_vec(vec, M)

mpl_toolkits.mplot3d.proj3d.proj_transform_vec_clip(vec, M)

mpl_toolkits.mplot3d.proj3d.rot_x(V, alpha)

mpl_toolkits.mplot3d.proj3d.test_lines_dists()

mpl_toolkits.mplot3d.proj3d.test_proj()

mpl_toolkits.mplot3d.proj3d.test_proj_draw_axes(M, s=1)

mpl_toolkits.mplot3d.proj3d.test_proj_make_M(E=None)

mpl_toolkits.mplot3d.proj3d.test_rot()

mpl_toolkits.mplot3d.proj3d.test_world()

mpl_toolkits.mplot3d.proj3d.transform(xs, ys, zs, M)
    Transform the points by the projection matrix

mpl_toolkits.mplot3d.proj3d.vec_pad_ones(xs, ys, zs)

mpl_toolkits.mplot3d.proj3d.view_transformation(E, R, V)

mpl_toolkits.mplot3d.proj3d.world_transformation(xmin, xmax, ymin, ymax, zmin, zmax)

### 39.3 mplot3d FAQ

#### 39.3.1 How is mplot3d different from MayaVi?

MayaVi2 is a very powerful and featureful 3D graphing library. For advanced 3D scenes and excellent rendering capabilities, it is highly recommended to use MayaVi2.
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.

### 39.3.2 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) cannot 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.

### 39.3.3 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 `axis3d` section of the mplot3d API documentation for more information.
Part VI

Toolkits
Toolkits are collections of application-specific functions that extend matplotlib.
BASEMAP (NOT DISTRIBUTED WITH MATPLOTLIB)

Plots data on map projections, with continental and political boundaries, see basemap docs.
CARTOPY (NOT DISTRIBUTED WITH MATPLOTLIB)

An alternative mapping library written for matplotlib v1.2 and beyond. 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:
mpl_toolkits.gtktools provides some utilities for working with GTK. This toolkit ships with matplotlib, but requires pygtk.
mpl_toolkits.exceltools provides some utilities for working with Excel. This toolkit ships with matplotlib, but requires xlwt.
mpl_toolkits.natgrid is an interface to natgrid C library for gridding irregularly spaced data. This requires a separate installation of the natgrid toolkit from the sourceforge download page.
mpl_toolkits.mplot3d provides some basic 3D plotting (scatter, surf, line, mesh) tools. Not the fastest or feature complete 3D library out there, but ships with matplotlib and thus may be a lighter weight solution for some use cases.
The matplotlib AxesGrid toolkit is a collection of helper classes to ease displaying multiple images in matplotlib. The AxesGrid toolkit is distributed with matplotlib source.
Part VII

The Matplotlib API
### PLOTTING COMMANDS SUMMARY

matplotlib.pyplot.plotting()

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>acorr</td>
<td>Plot the autocorrelation of x.</td>
</tr>
<tr>
<td>annotate</td>
<td>Create an annotation: a piece of text referring to a data point.</td>
</tr>
<tr>
<td>arrow</td>
<td>Add an arrow to the axes.</td>
</tr>
<tr>
<td>autoscale</td>
<td>Autoscale the axis view to the data (toggle).</td>
</tr>
<tr>
<td>axes</td>
<td>Add an axes to the figure.</td>
</tr>
<tr>
<td>axhline</td>
<td>Add a horizontal line across the axis.</td>
</tr>
<tr>
<td>axhspan</td>
<td>Add a horizontal span (rectangle) across the axis.</td>
</tr>
<tr>
<td>axis</td>
<td>Convenience method to get or set axis properties.</td>
</tr>
<tr>
<td>axvline</td>
<td>Add a vertical line across the axes.</td>
</tr>
<tr>
<td>axvspan</td>
<td>Add a vertical span (rectangle) across the axes.</td>
</tr>
<tr>
<td>bar</td>
<td>Make a bar plot.</td>
</tr>
<tr>
<td>barbs</td>
<td>Plot a 2-D field of barbs.</td>
</tr>
<tr>
<td>barh</td>
<td>Make a horizontal bar plot.</td>
</tr>
<tr>
<td>box</td>
<td>Turn the axes box on or off.</td>
</tr>
<tr>
<td>boxplot</td>
<td>Make a box and whisker plot.</td>
</tr>
<tr>
<td>broken_barh</td>
<td>Plot horizontal bars.</td>
</tr>
<tr>
<td>cla</td>
<td>Clear the current axes.</td>
</tr>
<tr>
<td>clabel</td>
<td>Label a contour plot.</td>
</tr>
<tr>
<td>clf</td>
<td>Clear the current figure.</td>
</tr>
<tr>
<td>clim</td>
<td>Set the color limits of the current image.</td>
</tr>
<tr>
<td>close</td>
<td>Close a figure window.</td>
</tr>
<tr>
<td>cohere</td>
<td>Plot the coherence between x and y.</td>
</tr>
<tr>
<td>colorbar</td>
<td>Add a colorbar to a plot.</td>
</tr>
<tr>
<td>contour</td>
<td>Plot contours.</td>
</tr>
<tr>
<td>contourf</td>
<td>Plot contours.</td>
</tr>
<tr>
<td>csd</td>
<td>Plot cross-spectral density.</td>
</tr>
<tr>
<td>delaxes</td>
<td>Remove an axes from the current figure.</td>
</tr>
<tr>
<td>draw</td>
<td>Redraw the current figure.</td>
</tr>
<tr>
<td>errorbar</td>
<td>Plot an errorbar graph.</td>
</tr>
<tr>
<td>eventplot</td>
<td>Plot identical parallel lines at specific positions.</td>
</tr>
</tbody>
</table>

Continued on next page
Table 47.1 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>figimage</td>
<td>Adds a non-resampled image to the figure.</td>
</tr>
<tr>
<td>figlegend</td>
<td>Place a legend in the figure.</td>
</tr>
<tr>
<td>figtext</td>
<td>Add text to figure.</td>
</tr>
<tr>
<td>figure</td>
<td>Creates a new figure.</td>
</tr>
<tr>
<td>fill</td>
<td>Plot filled polygons.</td>
</tr>
<tr>
<td>fill_between</td>
<td>Make filled polygons between two curves.</td>
</tr>
<tr>
<td>fill_betweenex</td>
<td>Make filled polygons between two horizontal curves.</td>
</tr>
<tr>
<td>findobj</td>
<td>Find artist objects.</td>
</tr>
<tr>
<td>gca</td>
<td>Return the current axis instance.</td>
</tr>
<tr>
<td>gcf</td>
<td>Return a reference to the current figure.</td>
</tr>
<tr>
<td>gci</td>
<td>Get the current colorable artist.</td>
</tr>
<tr>
<td>get_figlabels</td>
<td>Return a list of existing figure labels.</td>
</tr>
<tr>
<td>get_fignums</td>
<td>Return a list of existing figure numbers.</td>
</tr>
<tr>
<td>grid</td>
<td>Turn the axes grids on or off.</td>
</tr>
<tr>
<td>hexbin</td>
<td>Make a hexagonal binning plot.</td>
</tr>
<tr>
<td>hist</td>
<td>Plot a histogram.</td>
</tr>
<tr>
<td>hist2d</td>
<td>Make a 2D histogram plot.</td>
</tr>
<tr>
<td>hlines</td>
<td>Plot horizontal lines.</td>
</tr>
<tr>
<td>hold</td>
<td>Set the hold state.</td>
</tr>
<tr>
<td>imread</td>
<td>Read an image from a file into an array.</td>
</tr>
<tr>
<td>imsave</td>
<td>Save an array as in image file.</td>
</tr>
<tr>
<td>imshow</td>
<td>Display an image on the axes.</td>
</tr>
<tr>
<td>ion</td>
<td>Turn interactive mode off.</td>
</tr>
<tr>
<td>ioff</td>
<td>Turn interactive mode on.</td>
</tr>
<tr>
<td>ishold</td>
<td>Return the hold status of the current axes.</td>
</tr>
<tr>
<td>isinteractive</td>
<td>Return status of interactive mode.</td>
</tr>
<tr>
<td>legend</td>
<td>Place a legend on the current axes.</td>
</tr>
<tr>
<td>locator_params</td>
<td>Control behavior of tick locators.</td>
</tr>
<tr>
<td>loglog</td>
<td>Make a plot with log scaling on both the x and y axis.</td>
</tr>
<tr>
<td>margins</td>
<td>Set or retrieve autoscaling margins.</td>
</tr>
<tr>
<td>matshow</td>
<td>Display an array as a matrix in a new figure window.</td>
</tr>
<tr>
<td>minorticks_off</td>
<td>Remove minor ticks from the current plot.</td>
</tr>
<tr>
<td>minorticks_on</td>
<td>Display minor ticks on the current plot.</td>
</tr>
<tr>
<td>over</td>
<td>Call a function with hold(True).</td>
</tr>
<tr>
<td>pause</td>
<td>Pause for interval seconds.</td>
</tr>
<tr>
<td>pcolor</td>
<td>Create a pseudocolor plot of a 2-D array.</td>
</tr>
<tr>
<td>pcolormesh</td>
<td>Plot a quadrilateral mesh.</td>
</tr>
<tr>
<td>pie</td>
<td>Plot a pie chart.</td>
</tr>
<tr>
<td>plot</td>
<td>Plot lines and/or markers to the Axes.</td>
</tr>
<tr>
<td>plot_date</td>
<td>Plot with data with dates.</td>
</tr>
<tr>
<td>plotfile</td>
<td>Plot the data in in a file.</td>
</tr>
<tr>
<td>polar</td>
<td>Make a polar plot.</td>
</tr>
<tr>
<td>psd</td>
<td>Plot the power spectral density.</td>
</tr>
</tbody>
</table>

Continued on next page
Table 47.1 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>quiver</td>
<td>Plot a 2-D field of arrows.</td>
</tr>
<tr>
<td>quiverkey</td>
<td>Add a key to a quiver plot.</td>
</tr>
<tr>
<td>rc</td>
<td>Set the current rc params.</td>
</tr>
<tr>
<td>rc_context</td>
<td>Return a context manager for managing rc settings.</td>
</tr>
<tr>
<td>rcdefaults</td>
<td>Restore the default rc params.</td>
</tr>
<tr>
<td>rgrids</td>
<td>Get or set the radial gridlines on a polar plot.</td>
</tr>
<tr>
<td>savefig</td>
<td>Save the current figure.</td>
</tr>
<tr>
<td>sca</td>
<td>Set the current Axes instance to ax.</td>
</tr>
<tr>
<td>scatter</td>
<td>Make a scatter plot of x vs y, where x and y are sequence like objects of the same lengths.</td>
</tr>
<tr>
<td>sci</td>
<td>Set the current image.</td>
</tr>
<tr>
<td>semilogx</td>
<td>Make a plot with log scaling on the x axis.</td>
</tr>
<tr>
<td>semilogy</td>
<td>Make a plot with log scaling on the y axis.</td>
</tr>
<tr>
<td>set_cmap</td>
<td>Set the default colormap.</td>
</tr>
<tr>
<td>setp</td>
<td>Set a property on an artist object.</td>
</tr>
<tr>
<td>show</td>
<td>Display a figure.</td>
</tr>
<tr>
<td>specgram</td>
<td>Plot a spectrogram.</td>
</tr>
<tr>
<td>spy</td>
<td>Plot the sparsity pattern on a 2-D array.</td>
</tr>
<tr>
<td>stackplot</td>
<td>Draws a stacked area plot.</td>
</tr>
<tr>
<td>stem</td>
<td>Create a stem plot.</td>
</tr>
<tr>
<td>step</td>
<td>Make a step plot.</td>
</tr>
<tr>
<td>streamplot</td>
<td>Draws streamlines of a vector flow.</td>
</tr>
<tr>
<td>subplot</td>
<td>Return a subplot axes positioned by the given grid definition.</td>
</tr>
<tr>
<td>subplot2grid</td>
<td>Create a subplot in a grid.</td>
</tr>
<tr>
<td>subplot_tool</td>
<td>Launch a subplot tool window for a figure.</td>
</tr>
<tr>
<td>subplots</td>
<td>Create a figure with a set of subplots already made.</td>
</tr>
<tr>
<td>subplots_adjust</td>
<td>Tune the subplot layout.</td>
</tr>
<tr>
<td>suptitle</td>
<td>Add a centered title to the figure.</td>
</tr>
<tr>
<td>switch_backend</td>
<td>Switch the default backend.</td>
</tr>
<tr>
<td>table</td>
<td>Add a table to the current axes.</td>
</tr>
<tr>
<td>text</td>
<td>Add text to the axes.</td>
</tr>
<tr>
<td>theagrids</td>
<td>Get or set the theta locations of the gridlines in a polar plot.</td>
</tr>
<tr>
<td>tick_params</td>
<td>Change the appearance of ticks and tick labels.</td>
</tr>
<tr>
<td>ticklabel_format</td>
<td>Change the ScalarFormatter used by default for linear axes.</td>
</tr>
<tr>
<td>tight_layout</td>
<td>Automatically adjust subplot parameters to give specified padding.</td>
</tr>
<tr>
<td>title</td>
<td>Set a title of the current axes.</td>
</tr>
<tr>
<td>tricontour</td>
<td>Draw contours on an unstructured triangular grid.</td>
</tr>
<tr>
<td>tricontourf</td>
<td>Draw contours on an unstructured triangular grid.</td>
</tr>
<tr>
<td>triplot</td>
<td>Create a pseudocolor plot of an unstructured triangular grid.</td>
</tr>
<tr>
<td>twinx</td>
<td>Make a second axes that shares the x-axis.</td>
</tr>
<tr>
<td>twiny</td>
<td>Make a second axes that shares the y-axis.</td>
</tr>
<tr>
<td>vlines</td>
<td>Plot vertical lines.</td>
</tr>
<tr>
<td>xcorr</td>
<td>Plot the cross correlation between x and y.</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xkcd</td>
<td>Turns on xkcd sketch-style drawing mode.</td>
</tr>
<tr>
<td>xlabel</td>
<td>Set the x axis label of the current axis.</td>
</tr>
<tr>
<td>xlim</td>
<td>Get or set the x limits of the current axes.</td>
</tr>
<tr>
<td>xscale</td>
<td>Set the scaling of the x-axis.</td>
</tr>
<tr>
<td>xticks</td>
<td>Get or set the x-limits of the current tick locations and labels.</td>
</tr>
<tr>
<td>ylabel</td>
<td>Set the y axis label of the current axis.</td>
</tr>
<tr>
<td>ylim</td>
<td>Get or set the y-limits of the current axes.</td>
</tr>
<tr>
<td>yscale</td>
<td>Set the scaling of the y-axis.</td>
</tr>
<tr>
<td>yticks</td>
<td>Get or set the y-limits of the current tick locations and labels.</td>
</tr>
</tbody>
</table>

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:

```python
imshow(X, cmap=cm.hot)
```

or using the set_cmap() function:

```python
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** meant 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

The base colormaps are derived from those of the same name provided with Matlab:
<table>
<thead>
<tr>
<th>Colormap</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>autumn</td>
<td>sequential linearly-increasing shades of red-orange-yellow</td>
</tr>
<tr>
<td>bone</td>
<td>sequential increasing black-white color map with a tinge of blue, to emulate X-ray film</td>
</tr>
<tr>
<td>cool</td>
<td>linearly-decreasing shades of cyan-magenta</td>
</tr>
<tr>
<td>copper</td>
<td>sequential increasing shades of black-copper</td>
</tr>
<tr>
<td>flag</td>
<td>repetitive red-white-blue-black pattern (not cyclic at endpoints)</td>
</tr>
<tr>
<td>gray</td>
<td>sequential linearly-increasing black-to-white grayscale</td>
</tr>
<tr>
<td>hot</td>
<td>sequential black-red-yellow-white, to emulate blackbody radiation from an object at increasing temperatures</td>
</tr>
<tr>
<td>hsv</td>
<td>cyclic red-yellow-green-cyan-blue-magenta-red, formed by changing the hue component in the HSV color space</td>
</tr>
<tr>
<td>jet</td>
<td>a spectral map with dark endpoints, blue-cyan-yellow-red; based on a fluid-jet simulation by NCSA ¹</td>
</tr>
<tr>
<td>pink</td>
<td>sequential increasing pastel black-pink-white, meant for sepia tone colorization of photographs</td>
</tr>
<tr>
<td>prism</td>
<td>repetitive red-yellow-green-blue-purple-...-green pattern (not cyclic at endpoints)</td>
</tr>
<tr>
<td>spring</td>
<td>linearly-increasing shades of magenta-yellow</td>
</tr>
<tr>
<td>summer</td>
<td>sequential linearly-increasing shades of green-yellow</td>
</tr>
<tr>
<td>winter</td>
<td>linearly-increasing shades of blue-green</td>
</tr>
</tbody>
</table>

For the above list only, you can also set the colormap using the corresponding `pylab` shortcut interface function, similar to Matlab:

```python
imshow(X)
hot()
jet()
```

The next set of palettes are from the Yorick scientific visualisation package, an evolution of the GIST package, both by David H. Munro:

---

¹Rainbow colormaps, jet in particular, are considered a poor choice for scientific visualization by many researchers: [Rainbow Color Map (Still) Considered Harmful](https://www.c铁路/1994/05/13/)
### Colormap Description

<table>
<thead>
<tr>
<th>Colormap</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gist_earth</td>
<td>Mapmaker’s colors from dark blue deep ocean to green lowlands to brown highlands to white mountains</td>
</tr>
<tr>
<td>gist_heat</td>
<td>Sequential increasing black-red-orange-white, to emulate blackbody radiation from an iron bar as it grows hotter</td>
</tr>
<tr>
<td>gist_ncar</td>
<td>Pseudo-spectral black-blue-green-yellow-red-purple-white colormap from National Center for Atmospheric Research</td>
</tr>
<tr>
<td>gist_rainbow</td>
<td>Runs through the colors in spectral order from red to violet at full saturation (like hsv but not cyclic)</td>
</tr>
<tr>
<td>gist_stern</td>
<td>“Stern special” color table from Interactive Data Language software</td>
</tr>
</tbody>
</table>

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):

<table>
<thead>
<tr>
<th>Colormap</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BrBG</td>
<td>Brown, white, and blue-green</td>
</tr>
<tr>
<td>PiYG</td>
<td>Pink, white, and yellow-green</td>
</tr>
<tr>
<td>PRGn</td>
<td>Purple, white, and green</td>
</tr>
<tr>
<td>PuOr</td>
<td>Orange, white, and purple</td>
</tr>
<tr>
<td>RdBu</td>
<td>Red, white, and blue</td>
</tr>
<tr>
<td>RdGy</td>
<td>Red, white, and gray</td>
</tr>
<tr>
<td>RdYlBu</td>
<td>Red, yellow, and blue</td>
</tr>
<tr>
<td>RdYlGn</td>
<td>Red, yellow, and green</td>
</tr>
<tr>
<td>Spectral</td>
<td>Red, orange, yellow, green, and blue</td>
</tr>
</tbody>
</table>

ColorBrewer Sequential (luminance decreases monotonically):

---

2 Resembles “BkBlAqGrYeOrReViWh200” from NCAR Command Language. See Color Table Gallery
<table>
<thead>
<tr>
<th>Colormap</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blues</td>
<td>white to dark blue</td>
</tr>
<tr>
<td>BuGn</td>
<td>white, light blue, dark green</td>
</tr>
<tr>
<td>BuPu</td>
<td>white, light blue, dark purple</td>
</tr>
<tr>
<td>GnBu</td>
<td>white, light green, dark blue</td>
</tr>
<tr>
<td>Greens</td>
<td>white to dark green</td>
</tr>
<tr>
<td>Greys</td>
<td>white to black (not linear)</td>
</tr>
<tr>
<td>Oranges</td>
<td>white, orange, dark brown</td>
</tr>
<tr>
<td>OrRd</td>
<td>white, orange, dark red</td>
</tr>
<tr>
<td>PuBu</td>
<td>white, light purple, dark blue</td>
</tr>
<tr>
<td>PuBuGn</td>
<td>white, light purple, dark green</td>
</tr>
<tr>
<td>PuRd</td>
<td>white, light purple, dark red</td>
</tr>
<tr>
<td>Purples</td>
<td>white to dark purple</td>
</tr>
<tr>
<td>RdPu</td>
<td>white, pink, dark purple</td>
</tr>
<tr>
<td>Reds</td>
<td>white to dark red</td>
</tr>
<tr>
<td>YlGn</td>
<td>light yellow, dark green</td>
</tr>
<tr>
<td>YlGnBu</td>
<td>light yellow, light green, dark blue</td>
</tr>
<tr>
<td>YlOrBr</td>
<td>light yellow, orange, dark brown</td>
</tr>
<tr>
<td>YlOrRd</td>
<td>light yellow, orange, dark red</td>
</tr>
</tbody>
</table>

ColorBrewer Qualitative:

(For plotting nominal data, ListedColormap should be used, not LinearSegmentedColormap. Different sets of colors are recommended for different numbers of categories. These continuous versions of the qualitative schemes may be removed or converted in the future.)

- Accent
- Dark2
- Paired
- Pastel1
- Pastel2
- Set1
- Set2
- Set3

Other miscellaneous schemes:
<table>
<thead>
<tr>
<th>Colormap</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>afmhot</td>
<td>sequential black-orange-yellow-white blackbody spectrum, commonly used in atomic force microscopy</td>
</tr>
<tr>
<td>brg</td>
<td>blue-red-green</td>
</tr>
<tr>
<td>bwr</td>
<td>diverging blue-white-red</td>
</tr>
<tr>
<td>cool-warm</td>
<td>diverging blue-gray-red, meant to avoid issues with 3D shading, color blindness, and ordering of colors</td>
</tr>
<tr>
<td>CM-Rmap</td>
<td>“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.”</td>
</tr>
<tr>
<td>cube-helix</td>
<td>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.</td>
</tr>
<tr>
<td>gnu-plot</td>
<td>gnuplot’s traditional pm3d scheme (black-blue-red-yellow)</td>
</tr>
<tr>
<td>gnu-plot2</td>
<td>sequential color printable as gray (black-blue-violet-yellow-white)</td>
</tr>
<tr>
<td>ocean</td>
<td>green-blue-white</td>
</tr>
<tr>
<td>rainbow</td>
<td>spectral purple-blue-green-yellow-orange-red colormap with diverging luminance</td>
</tr>
<tr>
<td>seismic</td>
<td>diverging blue-white-red</td>
</tr>
<tr>
<td>nipy_spectral</td>
<td>black-purple-blue-green-yellow-red-white spectrum, originally from the Neuroimaging in Python project</td>
</tr>
<tr>
<td>terrain</td>
<td>mapmaker’s colors, blue-green-yellow-brown-white, originally from IGOR Pro</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Colormap</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gist_gray</td>
<td>identical to gray</td>
</tr>
<tr>
<td>gist_yarg</td>
<td>identical to gray_r</td>
</tr>
<tr>
<td>binary</td>
<td>identical to gray_r</td>
</tr>
<tr>
<td>spectral</td>
<td>identical to nipy_spectral</td>
</tr>
</tbody>
</table>

---

3See Diverging Color Maps for Scientific Visualization by Kenneth Moreland.
4See A Color Map for Effective Black-and-White Rendering of Color-Scale Images by Carey Rappaport
5Changed to distinguish from ColorBrewer’s Spectral map. spectral() still works, but set_cmap('nipy_spectral') is recommended for clarity.
This chapter is a log of changes to matplotlib that affect the outward-facing API. If updating matplotlib breaks your scripts, this list may help describe what changes may be necessary in your code or help figure out possible sources of the changes you are experiencing.

For new features that were added to matplotlib, please see What’s new in matplotlib.

48.1 Changes in 1.3.x

48.1.1 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.
  - The following kwargs to the legend function have been renamed:
    * pad -> borderpad
    * labelsep -> labelspaceing
    * 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 kwarg 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.liapunov 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.

### 48.1.2 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` -> `collection.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 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 mpl module is now deprecated. Those who relied on this module should transition to simply using `import matplotlib as mpl`. 
48.1.3 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:

  ```python
  import copy
  path = copy.deepcopy(Path.unit_circle())
  # 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 `axvline` and `axhline`, since their limits will not 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()`.

### 48.1.4 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 `plotfile()`. To draw the axes grid, set the `axes.grid` rcParam to `True`, or explicitly call `grid()`.

### 48.2 Changes in 1.2.x

• The classic option of the rc parameter `toolbar` is deprecated and will be removed in the next release.

• The `isvector()` method has been removed since it is no longer functional.

• The `rasterization_zorder` property on `Axes` 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 `pie()` and `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 `projection_factory()` is now deprecated in favour of axes class identification using `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:

```python
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)
```

This change means that third party objects can expose themselves as matplotlib axes by providing a `_as_mpl_axes` method. See *Adding new scales and projections to matplotlib* 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`:

```python
class MyTransform(mtrans.Transform):
    def transform(self, xy):
        ...
    transform_non_affine = transform
```

This approach will no longer function correctly and should be changed to:

```python
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:
Matplotlib, Release 1.3.0

```python
>>> 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.],
          [ 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:

```python
>>> 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.

- `ColorConverter`, `Colormap` and `Normalize` now subclasses `object`

- ContourSet instances no longer have a `transform` attribute. Instead, access the transform with the `get_transform` method.

## 48.3 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”.
48.4 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:

```python
draw_path_collection(self, master_transform, cliprect, clippath,
    clippath_trans, paths, all_transforms, offsets,
    offsetTrans, facecolors, edgecolors, linewidths,
    linestyles, antialiaseds, urls)
```

# is now

```python
draw_path_collection(self, gc, master_transform, paths, all_transforms,
    offsets, offsetTrans, facecolors, edgecolors,
    linewidths, linestyles, antialiaseds, urls)
```

```python
draw_quad_mesh(self, master_transform, cliprect, clippath,
    clippath_trans, meshWidth, meshHeight, coordinates,
    offsets, offsetTrans, facecolors, antialiased,
    showedges)
```

# is now

```python
draw_quad_mesh(self, gc, master_transform, meshWidth, meshHeight,
    coordinates, offsets, offsetTrans, facecolors,
    antialiased, showedges)
```

```python
draw_image(self, x, y, im, bbox, clippath=None, clippath_trans=None)
```

# is now

```python
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.

## 48.5 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.

48.6 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.label.Label are now deprecated and new set of parameters are introduced. The new parameters are given as a fraction of the font-size. Also, scattery-offsets, fancybox and columnspacing are added as keyword parameters.

<table>
<thead>
<tr>
<th>Deprecated</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td>pad</td>
<td>borderpad</td>
</tr>
<tr>
<td>labelsep</td>
<td>labelsspacing</td>
</tr>
<tr>
<td>handlelen</td>
<td>handlelength</td>
</tr>
<tr>
<td>handletextsep</td>
<td>handletextpad</td>
</tr>
<tr>
<td>axespad</td>
<td>borderaxespad</td>
</tr>
</tbody>
</table>

• 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 con-
cern 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.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.

48.7 Changes for 0.98.1

• Removed broken `matplotlib.axes3d` support and replaced it with a non-implemented error pointing
to 0.91.x

48.8 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.

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`)

### 48.8.1 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.Locator.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.

### 48.8. Changes for 0.98.0
### matplotlib.transforms

<table>
<thead>
<tr>
<th>Old method</th>
<th>New method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bbox.get_bounds()</td>
<td>transforms.Bbox.bounds</td>
</tr>
<tr>
<td>Bbox.width()</td>
<td>transforms.Bbox.width</td>
</tr>
<tr>
<td>Bbox.height()</td>
<td>transforms.Bbox.height</td>
</tr>
<tr>
<td>Bbox.intervalx().get_bounds()</td>
<td>transforms.Bbox.intervalx.get_bounds()</td>
</tr>
<tr>
<td>Bbox.intervalx().set_bounds()</td>
<td>transforms.Bbox.intervalx.set_bounds()</td>
</tr>
<tr>
<td>Bbox.intervaly().get_bounds()</td>
<td>transforms.Bbox.intervaly.get_bounds()</td>
</tr>
<tr>
<td>Bbox.intervaly().set_bounds()</td>
<td>transforms.Bbox.intervaly.set_bounds()</td>
</tr>
<tr>
<td>Bbox.xmin()</td>
<td>transforms.Bbox.x0 or transforms.Bbox.xmin</td>
</tr>
<tr>
<td>Bbox.ymin()</td>
<td>transforms.Bbox.y0 or transforms.Bbox.ymin</td>
</tr>
<tr>
<td>Bbox.xmax()</td>
<td>transforms.Bbox.x1 or transforms.Bbox.xmax</td>
</tr>
<tr>
<td>Bbox.ymax()</td>
<td>transforms.Bbox.y1 or transforms.Bbox.ymax</td>
</tr>
<tr>
<td>Bbox.overlaps(bboxes)</td>
<td>Bbox.count_overlaps(bboxes)</td>
</tr>
<tr>
<td>bbox_all(bboxes)</td>
<td>Bbox.union(bboxes) [transforms.Bbox.union() is a staticmethod.]</td>
</tr>
<tr>
<td>lbwh_to_bbox(l, b, w, h)</td>
<td>Bbox.from_bounds(x0, y0, w, h)</td>
</tr>
<tr>
<td>inverse_transform_bbox(bbox)</td>
<td>Bbox.inverse_transformed(trans)</td>
</tr>
<tr>
<td>Interval.contains_open(tuple, v)</td>
<td>interval_contains_open(tuple, v)</td>
</tr>
<tr>
<td>Interval.contains(tuple, v)</td>
<td>interval_contains(tuple, v)</td>
</tr>
<tr>
<td>identity_transform()</td>
<td>matplotlib.transforms.IdentityTransform</td>
</tr>
<tr>
<td>blend_xy_sep_transform(xtrans, ytrans)</td>
<td>blended_transform_factory(xtrans, ytrans)</td>
</tr>
<tr>
<td>scale_transform(xs, ys)</td>
<td>Affine2D().scale(xs, ys)</td>
</tr>
<tr>
<td>get_bbox_transform(boxin, boxout)</td>
<td>BboxTransform(boxin, boxout) or BBoxTransformFrom(boxin, boxout)</td>
</tr>
<tr>
<td>Transform.seq_xy_tup(points)</td>
<td>Transform.transform(points)</td>
</tr>
<tr>
<td>Transform.inverse_xy_tup(points)</td>
<td>Transform.inverted().transform(points)</td>
</tr>
</tbody>
</table>

### matplotlib.axes

<table>
<thead>
<tr>
<th>Old method</th>
<th>New method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Axes.get_position()</td>
<td>matplotlib.axes.Axes.get_position()</td>
</tr>
<tr>
<td>Axes.set_position()</td>
<td>matplotlib.axes.Axes.set_position()</td>
</tr>
<tr>
<td>Axes.toggle_log_linearity()</td>
<td>matplotlib.axes.Axes.set_yscale()</td>
</tr>
<tr>
<td>Subplot class</td>
<td>removed.</td>
</tr>
</tbody>
</table>

---

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.
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 refactoring allows for more than two scale types ('log' or 'linear'), it no longer makes sense to have a toggle. Axes.toggle_log_linearity() has been removed.
The Polar class has moved to `matplotlib.projections.polar`.

### `matplotlib.artist`

<table>
<thead>
<tr>
<th>Old method</th>
<th>New method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Artist.set_clip_path(path)</td>
<td>Artist.set_clip_path(path, transform)⁵</td>
</tr>
</tbody>
</table>

### `matplotlib.collections`

<table>
<thead>
<tr>
<th>Old method</th>
<th>New method</th>
</tr>
</thead>
<tbody>
<tr>
<td>linestyle</td>
<td>linestyles</td>
</tr>
</tbody>
</table>

### `matplotlib.colors`

<table>
<thead>
<tr>
<th>Old method</th>
<th>New method</th>
</tr>
</thead>
<tbody>
<tr>
<td>ColorConvertor.to_rgba_list(c)</td>
<td>ColorConvertor.to_rgba_array(c)</td>
</tr>
<tr>
<td></td>
<td><code>[matplotlib.colors.ColorConvertor.to_rgba_array(c) returns an Nx4 Numpy array of RGBA color quadruples.]</code></td>
</tr>
</tbody>
</table>

### `matplotlib.contour`

<table>
<thead>
<tr>
<th>Old method</th>
<th>New method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contour._segments</td>
<td>matplotlib.contour.Contour.get_paths<code>()</code> [Returns a list of matplotlib.path.Path instances.]</td>
</tr>
</tbody>
</table>

### `matplotlib.figure`

<table>
<thead>
<tr>
<th>Old method</th>
<th>New method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure.dpi.get() / Figure.dpi.set()</td>
<td><code>matplotlib.figure.Figure.dpi (a property)</code></td>
</tr>
</tbody>
</table>

### `matplotlib.patches`

<table>
<thead>
<tr>
<th>Old method</th>
<th>New method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Patch.get_verts()</td>
<td>matplotlib.patches.Patch.get_path() [Returns a matplotlib.path.Path instance]</td>
</tr>
</tbody>
</table>

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.

⁶Linestyles are now treated like all other collection attributes, i.e. a single value or multiple values may be provided.
**matplotlib.backend_bases**

<table>
<thead>
<tr>
<th>Old method</th>
<th>New method</th>
</tr>
</thead>
<tbody>
<tr>
<td>GraphicsContext.set_clip_rectangle(tuple)</td>
<td>GraphicsContext.set_clip_rectangle(bbox)</td>
</tr>
<tr>
<td>GraphicsContext.get_clip_path()</td>
<td>GraphicsContext.get_clip_path()</td>
</tr>
<tr>
<td>GraphicsContext.set_clip_path()</td>
<td>GraphicsContext.set_clip_path()</td>
</tr>
</tbody>
</table>

**RendererBase**

New methods:
- `draw_path(self, gc, path, transform, rgbFace)`
- `draw_markers(self, gc, marker_path, marker_trans, path, trans, rgbFace)`
- `<matplotlib.backend_bases.RendererBase.draw_markers()`
- `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`

---

<sup>7</sup> `matplotlib.backend_bases.GraphicsContext.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.

<sup>8</sup> `matplotlib.backend_bases.GraphicsContext.set_clip_path()` now only accepts a `matplotlib.transforms.TransformedPath` instance.
48.9 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.__init__`.

48.10 Changes for 0.91.1

48.11 Changes for 0.91.0

- Changed `cbook.is_file_like()` to `cbook.is_writable_file_like()` and corrected behavior.
- Added `ax` kwarg 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., subseting.
- `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` kwarg.
- 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
Matplotlib, Release 1.3.0

changed to \$\mathcal{R}\$. Alternatively, you may use the new \LaTeX-style font commands (\texttt{\textbackslash mathcal}, \texttt{\textbackslash mathrm}, \texttt{\textbackslash mathit}, \texttt{\textbackslash mathtt}) which do affect the following group, e.g., \$\mathcal{R}\$.

- Text creation commands have a new default linespacing and a new \texttt{linespacing} kwarg, which is a multiple of the maximum vertical extent of a line of ordinary text. The default is 1.2; \texttt{linespacing=2} would be like ordinary double spacing, for example.

- Changed default kwarg in \texttt{matplotlib.colors.Normalize.__init__()} to \texttt{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 \texttt{set_xlim()} and \texttt{set_ylim()} \texttt{True} by default; removed the \texttt{Axes} custom callback handling into a ‘callbacks’ attribute which is a \texttt{CallbackRegistry} instance. This now supports the ‘xlim_changed’ and ‘ylim_changed’ \texttt{Axes} events.

48.12 Changes for 0.90.1

The file \texttt{dviread.py} has a (very limited and fragile) dvi reader for \texttt{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 ‘\texttt{0.5}’. Added alpha kwarg to \texttt{ColorConverter.to_rgba_list}.

New method \texttt{set_bounds(vmin, vmax)} for formatters, locators sets the \texttt{viewInterval} and \texttt{dataInterval} from floats.

Removed deprecated \texttt{colorbar_classic}.

\texttt{Line2D.get_xdata} and \texttt{get_ydata valid_only=False} kwarg is replaced by \texttt{orig=True}. When True, it returns the original data, otherwise the processed data (masked, converted)

Some modifications to the \texttt{units} interface.
\texttt{units.ConversionInterface.tickers} renamed to \texttt{units.ConversionInterface.axisinfo} and it now returns a \texttt{units.AxisInfo} object rather than a tuple. This will make it easier to add axis info functionality (eg 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, \texttt{units.ConversionInterface.convert_to_value} is now simply named \texttt{units.ConversionInterface.convert}.

\texttt{Axes.errorbar} uses \texttt{Axes.vlines} and \texttt{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 \texttt{LineCollection} rather than a list of lines. The new return signature for \texttt{errorbar} is \texttt{ylins, caplines, errorcollections} where \texttt{errorcollections} is a \texttt{xerrcollection}, \texttt{yerrcollection}
Axes.vlines and Axes.hlines now create and return 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.

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

### 48.13 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.

Removed lineprops and markerprops from the Annotation code and replaced them with an arrow configurable with kwarg arrowprops. See examples/annotation_demo.py - JDH

48.14 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 anticlockwise. 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

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.

48.15 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.

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.

48.16 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.
Matplotlib, Release 1.3.0

Changed Figure.colorbar():
    kw argument order changed;
    if mappable arg is a non-filled ContourSet, colorbar() shows
    lines instead of 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.

48.17 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.

48.18 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.
48.19 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

```python
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

```python
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)
```
48.20 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].
48.21 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

48.22 Changes for 0.73

- Removed deprecated ColormapJet and friends
- Removed all error handling from the verbose object
- `figure num of zero` is now allowed

48.23 Changes for 0.72

- `Line2D`, `Text`, and `Patch copy_properties` renamed `update_from` and moved into artist base class
- `LineCollectitons.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
48.24 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


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, eg (amin, amax, asum)

48.25 Changes for 0.70

MplEvent factored into a base class Event and derived classes MouseEvent and KeyEvent

Removed definct set_measurement in wx toolbar
48.26 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, eg, figmanager.add_axes, you can now simply do figmanager.canvas.figure.add_axes.

48.27 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 (oblique, etc) or the rotation of the text. Use style and rotation instead.

48.28 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

    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

48.29 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)

48.30 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

48.31 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.
48.32 Changes for 0.54

48.32.1 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, eg you can say:

```python
c = scatter(blah, blah)
c.set_linewidth(1.0)
c.set_facecolor('r')
c.set_alpha(0.5)
```

or:

```python
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, eg 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:
48.32.2 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:

```python
f = Figure(figsize=(5,4), dpi=100)
a = Subplot(f, 111)
f.add_axis(a)
```

use:

```python
f = Figure(figsize=(5,4), dpi=100)
a = f.add_subplot(111)
```

That is, add_axis no longer exists and is replaced by:

```python
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

```python
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

```python
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 of 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

```python
l = Line2D(dpi, bbox, x, y, color, transx = transx, transy = transy, )
```

now all you need is

```python
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 leave 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, ie 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:

```python
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.

48.33 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

```python
    fig.draw()
```

In the new API, you do

```python
    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 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 intialized 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
gure.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.

48.34 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, eg 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

* 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, eg figManager.window for GTK and figManager.frame for wx

### 48.35 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
Matplotlib, Release 1.3.0

- Axes
  * removed figbg attribute
  * 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.
matplotlib.use(arg, warn=True, force=False)

Set the matplotlib backend to one of the known backends.

The argument is case-insensitive. warn specifies whether a warning should be issued if a backend has already been set up. force is an experimental flag that tells matplotlib to attempt to initialize a new backend by reloading the backend module.

**Note:** This function must be called before importing pyplot for the first time; or, if you are not using pyplot, it must be called before importing matplotlib.backends. If warn is True, a warning is issued if you try and call this after pylab or pyplot have been loaded. In certain black magic use cases, e.g. pyplot.switch_backend(), we are doing the reloading necessary to make the backend switch work (in some cases, e.g., pure image backends) so one can set warn=False to suppress the warnings.

To find out which backend is currently set, see matplotlib.get_backend().

matplotlib.get_backend()

Return the name of the current backend.

matplotlib.rcParams

An instance of RcParams for handling default matplotlib values.

matplotlib.rc(group, **kwargs)

Set the current rc params. 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, eg:

rc('lines', linewidth=2, color='r')

sets the current rc params and is equivalent to:

rcParams['lines.linewidth'] = 2
rcParams['lines.color'] = 'r'

The following aliases are available to save typing for interactive users:
<table>
<thead>
<tr>
<th>Alias</th>
<th>Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>'lw'</td>
<td>'linewidth'</td>
</tr>
<tr>
<td>'ls'</td>
<td>'linestyle'</td>
</tr>
<tr>
<td>'c'</td>
<td>'color'</td>
</tr>
<tr>
<td>'fc'</td>
<td>'facecolor'</td>
</tr>
<tr>
<td>'ec'</td>
<td>'edgecolor'</td>
</tr>
<tr>
<td>'mew'</td>
<td>'markeredgewidth'</td>
</tr>
<tr>
<td>'aa'</td>
<td>'antialiased'</td>
</tr>
</tbody>
</table>

Thus you could abbreviate the above rc command as:

```python
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:

```python
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 `rcdefaults()` to restore the default rc params after changes.

```python
matplotlib.matplotlib_fname()
```

Get the location of the config file.

The file location is determined in the following order:

- `$PWD/matplotlibrc`
- environment variable `MATPLOTLIBRC`
- `$MPLCONFIGDIR/matplotlib`
- On Linux, `- $HOME/.matplotlib/matplotlibrc, if it exists`
- or `$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` for a system-defined copy.

```python
class matplotlib.RcParams
```

A dictionary object including validation

validating functions are defined and associated with rc parameters in `matplotlib.rcsetup`
`matplotlib.rc_params(fail_on_error=False)`
Return a `matplotlib.RcParams` instance from the default matplotlib rc file.

`matplotlib.rc_params_from_file(fname, fail_on_error=False)`
Return a `matplotlib.RcParams` instance from the contents of the given filename.

class `matplotlib.rc_context(rc=None, fname=None)`
Return a context manager for managing rc settings.

This allows one to do:

```python
with mpl.rc_context(fname='screen.rc'):
    plt.plot(x, a)
    with mpl.rc_context(fname='print.rc'):
        plt.plot(x, b)
    plt.plot(x, c)
```

The ‘a’ vs ‘x’ and ‘c’ vs ‘x’ plots would have settings from ‘screen.rc’, while the ‘b’ vs ‘x’ plot would have settings from ‘print.rc’.

A dictionary can also be passed to the context manager:

```python
with mpl.rc_context(rc={'text.usetex': True}, fname='screen.rc'):
    plt.plot(x, a)
```

The ‘rc’ dictionary takes precedence over the settings loaded from ‘fname’. Passing a dictionary only is also valid.
AFM (ADOBE FONT METRICS INTERFACE)

50.1 matplotlib.afm

This is 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 requires only built-in python libs:

```python
>>> from matplotlib import rcParams
>>> import os.path
...  
>>> afm_fname = os.path.join(rcParams['datapath'],
...    'fonts', 'afm', 'ptmr8a.afm')
>>> from matplotlib.afm import AFM
>>> afm = AFM(open(afm_fname))
>>> 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]
```

class matplotlib.afm.AFM(fh)

Bases: object

Parse the AFM file in file object fh

get_angle()

Return the fontangle as float
get_bbox_char(c, isord=False)

get_capheight()
    Return the cap height as float

get_familyname()
    Return the font family name, e.g., ‘Times’

get_fontname()
    Return the font name, e.g., ‘Times-Roman’

get_fullname()
    Return the font full name, e.g., ‘Times-Roman’

get_height_char(c, isord=False)
    Get the height of character c from the bounding box. This is the ink height (space is 0)

get_horizontal_stem_width()
    Return the standard horizontal stem width as float, or None if not specified in AFM file.

get_kern_dist(c1, c2)
    Return the kerning pair distance (possibly 0) for chars c1 and c2

get_kern_dist_from_name(name1, name2)
    Return the kerning pair distance (possibly 0) for chars name1 and name2

get_name_char(c, isord=False)
    Get the name of the character, ie, ‘;’ is ‘semicolon’

get_str_bbox(s)
    Return the string bounding box

get_str_bbox_and_descent(s)
    Return the string bounding box

get_underline_thickness()
    Return the underline thickness as float

get_vertical_stem_width()
    Return the standard vertical stem width as float, or None if not specified in AFM file.

get_weight()
    Return the font weight, e.g., ‘Bold’ or ‘Roman’

get_width_char(c, isord=False)
    Get the width of the character from the character metric WX field

get_width_from_char_name(name)
    Get the width of the character from a type1 character name

get_xheight()
    Return the xheight as float

string_width_height(s)
    Return the string width (including kerning) and string height as a (w, h) tuple.
matplotlib.afm.parse_afm(fh)

Parse the Adobe Font Metics file in file handle fh. Return value is a (dhead, dcmetrics, dkernpairs, dcomposite) tuple where dhead is a _parse_header() dict, dcmetrics is a _parse_composites() dict, dkernpairs is a _parse_kern_pairs() dict (possibly {}), and dcomposite is a _parse_composites() dict (possibly {}})
51.1 matplotlib.animation

```python
class matplotlib.animation.AVConvBase
    Bases: matplotlib.animation.FFMpegBase
    args_key = 'animation.avconv_args'

    exec_key = 'animation.avconv_path'

class matplotlib.animation.AVConvFileWriter(*args, **kwargs)
    Bases: matplotlib.animation.AVConvBase, matplotlib.animation.FFMpegFileWriter

class matplotlib.animation.AVConvWriter(fps=5, codec=None, bitrate=None, extra_args=None, metadata=None)
    Bases: matplotlib.animation.AVConvBase, matplotlib.animation.FFMpegWriter

    Construct a new MovieWriter object.

    fps: int Framerate for movie.

    codec: string or None, optional The codec to use. If None (the default) the setting in the rcParam animation.codec is used.

    bitrate: int or None, optional The bitrate for the saved movie file, which is one way to control the output file size and quality. The default value is None, which uses the value stored in the rcParam animation.bitrate. A value of -1 implies that the bitrate should be determined automatically by the underlying utility.

    extra_args: list of strings or None A list of extra string arguments to be passed to the underlying movie utility. The default is None, which passes the additional arguments in the 'animation.extra_args' rcParam.

    metadata: dict of string:string or None 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.
```

```python
class matplotlib.animation.Animation(fig, event_source=None, blit=False)
    Bases: object
```
This class wraps the creation of an animation using matplotlib. It is only a base class which should be subclassed to provide needed behavior.

`fig` is the figure object that is used to get draw, resize, and any other needed events.

`event_source` is 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` is a boolean that controls whether blitting is used to optimize drawing.

```
new_frame_seq()
Creates a new sequence of frame information.
```

```
new_saved_frame_seq()
Creates a new sequence of saved/cached frame information.
```

```
save(filename, writer=None, fps=None, dpi=None, codec=None, bitrate=None, extra_args=None, metadata=None, extra_anim=None, savefig_kwargs=None)
Saves a movie file by drawing every frame.
```

`filename` is the output filename, e.g., `mymovie.mp4`

`writer` is either an instance of `MovieWriter` or a string key that identifies a class to use, such as ‘ffmpeg’ or ‘mencoder’. If nothing is passed, the value of the rcparam `animation.writer` is used.

`fps` is the frames per second in the movie. Defaults to None, which will use the animation’s specified interval to set the frames per second.

`dpi` controls the dots per inch for the movie frames. This combined with the figure’s size in inches controls the size of the movie.

`codec` is the video codec to be used. Not all codecs are supported by a given `MovieWriter`. If none is given, this defaults to the value specified by the rcparam `animation.codec`.

`bitrate` specifies the amount of bits used per second in the compressed movie, in kilobits per second. A higher number means a higher quality movie, but at the cost of increased file size. If no value is given, this defaults to the value given by the rcparam `animation.bitrate`.

`extra_args` is a list of extra string arguments to be passed to the underlying movie utility. The default is None, which passes the additional arguments in the ‘animation.extra_args’ rcParam.

`metadata` is 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.

`extra_anim` is a list of additional `Animation` objects that should be included in the saved movie file. These need to be from the same `matplotlib.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` is a dictionary containing keyword arguments to be passed on to the ‘savefig’ command which is called repeatedly to save the individual frames. This can be used to set tight bounding boxes, for example.

```
class matplotlib.animation.ArtistAnimation(fig, artists, *args, **kwargs)
Bases: matplotlib.animation.TimedAnimation
```
Before calling this function, all plotting should have taken place and the relevant artists saved.

frame_info is a list, with each list entry a collection of artists that represent what needs to be enabled on each frame. These will be disabled for other frames.

class matplotlib.animation.FFMpegBase

    args_key = ‘animation.ffmpeg_args’

    exec_key = ‘animation.ffmpeg_path’

    output_args

class matplotlib.animation.FFMpegFileWriter(*args, **kwargs)
    Bases: matplotlib.animation.MovieWriter, matplotlib.animation.FFMpegBase

    supported_formats = ['png', 'jpeg', 'ppm', 'tiff', 'sgi', 'bmp', 'pbm', 'raw', 'rgba']

class matplotlib.animation.FFMpegWriter(fps=5, codec=None, bitrate=None, extra_args=None, metadata=None)
    Bases: matplotlib.animation.MovieWriter, matplotlib.animation.FFMpegBase

    Construct a new MovieWriter object.

    fps: int Framerate for movie.

    codec: string or None, optional The codec to use. If None (the default) the setting in the rcParam animation.codec is used.

    bitrate: int or None, optional The bitrate for the saved movie file, which is one way to control the output file size and quality. The default value is None, which uses the value stored in the rcParam animation.bitrate. A value of -1 implies that the bitrate should be determined automatically by the underlying utility.

    extra_args: list of strings or None A list of extra string arguments to be passed to the underlying movie utility. The default is None, which passes the additional arguments in the ‘animation.extra_args’ rcParam.

    metadata: dict of string:string or None 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.

class matplotlib.animation.FileMovieWriter(*args, **kwargs)
    Bases: matplotlib.animation.MovieWriter

    MovieWriter subclass that handles writing to a file.

    cleanup()

    finish()
frame_format
Format (png, jpeg, etc.) to use for saving the frames, which can be decided by the individual subclasses.

setup(fig, outfile, dpi, frame_prefix='._tmp', clear_temp=True)
Perform setup for writing the movie file.
fig: matplotlib.Figure instance The figure object that contains the information for frames
outfile: string The filename of the resulting movie file
dpi: int The DPI (or resolution) for the file. This controls the size in pixels of the resulting movie file.
frame_prefix: string, optional The filename prefix to use for the temporary files. Defaults to '_tmp'
clear_temp: bool Specifies whether the temporary files should be deleted after the movie is written. (Useful for debugging.) Defaults to True.

class matplotlib.animation.FuncAnimation(fig, func, frames=None, init_func=None, fargs=None, save_count=None, **kwargs)
Bases: matplotlib.animation.TimedAnimation
Makes an animation by repeatedly calling a function func, passing in (optional) arguments in fargs.
frames can be a generator, an iterable, or a number of frames.
init_func is 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.
If blit=True, func and init_func should return an iterable of drawables to clear.

new_frame_seq()

new_saved_frame_seq()

class matplotlib.animation.ImageMagickBase

args_key = ‘animation.convert_args’
delay

exec_key = ‘animation.convert_path’

output_args

class matplotlib.animation.ImageMagickFileWriter(*args, **kwargs)
Bases: matplotlib.animation.FileMovieWriter, matplotlib.animation.ImageMagickBase

supported_formats = ['png', 'jpeg', 'ppm', 'tiff', 'sgi', 'bmp', 'pbm', 'raw', 'rgba']
class matplotlib.animation.ImageMagickWriter(
    fps=5, codec=None, bitrate=None, extra_args=None, metadata=None)
Bases: matplotlib.animation.MovieWriter, matplotlib.animation.ImageMagickBase

Construct a new MovieWriter object.

fps: int Framerate for movie.

codec: string or None, optional The codec to use. If None (the default) the setting in the rcParam animation.codec is used.

bitrate: int or None, optional The bitrate for the saved movie file, which is one way to control the output file size and quality. The default value is None, which uses the value stored in the rcParam animation.bitrate. A value of -1 implies that the bitrate should be determined automatically by the underlying utility.

extra_args: list of strings or None A list of extra string arguments to be passed to the underlying movie utility. The default is None, which passes the additional arguments in the ‘animation.extra_args’ rcParam.

metadata: dict of string:string or None 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.

class matplotlib.animation.MencoderBase

allowed_metadata = ['name', 'artist', 'genre', 'subject', 'copyright', 'srcform', 'comment']

args_key = 'animation.mencoder_args'

exec_key = 'animation.mencoder_path'

output_args

class matplotlib.animation.MencoderFileWriter(*args, **kwargs)
Bases: matplotlib.animation.FileMovieWriter, matplotlib.animation.MencoderBase

supported_formats = ['png', 'jpeg', 'tga', 'sgi']

class matplotlib.animation.MencoderWriter(
    fps=5, codec=None, bitrate=None, extra_args=None, metadata=None)
Bases: matplotlib.animation.MovieWriter, matplotlib.animation.MencoderBase

Construct a new MovieWriter object.

fps: int Framerate for movie.

codec: string or None, optional The codec to use. If None (the default) the setting in the rcParam animation.codec is used.
**bitrate: int or None, optional** The bitrate for the saved movie file, which is one way to control the output file size and quality. The default value is None, which uses the value stored in the rcParam `animation.bitrate`. A value of -1 implies that the bitrate should be determined automatically by the underlying utility.

**extra_args: list of strings or None** A list of extra string arguments to be passed to the underlying movie utility. The default is None, which passes the additional arguments in the ‘animation.extra_args’ rcParam.

**metadata: dict of string:string or None** 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.

```python
class matplotlib.animation.MovieWriter(fps=5, codec=None, bitrate=None, extra_args=None, metadata=None)
```

Bases: object

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:

```python
with moviewriter.saving('myfile.mp4'):
    # Iterate over frames
    moviewriter.grab_frame()
```

The use of the context manager ensures that setup and cleanup are performed as necessary.

**frame_format: string** The format used in writing frame data, defaults to ‘rgba’

Construct a new MovieWriter object.

**fps: int** Framerate for movie.

**codec: string or None, optional** The codec to use. If None (the default) the setting in the rcParam `animation.codec` is used.

**bitrate: int or None, optional** The bitrate for the saved movie file, which is one way to control the output file size and quality. The default value is None, which uses the value stored in the rcParam `animation.bitrate`. A value of -1 implies that the bitrate should be determined automatically by the underlying utility.

**extra_args: list of strings or None** A list of extra string arguments to be passed to the underlying movie utility. The default is None, which passes the additional arguments in the ‘animation.extra_args’ rcParam.

**metadata: dict of string:string or None** 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.

**classmethod bin_path()**

Returns 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()
    Clean-up and collect the process used to write the movie file.

finish()
    Finish any processing for writing the movie.

frame_size
    A tuple (width, height) in pixels of a movie frame.

grab_frame(**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' command that saves the figure.

classmethod isAvailable()
    Check to see if a MovieWriter subclass is actually available by running the commandline tool.

saving(*args, **kwds)
    Context manager to facilitate writing the movie file.

    *args are any parameters that should be passed to setup.

setup(fig, outfile, dpi, *args)
    Perform setup for writing the movie file.

    fig: matplotlib.Figure instance The figure object that contains the information for frames
    outfile: string The filename of the resulting movie file
    dpi: int The DPI (or resolution) for the file. This controls the size in pixels of the resulting movie
    file.

class matplotlib.animation.MovieWriterRegistry
    Bases: object

    is_available(name)

list()
    Get a list of available MovieWriters.

register(name)

class matplotlib.animation.TimedAnimation(fig, interval=200, repeat_delay=None, repeat=True, event_source=None, *args, **kwargs)
    Bases: matplotlib.animation.Animation

    Animation subclass that supports time-based animation, drawing a new frame every interval milliseconds.

    repeat controls whether the animation should repeat when the sequence of frames is completed.

    repeat_delay optionally adds a delay in milliseconds before repeating the animation.
52.1 matplotlib.artist

```python
class matplotlib.artist.Artist
    Bases: object

    Abstract base class for someone who renders into a FigureCanvas.
```
add_callback(func)
    Adds a callback function that will be called whenever one of the Artist's properties changes.
    Returns an id that is useful for removing the callback with remove_callback() later.

aname = 'Artist'

contains(mouseevent)
    Test whether the artist contains the mouse event.
    Returns the truth value and a dictionary of artist specific details of selection, such as which points are contained in the pick radius. See individual artists for details.

convert_xunits(x)
    For artists in an axes, if the xaxis has units support, convert x using xaxis unit type

convert_yunits(y)
    For artists in an axes, if the yaxis has units support, convert y using yaxis unit type

draw(renderer, *args, **kwargs)
    Derived classes drawing method

findobj(match=None, include_self=True)
    Find artist objects.
    Recursively find all Artist instances contained in self.
    match can be
    • None: return all objects contained in artist.
    • function with signature boolean = match(artist) used to filter matches
    • class instance: e.g., Line2D. Only return artists of class type.
    If include_self is True (default), include self in the list to be checked for a match.

get_agg_filter()
    return filter function to be used for agg filter

get_alpha()
    Return the alpha value used for blending - not supported on all backends

get_animated()
    Return the artist’s animated state

get_axes()
    Return the Axes instance the artist resides in, or None

get_children()
    Return a list of the child Artist's this :class:`Artist` contains.

get_clip_box()
    Return artist clipbox

get_clip_on()
    Return whether artist uses clipping

get_clip_path()
    Return artist clip path
get_contains()  
Return the _contains test used by the artist, or None for default.

get_figure()  
Return the Figure instance the artist belongs to.

get_gid()  
Returns the group id

get_label()  
Get the label used for this artist in the legend.

get_path_effects()  

get_picker()  
Return the picker object used by this artist

get_rasterized()  
return True if the artist is to be rasterized

get_sketch_params()  
Returns the sketch parameters for the artist.  
Returns sketch_params : 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 shrunk or expanded.

May return ‘None’ if no sketch parameters were set.

get_snap()  
Returns 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

Only supported by the Agg and MacOSX backends.

get_transform()  
Return the Transform instance used by this 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.

get_url()  
Returns the url

get_visible()  
Return the artist’s visibility

get_window_extent(renderer)  
Get the axes bounding box in display space. Subclasses should override for inclusion in the bounding box “tight” calculation. Default is to return an empty bounding box at 0, 0.
get_zorder()
Return the Artist's zorder.

have_units()
Return True if units are set on the x or y axes

hitlist(event)
List the children of the artist which contain the mouse event event.

is_figure_set()
Returns True if the artist is assigned to a Figure.

is_transform_set()
Returns True if Artist has a transform explicitly set.

pchanged()
Fire an event when property changed, calling all of the registered callbacks.

pick(mouseevent)
call signature:

pick(mouseevent)

each child artist will fire a pick event if mouseevent is over the artist and the artist has picker set

pickable()
Return True if Artist is pickable.

properties()
return a dictionary mapping property name -> value for all Artist props

remove()
Remove the artist from the figure if possible. The effect will not be visible unti
the figure is redrawn, e.g., with matplotlib.axes.Axes.draw_idle(). Call matplotlib.axes.Axes.relim() to update the axes limits if desired.

Note: releim() will not see collections even if the collection was added to axes with autolim = True.

Note: there is no support for removing the artist’s legend entry.

remove_callback(oid)
Remove a callback based on its id.

See also:

add_callback() For adding callbacks

set(**kwargs)
A tkstyle set command, pass kwargs to set properties

set_agg_filter(filter_func)
set agg_filter fuction.

set_alpha(alpha)
Set the alpha value used for blending - not supported on all backends.
`set_animated(b)`
Set the artist’s animation state.

ACCEPTS: [True | False]

`set_axes(axes)`
Set the Axes instance in which the artist resides, if any.

ACCEPTS: an Axes instance

`set_clip_box(clipbox)`
Set the artist’s clip Bbox.

ACCEPTS: a matplotlib.transforms.Bbox instance

`set_clip_on(b)`
Set whether artist uses clipping.

ACCEPTS: [True | False]

`set_clip_path(path, transform=None)`
Set the artist’s clip path, which may be:

• a Patch (or subclass) instance
• a Patch instance, in which case an optional Transform instance may be provided, which will be applied to the path before using it for clipping.
• None, to remove the clipping path

For efficiency, if the path happens to be an axis-aligned rectangle, this method will set the clipping box to the corresponding rectangle and set the clipping path to None.

ACCEPTS: [(Path, Transform) | Patch | None]

`set_contains(picker)`
Replace the contains test used by this artist. The new picker should be a callable function which determines whether the artist is hit by the mouse event:

hit, props = picker(artist, mouseevent)

If the mouse event is over the artist, return hit = True and props is a dictionary of properties you want returned with the contains test.

ACCEPTS: a callable function

`set_figure(fig)`
Set the Figure instance the artist belongs to.

ACCEPTS: a matplotlib.figure.Figure instance

`set_gid(gid)`
Sets the (group) id for the artist

ACCEPTS: an id string

`set_label(s)`
Set the label to s for auto legend.
 ACCEPTS: string or anything printable with ‘%s’ conversion.

**set_lod**(on)**
Set Level of Detail on or off. If on, the artists may examine things like the pixel width of the axes and draw a subset of their contents accordingly

ACCEPTS: [True | False]

**set_path_effects**(path_effects)**
set path_effects, which should be a list of instances of matplotlib.path.effect._Base class or its derivatives.

**set_picker**(picker)**
Set the epsilon for picking used by this artist

*picker* 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 float: if picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if it’s 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
- 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.

ACCEPTS: [None | float | boolean | callable]

**set_rasterized**(rasterized)**
Force rasterized (bitmap) drawing in vector backend output.

Defaults to None, which implies the backend’s default behavior

ACCEPTS: [True | False | None]

**set_sketch_params**(scale=None, length=None, randomness=None)**
Sets the 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**(snap)**
Sets 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.

Only supported by the Agg and MacOSX backends.

**set_transform**
- Set the `Transform` instance used by this artist.
- **ACCEPTS:** `Transform` instance

**set_url**
- Sets the url for the artist.
- **ACCEPTS:** a url string

**set_visible**
- Set the artist's visibility.
- **ACCEPTS:** [True | False]

**set_zorder**
- Set the zorder for the artist. Artists with lower zorder values are drawn first.
- **ACCEPTS:** any number

**update**
- Update the properties of this `Artist` from the dictionary `prop`.

**update_from**
- Copy properties from `other` to `self`.

**zorder = 0**

```python
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 sequence of `Artists`. If a sequence is used, we assume it is a homogeneous sequence (all `Artists` are of the same type) and it is your responsibility to make sure this is so.

**aliased_name**
- 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**
- 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'

**findobj**
- Recursively find all `matplotlib.artist.Artist` instances contained in `self`.

If `match` is not None, it can be
function with signature boolean = match(artist)
used to filter matches.

get_aliases()
Get a dict mapping fullname -> alias for each alias in the ArtistInspector.
e.g., for lines:

{ 'markerfacecolor': 'mfc',
  'linewidth' : 'lw',
}

get_setters()
Get the attribute strings with setters for object. e.g., for a line, return ['markerfacecolor',
  'linewidth', ....].

get_valid_values(attr)
Get the legal arguments for the setter associated with attr.
This is done by querying the docstring of the function set_attr for a line that begins with ACEPTS:
e.g., for a line linestyle, return "[ '- ' | '-.' | '--' | ':' | 'steps' | 'None' ]"

is_alias(o)
Return True if method object o is an alias for another function.

pprint_getters()
Return the getters and actual values as list of strings.

pprint_setters(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(prop=None, leadingspace=2)
If prop is None, return a list of strings of all settable properties and their valid values. Format the output for ReST
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()
return a dictionary mapping property name -> value

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-dependant renderer attributes or making other setup function calls, such as starting and flushing a mixed-mode renderer.

matplotlib.artist.get(obj, property=None)
Return the value of object’s property. property is an optional string for the property you want to return
Example usage:

```python
getp(obj)  # get all the object properties
getp(obj, 'linestyle')  # get the linestyle property
```

`obj` is a `Artist` instance, e.g., `Line2D` or an instance of a `Axes` or `matplotlib.text.Text`. If the `property` is 'somename', this function returns

```python
obj.get_somename()
```

`getp()` can be used to query all the gettable properties with `getp(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:

```python
property or alias = value
```

**Example usage:**

```python
getp(obj)  # get all the object properties
getp(obj, 'linestyle')  # get the linestyle property
```

`obj` is a `Artist` instance, e.g., `Line2D` or an instance of a `Axes` or `matplotlib.text.Text`. If the `property` is 'somename', this function returns

```python
obj.get_somename()
```

`getp()` can be used to query all the gettable properties with `getp(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:

```python
property or alias = value
```

**Example usage:**

```python
getp(obj)  # get all the object properties
getp(obj, 'linestyle')  # get the linestyle property
```

`obj` is a `Artist` instance, e.g., `Line2D` or an instance of a `Axes` or `matplotlib.text.Text`. If the `property` is 'somename', this function returns

```python
obj.get_somename()
```

`getp()` can be used to query all the gettable properties with `getp(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:

```python
property or alias = value
```

**Example usage:**

```python
getp(obj)  # get all the object properties
getp(obj, 'linestyle')  # get the linestyle property
```

`obj` is a `Artist` instance, e.g., `Line2D` or an instance of a `Axes` or `matplotlib.text.Text`. If the `property` is 'somename', this function returns

```python
obj.get_somename()
```

`getp()` can be used to query all the gettable properties with `getp(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:

```python
property or alias = value
```

**Example usage:**

```python
getp(obj)  # get all the object properties
getp(obj, 'linestyle')  # get the linestyle property
```

`obj` is a `Artist` instance, e.g., `Line2D` or an instance of a `Axes` or `matplotlib.text.Text`. If the `property` is 'somename', this function returns

```python
obj.get_somename()
```

`getp()` can be used to query all the gettable properties with `getp(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:

```python
property or alias = value
```

**Example usage:**

```python
getp(obj)  # get all the object properties
getp(obj, 'linestyle')  # get the linestyle property
```

`obj` is a `Artist` instance, e.g., `Line2D` or an instance of a `Axes` or `matplotlib.text.Text`. If the `property` is 'somename', this function returns

```python
obj.get_somename()
```

`getp()` can be used to query all the gettable properties with `getp(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:

```python
property or alias = value
```

**Example usage:**

```python
getp(obj)  # get all the object properties
getp(obj, 'linestyle')  # get the linestyle property
```

`obj` is a `Artist` instance, e.g., `Line2D` or an instance of a `Axes` or `matplotlib.text.Text`. If the `property` is 'somename', this function returns

```python
obj.get_somename()
```

`getp()` can be used to query all the gettable properties with `getp(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:

```python
property or alias = value
```

**Example usage:**

```python
getp(obj)  # get all the object properties
getp(obj, 'linestyle')  # get the linestyle property
```

`obj` is a `Artist` instance, e.g., `Line2D` or an instance of a `Axes` or `matplotlib.text.Text`. If the `property` is 'somename', this function returns

```python
obj.get_somename()
```

`getp()` can be used to query all the gettable properties with `getp(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:

```python
property or alias = value
```
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:

```python
>>> setp(line, 'linestyle')
linestyle: [ ' '-' | '--' | '-.' | ':' | 'steps' | 'None' ]
```

If you want to see all the properties that can be set, and their possible values, you can do:

```python
>>> setp(line)
... long output listing omitted
```

The `setp()` function operates on a single instance or a list 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. For example, suppose you have a list of two lines, the following will make both lines thicker and red:

```python
>>> x = arange(0,1.0,0.01)
>>> y1 = sin(2*pi*x)
>>> y2 = sin(4*pi*x)
>>> lines = plot(x, y1, x, y2)
>>> setp(lines, linewidth=2, color='r')
```

The `setp()` function works with the MATLAB style string/value pairs or with python kwargs. For example, the following are equivalent:

```python
>>> setp(lines, 'linewidth', 2, 'color', 'r')  # MATLAB style
>>> setp(lines, linewidth=2, color='r')       # python style
```

## 52.2 matplotlib.lines

This module contains all the 2D line class which can draw with a variety of line styles, markers and colors.

```python
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='full', 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, eg one can create “stepped” lines in various styles.

Create a `Line2D` instance with `x` and `y` data in sequences `xdata`, `ydata`.

The kwargs are `Line2D` properties:
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>([Path, Transform]</td>
</tr>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>data</td>
<td>2D array (rows are x, y) or two 1D arrays</td>
</tr>
<tr>
<td>drawstyle</td>
<td>['default'</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>['full'</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with '%s' conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['-.'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>marker</td>
<td>unknown</td>
</tr>
<tr>
<td>markeredgecolor or mec</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markeredgewidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>
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.

**contains** *(mouseevent)*
Test whether the mouse event occurred on the line. The pick radius determines the precision of the location test (usually within five points of the value). Use `get_pickradius()` or `set_pickradius()` to view or modify it.

Returns `True` if any values are within the radius along with `{ 'ind': pointlist }`, where `pointlist` is the set of points within the radius.

TODO: sort returned indices by distance

**draw** *(artist, renderer, *args, **kwargs)*
Draw the Line with `renderer` unless visibility is False

**drawStyleKeys** = ['default', 'steps-mid', 'steps-pre', 'steps-post', 'steps']

**drawStyles** = {'default': '_draw_lines', 'steps-mid': '_draw_steps_mid', 'steps': '_draw_steps_pre', 'steps-pre': '_draw_steps_pre', 'steps-post': '_draw_steps_post'}

**fillStyles** = ('full', 'left', 'right', 'bottom', 'top', 'none')

**filled_markers** = (o', 'v', '^', '<', '>', '8', 's', 'p', '*', 'h', 'H', 'D', 'd')

**get_aa()**
alias for `get_antialiased`

**get_antialiased()**

**get_c()**
alias for `get_color`

**get_color()**

**get_dash_capstyle()**
Get the cap style for dashed linestyles

**get_dash_joinstyle()**
Get the join style for dashed linestyles

**get_data**(orig=True)
Return the xdata, ydata.

If `orig` is `True`, return the original data.

**get_drawstyle()**

**get_fillstyle()**
return the marker fillstyle
get linestyle()

get linewidth()

get ls()
    alias for get linestyle
get lw()
    alias for get linewidth

get marker()

get markeredgecolor()

get markeredgewidth()

get markerfacecolor()

get markerfacecoloralt()

get markersize()

get markevery()
    return the markevery setting
get mec()
    alias for get markeredgecolor
get mew()
    alias for get markeredgewidth

get mfc()
    alias for get markerfacecolor
get mfcalt(alt=False)
    alias for get markerfacecoloralt

get ms()
    alias for get markersize

get path()
    Return the Path object associated with this line.

get pickradius()
    return the pick radius used for containment tests

get solid capstyle()
    Get the cap style for solid linestyles
get_solid_joinstyle()
   Get the join style for solid linestyles

get_window_extent(renderer)

get_xdata(orig=True)
   Return the xdata.
   
   If orig is True, return the original data, else the processed data.

get_xydata()
   Return the xy data as a Nx2 numpy array.

get_ydata(orig=True)
   Return the ydata.
   
   If orig is True, return the original data, else the processed data.

is_dashed()
   return True if line is dashstyle

lineStyles = {‘‘: ‘_draw_nothing’, ‘ ‘: ‘_draw_nothing’, ‘None’: ‘_draw_nothing’, ‘-‘: ‘_draw_dashed’, ‘-.’: ‘_draw_dash_dot’, ‘-.’: ‘_draw_solid’}


recache(always=False)

recache_always()

set_aa(val)
   alias for set_antialiased

set_antialiased(b)
   True if line should be drawin with antialiased rendering
   
   ACCEPTS: [True | False]

set_axes(ax)
   Set the Axes instance in which the artist resides, if any.
   
   ACCEPTS: an Axes instance

set_c(val)
   alias for set_color

set_color(color)
   Set the color of the line
   
   ACCEPTS: any matplotlib color

set_dash_capstyle(s)
   Set the cap style for dashed linestyles
ACCEPES: ['butt' | 'round' | 'projecting']

set_dash_joinstyle(s)
Set the join style for dashed linestyles ACCEPES: ['miter' | 'round' | 'bevel']

set_dashes(seq)
Set the dash sequence, sequence of dashes with on off ink in points. If seq is empty or if seq = (None, None), the linestyle will be set to solid.
ACCEPES: sequence of on/off ink in points

set_data(*args)
Set the x and y data
ACCEPES: 2D array (rows are x, y) or two 1D arrays

set_drawstyle(drawstyle)
Set the drawstyle of the plot
‘default’ connects the points with lines. The steps variants produce step-plots. ‘steps’ is equivalent to ‘steps-pre’ and is maintained for backward-compatibility.
ACCEPES: ['default' | ‘steps’ | ‘steps-pre’ | ‘steps-mid’ | ‘steps-post’]

set_fillstyle(fs)
Set the marker fill style; ‘full’ means fill the whole marker. ‘none’ means no filling; other options are for half-filled markers.
ACCEPES: ['full' | ‘left’ | ‘right’ | ‘bottom’ | top | ‘none’]

set_linestyle(linestyle)
Set the linestyle of the line (also accepts drawstyles)

<table>
<thead>
<tr>
<th>linestyle</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'-'</td>
<td>solid</td>
</tr>
<tr>
<td>'--'</td>
<td>dashed</td>
</tr>
<tr>
<td>'-.'</td>
<td>dash_dot</td>
</tr>
<tr>
<td>':'</td>
<td>dotted</td>
</tr>
<tr>
<td>'None'</td>
<td>draw nothing</td>
</tr>
<tr>
<td>' '</td>
<td>draw nothing</td>
</tr>
<tr>
<td>''</td>
<td>draw nothing</td>
</tr>
</tbody>
</table>

‘steps’ is equivalent to ‘steps-pre’ and is maintained for backward-compatibility.

See also:

set_drawstyle() To set the drawing style (stepping) of the plot.

ACCEPES: ['-' | '--' | '-' | '.' | ':' | 'None' | ' ' | ' ']

and any drawstyle in combination with a linestyle, e.g., ‘steps--’.

set_linewidth(w)
Set the line width in points
ACCEPES: float value in points
set_ls(val)
    alias for set_linestyle

set_lw(val)
    alias for set_linewidth

set_marker(marker)
    Set the line marker
    Parameters marker: marker style :
        See markers for full description of possible argument

set_markeredgecolor(ec)
    Set the marker edge color
    ACCEPTS: any matplotlib color

set_markeredgewidth(ew)
    Set the marker edge width in points
    ACCEPTS: float value in points

set_markerfacecolor(fc)
    Set the marker face color.
    ACCEPTS: any matplotlib color

set_markerfacecoloralt(fc)
    Set the alternate marker face color.
    ACCEPTS: any matplotlib color

set_markersize(sz)
    Set the marker size in points
    ACCEPTS: float

set_markevery(every)
    Set the markevery property to subsample the plot when using markers. e.g., if markevery=5, every 5-th marker will be plotted. every can be
    None Every point will be plotted
    an integer N Every N-th marker will be plotted starting with marker 0
    A length-2 tuple of integers every=(start, N) will start at point start and plot every N-th marker
    ACCEPTS: None | integer | (startind, stride)

set_mec(val)
    alias for set_markeredgecolor

set_mew(val)
    alias for set_markeredgewidth

set_mfc(val)
    alias for set_markerfacecolor

set_mfcalt(val)
    alias for set_markerfacecoloralt
set_ms(val)
    alias for set_markersize

set_picker(p)
    Sets the event picker details for the line.
    ACCEPTS: float distance in points or callable pick function fn(artist, event)

set_pickradius(d)
    Sets the pick radius used for containment tests
    ACCEPTS: float distance in points

set_solid_capstyle(s)
    Set the cap style for solid linestyles
    ACCEPTS: ['butt', 'round', 'projecting']

set_solid_joinstyle(s)
    Set the join style for solid linestyles ACCEPTS: ['miter', 'round', 'bevel']

set_transform(t)
    set the Transformation instance used by this artist
    ACCEPTS: a matplotlib.transforms.Transform instance

set_xdata(x)
    Set the data np.array for x
    ACCEPTS: 1D array

set_ydata(y)
    Set the data np.array for y
    ACCEPTS: 1D array

update_from(other)
    copy properties from other to self

validCap = ('butt', 'round', 'projecting')

validJoin = ('miter', 'round', 'bevel')

zorder = 2

class matplotlib.lines.VertexSelector(line)
    Manage the callbacks to maintain a list of selected vertices for matplotlib.lines.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 = plt.figure()
ax = fig.add_subplot(111)
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 matplotlib.lines.Line2D instance. The line should already be added to some matplotlib.axes.Axes instance and should have the picker property set.

onpick(event)
    When the line is picked, update the set of selected indicies.

process_selected(ind, xs, ys)
    Default “do nothing” implementation of the process_selected() method.

    ind are the indices of the selected vertices. xs and ys are the coordinates of the selected vertices.

matplotlib.lines.segment_hits(cx, cy, x, y, radius)
    Determine if any line segments are within radius of a point. Returns the list of line segments that are within that radius.

52.3 matplotlib.patches

class matplotlib.patches.Arc(xy, width, height, angle=0.0, theta1=0.0, theta2=360.0, **kwargs)
    Bases: matplotlib.patches.Ellipse

An elliptical arc. Because it performs various optimizations, it can not 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.

The following args are supported:

xy center of ellipse

width length of horizontal axis

height length of vertical axis

angle rotation in degrees (anti-clockwise)

theta1 starting angle of the arc in degrees
**theta2** ending angle of the arc in degrees

If *theta1* and *theta2* are not provided, the arc will form a complete ellipse.

Valid kwargs are:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased</td>
<td>or aa [True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor</td>
<td>or ec mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>facecolor</td>
<td>or fc mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[‘/’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyles</td>
<td>or ls [‘solid’</td>
</tr>
<tr>
<td>linewidth</td>
<td>or lw float or None for default</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

**draw**(artist, renderer, *args, **kwargs)

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.


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 by 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:
   

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 `matplotlib.path.Path.arc()`.

```python
class matplotlib.patches.Arrow(x, y, dx, dy, width=1.0, **kwargs)
    Bases: matplotlib.patches.Patch

    An arrow patch.

    Draws an arrow, starting at (x, y), direction and length given by (dx, dy) the width of the arrow is scaled by `width`.

    Valid kwargs are:
```

```
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>facecolor</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[''</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle</td>
<td>['solid'</td>
</tr>
<tr>
<td>linewidth</td>
<td>float or None for default</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

get_patch_transform()

get_path()

class matplotlib.patches.ArrowStyle
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

<table>
<thead>
<tr>
<th>Class</th>
<th>Name</th>
<th>Attrs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Curve</td>
<td>-</td>
<td>None</td>
</tr>
<tr>
<td>CurveB</td>
<td>-&gt;</td>
<td>head_length=0.4,head_width=0.2</td>
</tr>
<tr>
<td>BracketB</td>
<td>[</td>
<td>widthB=1.0,lengthB=0.2,angleB=None</td>
</tr>
<tr>
<td>CurveFilledB</td>
<td>-&gt;</td>
<td>head_length=0.4,head_width=0.2</td>
</tr>
<tr>
<td>CurveA</td>
<td>&lt;</td>
<td>head_length=0.4,head_width=0.2</td>
</tr>
<tr>
<td>CurveAB</td>
<td>&lt;-&gt;</td>
<td>head_length=0.4,head_width=0.2</td>
</tr>
<tr>
<td>CurveFilledA</td>
<td>&lt;[-]</td>
<td>head_length=0.4,head_width=0.2</td>
</tr>
<tr>
<td>CurveFilledAB</td>
<td>&lt;[-]</td>
<td>head_length=0.4,head_width=0.2</td>
</tr>
<tr>
<td>BracketA</td>
<td>]</td>
<td>widthA=1.0,lengthA=0.2,angleA=None</td>
</tr>
<tr>
<td>BracketAB</td>
<td>]-&gt;[</td>
<td>widthA=1.0,lengthA=0.2,angleA=None,widthB=1.0,lengthB=0.2,angleB=None</td>
</tr>
<tr>
<td>Fancy</td>
<td>fancy</td>
<td>head_length=0.4,head_width=0.4,tail_width=0.4</td>
</tr>
<tr>
<td>Simple</td>
<td>simple</td>
<td>head_length=0.5,head_width=0.5,tail_width=0.2</td>
</tr>
<tr>
<td>Wedge</td>
<td>wedge</td>
<td>tail_width=0.3,shrink_factor=0.5</td>
</tr>
<tr>
<td>BarAB</td>
<td>[-]</td>
<td>widthA=1.0,angleA=None,widthB=1.0,angleB=None</td>
</tr>
</tbody>
</table>

An instance of any arrow style class is an 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 witch the arrow will be drawn. mutation_size and aspect_ratio has a 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.
class BarAB(widthA=1.0, angleA=None, widthB=1.0, angleB=None)
    Bases: matplotlib.patches._Bracket

    An arrow with a bar(|) at both ends.
    widthA width of the bracket
    lengthA length of the bracket
    angleA angle between the bracket and the line
    widthB width of the bracket
    lengthB length of the bracket
    angleB angle between the bracket and the line

class ArrowStyle.BraceletA(widthA=1.0, lengthA=0.2, angleA=None)
    Bases: matplotlib.patches._Bracket

    An arrow with a bracket([) at its end.
    widthA width of the bracket
    lengthA length of the bracket
    angleA angle between the bracket and the line

class ArrowStyle.BraceletAB(widthA=1.0, lengthA=0.2, angleA=None, widthB=1.0, lengthB=0.2, angleB=None)
    Bases: matplotlib.patches._Bracket
An arrow with a bracket([) at both ends.
- **widthA** width of the bracket
- **lengthA** length of the bracket
- **angleA** angle between the bracket and the line
- **widthB** width of the bracket
- **lengthB** length of the bracket
- **angleB** angle between the bracket and the line

```
class ArrowStyle.BracketB(widthB=1.0, lengthB=0.2, angleB=None)
```
Bases: matplotlib.patches._Bracket

An arrow with a bracket([) at its end.
- **widthB** width of the bracket
- **lengthB** length of the bracket
- **angleB** angle between the bracket and the line

```
class ArrowStyle.CurveA(head_length=0.4, head_width=0.2)
```
Bases: matplotlib.patches._Curve

A simple curve without any arrow head.

```
class ArrowStyle.CurveAB(head_length=0.4, head_width=0.2)
```
Bases: matplotlib.patches._Curve

An arrow with heads both at the begin and the end point.
- **head_length** length of the arrow head
- **head_width** width of the arrow head

```
class ArrowStyle.CurveB(head_length=0.4, head_width=0.2)
```
Bases: matplotlib.patches._Curve

An arrow with a head at its end point.
- **head_length** length of the arrow head
- **head_width** width of the arrow head

```
class ArrowStyle.CurveFilledA(head_length=0.4, head_width=0.2)
```
Bases: matplotlib.patches._Curve

An arrow with filled triangle head at the begin.
- **head_length** length of the arrow head
- **head_width** width of the arrow head

```
class ArrowStyle.CurveFilledAB(head_length=0.4, head_width=0.2)
```
Bases: matplotlib.patches._Curve

An arrow with filled triangle heads both at the begin and the end point.
- **head_length** length of the arrow head
- **head_width** width of the arrow head
class ArrowStyle.CurveFilledB(head_length=0.4, head_width=0.2)
    Bases: matplotlib.patches._Curve

    An arrow with filled triangle head at the end.
    head_length length of the arrow head
    head_width width of the arrow head

class ArrowStyle.Fancy(head_length=0.4, head_width=0.4, tail_width=0.4)
    Bases: matplotlib.patches._Base

    A fancy arrow. Only works with a quadratic bezier curve.
    head_length length of the arrow head
    head_width width of the arrow head
    tail_width width of the arrow tail
    transmute(path, mutation_size, linewidth)

class ArrowStyle.Simple(head_length=0.5, head_width=0.5, tail_width=0.2)
    Bases: matplotlib.patches._Base

    A simple arrow. Only works with a quadratic bezier curve.
    head_length length of the arrow head
    head_width width of the arrow head
    tail_width width of the arrow tail
    transmute(path, mutation_size, linewidth)

class ArrowStyle.Wedge(tail_width=0.3, shrink_factor=0.5)
    Bases: matplotlib.patches._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.
    tail_width width of the tail
    shrink_factor fraction of the arrow width at the middle point
    transmute(path, mutation_size, linewidth)

class matplotlib.patches.BoxStyle
    Bases: matplotlib.patches._Style

    BoxStyle is a container class which defines several boxstyle classes, which are used for
    FancyBoxPatch.

    A style object can be created as:

    BoxStyle.Round(pad=0.2)

    or:

    BoxStyle("Round", pad=0.2)

    or:
BoxStyle("Round, pad=0.2")

Following boxstyle classes are defined.

<table>
<thead>
<tr>
<th>Class</th>
<th>Name</th>
<th>Attrs</th>
</tr>
</thead>
<tbody>
<tr>
<td>LArrow</td>
<td>larrow</td>
<td>pad=0.3</td>
</tr>
<tr>
<td>RArrow</td>
<td>rarrow</td>
<td>pad=0.3</td>
</tr>
<tr>
<td>Round</td>
<td>round</td>
<td>pad=0.3, rounding_size=None</td>
</tr>
<tr>
<td>Round4</td>
<td>round4</td>
<td>pad=0.3, rounding_size=None</td>
</tr>
<tr>
<td>Roundtooth</td>
<td>roundtooth</td>
<td>pad=0.3, tooth_size=None</td>
</tr>
<tr>
<td>Sawtooth</td>
<td>sawtooth</td>
<td>pad=0.3, tooth_size=None</td>
</tr>
<tr>
<td>Square</td>
<td>square</td>
<td>pad=0.3</td>
</tr>
</tbody>
</table>

An instance of any boxstyle class is an callable object, whose call signature is:

```python
__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.
Bases: matplotlib.patches._Base

(Left) Arrow Box

\texttt{transmute}(x0, y0, width, height, mutation_size)

class BoxStyle.RArrow(\texttt{pad}=0.3)

Bases: matplotlib.patches.LArrow

(Right) Arrow Box

\texttt{transmute}(x0, y0, width, height, mutation_size)

class BoxStyle.Round(\texttt{pad}=0.3, rounding_size=None)

Bases: matplotlib.patches._Base

A box with round corners.

\texttt{pad} amount of padding

\texttt{rounding_size} rounding radius of corners. \texttt{pad} if None

\texttt{transmute}(x0, y0, width, height, mutation_size)

class BoxStyle.Round4(\texttt{pad}=0.3, rounding_size=None)

Bases: matplotlib.patches._Base

Another box with round edges.

\texttt{pad} amount of padding

\texttt{rounding_size} rounding size of edges. \texttt{pad} if None

\texttt{transmute}(x0, y0, width, height, mutation_size)

class BoxStyle.Roundtooth(\texttt{pad}=0.3, tooth_size=None)

Bases: matplotlib.patches.Sawtooth

A roundtooth(?) box.

\texttt{pad} amount of padding

\texttt{tooth_size} size of the sawtooth. \texttt{pad} if None

\texttt{transmute}(x0, y0, width, height, mutation_size)

class BoxStyle.Sawtooth(\texttt{pad}=0.3, tooth_size=None)

Bases: matplotlib.patches._Base

A sawtooth box.

\texttt{pad} amount of padding

\texttt{tooth_size} size of the sawtooth. \texttt{pad} if None

\texttt{transmute}(x0, y0, width, height, mutation_size)

class BoxStyle.Square(\texttt{pad}=0.3)

Bases: matplotlib.patches._Base

A simple square box.
**pad** amount of padding

**transmute**(x0, y0, width, height, mutation_size)

class `matplotlib.patches.Circle`(xy, radius=5, **kwargs)

Bases: `matplotlib.patches.Ellipse`

A circle patch.

Create true circle at center xy = (x, y) with given `radius`. Unlike `CirclePolygon` which is a polygonal approximation, this uses Bézier splines and is much closer to a scale-free circle.

Valid kwarg are:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an <code>Axes</code> instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td><code>matplotlib</code> color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or ec</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>facecolor or fc</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[‘-’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>[‘solid’</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float or None for default</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td><code>Transform</code> instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

**get_radius()**

return the radius of the circle

**radius**

return the radius of the circle
**set_radius** *(radius)*
Set the radius of the circle

ACCEPTS: float

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 kwargs are:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or ec</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>facecolor or fc</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[''</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['solid'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float or None for default</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>
class matplotlib.patches.ConnectionPatch(xyA, xyB, coordsA, coordsB=None, axesA=None, axesB=None, arrowstyle='-', arrow_transmuter=None, connectionstyle='arc3', connector=None, 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 ConnectionPatch class is to make connecting lines between two points (possibly in different axes).

Connect point xyA in coordsA with point xyB in coordsB

Valid keys are

<table>
<thead>
<tr>
<th>Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>arrowstyle</td>
<td>the arrow style</td>
</tr>
<tr>
<td>connectionstyle</td>
<td>the connection style</td>
</tr>
<tr>
<td>relpos</td>
<td>default is (0.5, 0.5)</td>
</tr>
<tr>
<td>patchA</td>
<td>default is bounding box of the text</td>
</tr>
<tr>
<td>patchB</td>
<td>default is None</td>
</tr>
<tr>
<td>shrinkA</td>
<td>default is 2 points</td>
</tr>
<tr>
<td>shrinkB</td>
<td>default is 2 points</td>
</tr>
<tr>
<td>mutation_scale</td>
<td>default is text size (in points)</td>
</tr>
<tr>
<td>mutation_aspect</td>
<td>default is 1.</td>
</tr>
<tr>
<td>?</td>
<td>any key for matplotlib.patches.PathPatch</td>
</tr>
</tbody>
</table>

coordsA and coordsB are strings that indicate the coordinates of xyA and xyB.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘figure points’</td>
<td>points from the lower left corner of the figure</td>
</tr>
<tr>
<td>‘figure pixels’</td>
<td>pixels from the lower left corner of the figure</td>
</tr>
<tr>
<td>‘figure fraction’</td>
<td>0,0 is lower left of figure and 1,1 is upper, right</td>
</tr>
<tr>
<td>‘axes points’</td>
<td>points from lower left corner of axes</td>
</tr>
<tr>
<td>‘axes pixels’</td>
<td>pixels from lower left corner of axes</td>
</tr>
<tr>
<td>‘axes fraction’</td>
<td>0,1 is lower left of axes and 1,1 is upper right</td>
</tr>
<tr>
<td>‘data’</td>
<td>use the coordinate system of the object being annotated (default)</td>
</tr>
<tr>
<td>‘offset points’</td>
<td>Specify an offset (in points) from the xy value</td>
</tr>
<tr>
<td>‘polar’</td>
<td>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.</td>
</tr>
</tbody>
</table>

draw(renderer)
Draw.

**get_annotation_clip()**

Return `annotation_clip` attribute. See `set_annotation_clip()` for the meaning of return values.

**get_path_in_displaycoord()**

Return the mutated path of the arrow in the display coord

**set_annotation_clip(b)**

set `annotation_clip` attribute.

- **True**: the annotation will only be drawn when `self.xy` is inside the axes.
- **False**: the annotation will always be drawn regardless of its position.
- **None**: the self.xy will be checked only if `xycoords` is “data”

---

### class matplotlib.patches.ConnectionStyle

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:

<table>
<thead>
<tr>
<th>Class</th>
<th>Name</th>
<th>Attrs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Angle</td>
<td>angle</td>
<td><code>angleA=90,angleB=0,rad=0.0</code></td>
</tr>
<tr>
<td>Angle3</td>
<td>angle3</td>
<td><code>angleA=90,angleB=0</code></td>
</tr>
<tr>
<td>Arc</td>
<td>arc</td>
<td><code>angleA=0,angleB=0,armA=None,armB=None,rad=0.0</code></td>
</tr>
<tr>
<td>Arc3</td>
<td>arc3</td>
<td><code>rad=0.0</code></td>
</tr>
<tr>
<td>Bar</td>
<td>bar</td>
<td><code>armA=0.0,armB=0.0,fraction=0.3,angle=None</code></td>
</tr>
</tbody>
</table>

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.

### class Angle

```
Angle(angleA=90, angleB=0, rad=0.0)
```

Bases: matplotlib.patches._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 crosses the start (or end) point and has an angle of angleA (or angleB). 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

```python
class ConnectionStyle.Angle3(angleA=90, angleB=0)
Bases: matplotlib.patches._Base

Creates a simple quadratic bezier curve between two points. The middle control points is placed at the intersecting point of two lines which crosses the start (or end) point and has an angle of angleA (or angleB).

- **angleA**: starting angle of the path
- **angleB**: ending angle of the path

```python
class ConnectionStyle.Arc(angleA=0, angleB=0, armA=None, armB=None, rad=0.0)
Bases: matplotlib.patches._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

```python
class ConnectionStyle.Arc3(rad=0.0)
Bases: matplotlib.patches._Base

Creates a simple quadratic bezier curve between two points. The curve is created so that the middle control points (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

```python
class ConnectionStyle.Bar(armA=0.0, armB=0.0, fraction=0.3, angle=None)
Bases: matplotlib.patches._Base

A line with angle between A and B with armA and armB. One of the arm is extend so that they are connected in a right angle. The length of armA is determined by $(armA + fraction \times AB\ distance)$. Same for armB.

- **armA**: minimum length of armA
\textit{armB} : minimum length of armB

\textit{fraction} [a fraction of the distance between two points that] will be added to armA and armB.

\textit{angle} [angle of the connecting line (if None, parallel to A and B)]

\texttt{connect(posA, posB)}

\begin{verbatim}
class matplotlib.patches.Ellipse(xy, width, height, angle=0.0, **kwargs)

Bases: matplotlib.patches.Patch

A scale-free ellipse.

xy center of ellipse

width total length (diameter) of horizontal axis

height total length (diameter) of vertical axis

angle rotation in degrees (anti-clockwise)

Valid kwargs are:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or ec</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>facecolor or fc</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[‘/’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>[‘solid’</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float or None for default</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>
\end{verbatim}
contains\((ev)\)

\texttt{get\_patch\_transform()}\n
\texttt{get\_path()}\n
Return the vertices of the rectangle

class \texttt{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: \texttt{matplotlib.patches.Polygon}\n
Like Arrow, but lets you set head width and head height independently.

**Constructor arguments**

\begin{itemize}
  \item \texttt{width}: float (default: 0.001) width of full arrow tail
  \item \texttt{length\_includes\_head}: [True | False] (default: False) True if head is to be counted in calculating the length.
  \item \texttt{head\_width}: float or None (default: 3*width) total width of the full arrow head
  \item \texttt{head\_length}: float or None (default: 1.5 * head\_width) length of arrow head
  \item \texttt{shape}: ['full', 'left', 'right'] (default: 'full') draw the left-half, right-half, or full arrow
  \item \texttt{overhang}: float (default: 0) fraction that the arrow is swept back (0 overhang means triangular shape). Can be negative or greater than one.
  \item \texttt{head\_starts\_at\_zero}: [True | False] (default: False) if True, the head starts being drawn at coordinate 0 instead of ending at coordinate 0.
\end{itemize}

Other valid kwargs (inherited from \texttt{Patch}) are:
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or ec</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>facecolor or fc</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>['/'</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['solid'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float or None for default</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

class matplotlib.patches.FancyArrowPatch(posA=None, posB=None, path=None, arrowstyle='simple', arrow_transmuter=None, connectionstyle='arc3', connector=None, patchA=None, patchB=None, shrinkA=2.0, shrinkB=2.0, mutation_scale=1.0, mutation_aspect=None, dpi_cor=1.0, **kwargs)

Bases: matplotlib.patches.Patch

A fancy arrow patch. It draws an arrow using the :class:ArrowStyle.

If posA and posB is given, a path connecting two point are created according to the connectionstyle. The path will be clipped with patchA and patchB and further shirnked by shrinkA and shrinkB. An arrow is drawn along this resulting path using the arrowstyle parameter. If path provided, an arrow is drawn along this path and patchA, patchB, shrinkA, and shrinkB are ignored.

The connectionstyle describes how posA and posB are connected. It can be an instance of the ConnectionStyle class (matplotlib.patches.ConnectionStyle) or a string of the connectionstyle name, with optional comma-separated attributes. The following connection styles are available.
Matplotlib, Release 1.3.0

<table>
<thead>
<tr>
<th>Class</th>
<th>Name</th>
<th>Attrs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Angle</td>
<td>angle</td>
<td>angleA=90,angleB=0,rad=0.0</td>
</tr>
<tr>
<td>Angle3</td>
<td>angle3</td>
<td>angleA=90,angleB=0</td>
</tr>
<tr>
<td>Arc</td>
<td>arc</td>
<td>angleA=0,angleB=0,armA=None,armB=None,rad=0.0</td>
</tr>
<tr>
<td>Arc3</td>
<td>arc3</td>
<td>rad=0.0</td>
</tr>
<tr>
<td>Bar</td>
<td>bar</td>
<td>armA=0.0,armB=0.0,fraction=0.3,angle=None</td>
</tr>
</tbody>
</table>

The *arrowstyle* describes how the fancy arrow will be drawn. It can be string of the available arrowstyle names, with optional comma-separated attributes, or one of the ArrowStyle instance. The optional attributes are meant to be scaled with the *mutation_scale*. The following arrow styles are available.

<table>
<thead>
<tr>
<th>Class</th>
<th>Name</th>
<th>Attrs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Curve</td>
<td>-</td>
<td>None</td>
</tr>
<tr>
<td>CurveB</td>
<td>-&gt;</td>
<td>head_length=0.4,head_width=0.2</td>
</tr>
<tr>
<td>BracketB</td>
<td>-[</td>
<td>widthB=1.0,lengthB=0.2,angleB=None</td>
</tr>
<tr>
<td>Curve-FilledB</td>
<td>-</td>
<td>&gt;</td>
</tr>
<tr>
<td>CurveA</td>
<td>&lt;-</td>
<td>head_length=0.4,head_width=0.2</td>
</tr>
<tr>
<td>CurveAB</td>
<td>&lt;-&gt;</td>
<td>head_length=0.4,head_width=0.2</td>
</tr>
<tr>
<td>Curve-FilledA</td>
<td>&lt;</td>
<td>-</td>
</tr>
<tr>
<td>Curve-FilledAB</td>
<td></td>
<td>&lt;</td>
</tr>
<tr>
<td>BracketA</td>
<td>]-</td>
<td>widthA=1.0,lengthA=0.2,angleA=None</td>
</tr>
<tr>
<td>BracketAB</td>
<td>]-[</td>
<td>widthA=1.0,lengthA=0.2,angleA=None,widthB=1.0,lengthB=0.2,angleB=None</td>
</tr>
<tr>
<td>Fancy</td>
<td>fancy</td>
<td>head_length=0.4,head_width=0.4,tail_width=0.4</td>
</tr>
<tr>
<td>Simple</td>
<td>simple</td>
<td>head_length=0.5,head_width=0.5,tail_width=0.2</td>
</tr>
<tr>
<td>Wedge</td>
<td>wedge</td>
<td>tail_width=0.3,shrink_factor=0.5</td>
</tr>
<tr>
<td>BarAB</td>
<td></td>
<td>-</td>
</tr>
</tbody>
</table>

*mutation_scale* [a value with which attributes of arrowstyle] (e.g., head_length) will be scaled. default=1.

*mutation_aspect* [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. default=None.

Valid kwargs are:
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an <code>Axes</code> instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td><code>matplotlib</code> color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>facecolor</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[ ‘/’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle</td>
<td>[‘solid’</td>
</tr>
<tr>
<td>linewidth</td>
<td>float or None for default</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td><code>Transform</code> instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

draw(renderer)

get_arrowstyle()  
Return the arrowstyle object

get_connectionstyle()  
Return the ConnectionStyle instance

get_dpi_cor()  
dpi_cor is currently used for linewidth-related things and shink factor. Mutation scale is not affected by this.

get_mutation_aspect()  
Return the aspect ratio of the bbox mutation.

get_mutation_scale()  
Return the mutation scale.
get_path()  
return the path of the arrow in the data coordinate. Use get_path_in_displaycoord() method to  
retrieve the arrow path in the display coord.

get_path_in_displaycoord()  
Return the mutated path of the arrow in the display coord

set_arrowstyle(arrowstyle=None, **kw)  
Set the arrow style.  

arrowstyle can be a string with arrowstyle name with optional comma-separated attributes.  
Alternatively, the attrs can be provided as keywords.

set_arrowstyle("Fancy,head_length=0.2") set_arrowstyle("fancy", head_length=0.2)  
Old attrs simply are forgotten.

Without argument (or with arrowstyle=None), return available box styles as a list of strings.

set_connectionstyle(connectionstyle, **kw)  
Set the connection style.  

connectionstyle can be a string with connectionstyle name with optional comma-separated attributes. Alternatively, the attrs can be provided as keywords.

set_connectionstyle("arc,angleA=0,armA=30,rad=10") set_connectionstyle("arc", angleA=0,armA=30,rad=10)  
Old attrs simply are forgotten.

Without argument (or with connectionstyle=None), return available styles as a list of strings.

set_dpi_cor(dpi_cor)  
dpi_cor is currently used for linewidth-related things and shink factor. Mutation scale is not  
affected by this.

set_mutation_aspect(aspect)  
Set the aspect ratio of the bbox mutation.  

ACCEPTS: float

set_mutation_scale(scale)  
Set the mutation scale.  

ACCEPTS: float

set_patchA(patchA)  
set the begin patch.

set_patchB(patchB)  
set the begin patch

set_positions(posA, posB)  
set the begin end end positions of the connecting path. Use current vlauie if None.

class matplotlib.patches.FancyBboxPatch(xy, width, height, boxstyle='round',  
bbox_transmuter=None, mutation_scale=1.0,  
mutation_aspect=None, **kwargs)
Draw a fancy box around a rectangle with lower left at \(xy=(x, y)\) with specified width and height. 

**FancyBboxPatch** class is similar to **Rectangle** class, but it draws a fancy box around the rectangle. The transformation of the rectangle box to the fancy box is delegated to the **BoxTransmuterBase** and its derived classes.

\(xy\) = lower left corner

**width, height**

**boxstyle** determines what kind of fancy box will be drawn. It can be a string of the style name with a comma separated attribute, or an instance of **BoxStyle**. Following box styles are available.

<table>
<thead>
<tr>
<th>Class</th>
<th>Name</th>
<th>Attrs</th>
</tr>
</thead>
<tbody>
<tr>
<td>LArrow</td>
<td>larrow</td>
<td>pad=0.3</td>
</tr>
<tr>
<td>RArrow</td>
<td>rarrow</td>
<td>pad=0.3</td>
</tr>
<tr>
<td>Round</td>
<td>round</td>
<td>pad=0.3,rounding_size=None</td>
</tr>
<tr>
<td>Round4</td>
<td>round4</td>
<td>pad=0.3,rounding_size=None</td>
</tr>
<tr>
<td>Roundtooth</td>
<td>roundtooth</td>
<td>pad=0.3,tooth_size=None</td>
</tr>
<tr>
<td>Sawtooth</td>
<td>sawtooth</td>
<td>pad=0.3,tooth_size=None</td>
</tr>
<tr>
<td>Square</td>
<td>square</td>
<td>pad=0.3</td>
</tr>
</tbody>
</table>

**mutation_scale** : a value with which attributes of boxstyle (e.g., pad) will be scaled. default=1.

**mutation_aspect** : 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. default=None.

Valid kwargs are:
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or ec</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>facecolor or fc</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[‘/’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>[‘solid’</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float or None for default</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

get_bbox()

get_boxstyle()
    Return the boxstyle object

get_height()
    Return the height of the rectangle

get_mutation_aspect()
    Return the aspect ratio of the bbox mutation.

get_mutation_scale()
    Return the mutation scale.

get_path()
    Return the mutated path of the rectangle

get_width()
Return the width of the rectangle

get_x()

Return the left coord of the rectangle

get_y()

Return the bottom coord of the rectangle

set_bounds(*args)

Set the bounds of the rectangle: l,b,w,h

ACCEPTS: (left, bottom, width, height)

set_boxstyle(boxstyle=None, **kw)

Set the box style.

boxstyle can be a string with boxstyle name with optional comma-separated attributes. Alternatively, the attrs can be provided as keywords:

set_boxstyle("round,pad=0.2")
set_boxstyle("round", pad=0.2)

Old attrs simply are forgotten.

Without argument (or with boxstyle = None), it returns available box styles.

set_height(h)

Set the width rectangle

ACCEPTS: float

set_mutation_aspect(aspect)

Set the aspect ratio of the bbox mutation.

ACCEPTS: float

set_mutation_scale(scale)

Set the mutation scale.

ACCEPTS: float

set_width(w)

Set the width rectangle

ACCEPTS: float

set_x(x)

Set the left coord of the rectangle

ACCEPTS: float

set_y(y)

Set the bottom coord of the rectangle

ACCEPTS: float
class matplotlib.patches.Patch(edgecolor=None, facecolor=None, color=None, linewidth=None, linestyle=None, antialiased=None, hatch=None, fill=True, **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:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased</td>
<td>or aa [True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>a matplotlib.color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>facecolor</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[/</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle</td>
<td>[‘solid’</td>
</tr>
<tr>
<td>linewidth</td>
<td>float or None for default</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None]float</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

contains(mouseevent, radius=None)

Test whether the mouse event occurred in the patch.

Returns T/F, {}

does_contain_point(point, radius=None)

Returns True if the given point is inside the path (transformed with its transform attribute).
draw(artist, renderer, *args, **kwargs)
    Draw the Patch to the given renderer.

fill
    return whether fill is set

get_aa()
    Returns True if the Patch is to be drawn with antialiasing.

get_antialiased()
    Returns True if the Patch is to be drawn with antialiasing.

get_data_transform()
    Return the Transform instance which maps data coordinates to physical coordinates.

get_ec()
    Return the edge color of the Patch.

get_edgecolor()
    Return the edge color of the Patch.

get_extents()
    Return a Bbox object defining the axis-aligned extents of the Patch.

get_facecolor()
    Return the face color of the Patch.

get_fc()
    Return the face color of the Patch.

get_fill()
    return whether fill is set

get_hatch()
    Return the current hatching pattern

get_linestyle()
    Return the linestyle. Will be one of ['solid' | 'dashed' | 'dashdot' | 'dotted']

get_linewidth()
    Return the line width in points.

get_ls()
    Return the linestyle. Will be one of ['solid' | 'dashed' | 'dashdot' | 'dotted']

get_lw()
    Return the line width in points.

get_patch_transform()
    Return the Transform instance which takes 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()
    Return the path of this patch
get_transform()
    Return the Transform applied to the Patch.

get_verts()
    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(renderer=None)

set_aa(aa)
    alias for set_antialiased

set_alpha(alpha)
    Set the alpha transparency of the patch.
    ACCEPTS: float or None

set_antialiased(aa)
    Set whether to use antialiased rendering
    ACCEPTS: [True | False] or None for default

set_color(c)
    Set both the edgecolor and the facecolor.
    ACCEPTS: matplotlib color spec

    See also:

    set_facecolor(), set_edgecolor() For setting the edge or face color individually.

set_ec(color)
    alias for set_edgecolor

set_edgecolor(color)
    Set the patch edge color
    ACCEPTS: mpl color spec, or None for default, or ‘none’ for no color

set_facecolor(color)
    Set the patch face color
    ACCEPTS: mpl color spec, or None for default, or ‘none’ for no color

set_fc(color)
    alias for set_facecolor

set_fill(b)
    Set whether to fill the patch
    ACCEPTS: [True | False]

set_hatch(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.

ACCEPTS: ['/', '\', '|', '-', '+', 'x', 'o', 'O', '.']

**set_linestyle**(ls)
Set the patch linestyle

ACCEPTS: ['solid' | 'dashed' | 'dashdot' | 'dotted']

**set_linewidth**(w)
Set the patch linewidth in points

ACCEPTS: float or None for default

**set_ls**(ls)
alias for set_linestyle

**set_lw**(lw)
alias for set_linewidth

**update_from**(other)
Updates this Patch from the properties of other.

**zorder** = 1

**class** matplotlib.patches.PathPatch**(path, **kwargs)**

Bases: matplotlib.patches.Patch

A general polycurve path patch.

$path$ is a matplotlib.path.Path object.

Valid kwargs are:
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or ec</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>facecolor or fc</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>['/'</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['solid'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float or None for default</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

See also:

**Patch** For additional kwargs

```python
get_path()
```

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 kwargs are:
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or ec</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>facecolor or fc</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[’/’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>[’solid’</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float or None for default</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

**See also:**

**Patch** For additional kwargs

- **get_closed()**
- **get_path()**
- **get_xy()**
- **set_closed(closed)**
- **set_xy(xy)**
xy
Set/get the vertices of the polygon. This property is provided for backward compatibility with matplotlib 0.91.x only. New code should use get_xy() and set_xy() instead.

class matplotlib.patches.Rectangle(xy, width, height, angle=0.0, **kwargs)
Bases: matplotlib.patches.Patch

Draw a rectangle with lower left at $xy = (x, y)$ with specified $width$ and $height$.

angle rotation in degrees (anti-clockwise)

fill is a boolean indicating whether to fill the rectangle

Valid kwargs are:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or ec</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>facecolor or fc</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>['/'</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with '%s' conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['solid'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float or None for default</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

contains(mouseevent)

get_bbox()
get_height()  
Return the height of the rectangle

get_patch_transform()

get_path()  
Return the vertices of the rectangle

get_width()  
Return the width of the rectangle

get_x()  
Return the left coord of the rectangle

get_xy()  
Return the left and bottom coords of the rectangle

get_y()  
Return the bottom coord of the rectangle

set_bounds(*args)  
Set the bounds of the rectangle: l,b,w,h  
ACCEPTS: (left, bottom, width, height)

set_height(h)  
Set the width rectangle  
ACCEPTS: float

set_width(w)  
Set the width rectangle  
ACCEPTS: float

set_x(x)  
Set the left coord of the rectangle  
ACCEPTS: float

set_xy(xy)  
Set the left and bottom coords of the rectangle  
ACCEPTS: 2-item sequence

set_y(y)  
Set the bottom coord of the rectangle  
ACCEPTS: float

xy  
Return the left and bottom coords of the rectangle

class matplotlib.patches.RegularPolygon(xy, numVertices, radius=5, orientation=0, **kwargs)

Bases: matplotlib.patches.Patch

52.3. matplotlib.patches
A regular polygon patch.

Constructor arguments:

* `xy` A length 2 tuple \((x, y)\) of the center.

* `numVertices` the number of vertices.

* `radius` The distance from the center to each of the vertices.

* `orientation` rotates the polygon (in radians).

Valid kwargs are:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>agg_filter</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>alpha</code></td>
<td>float or None</td>
</tr>
<tr>
<td><code>animated</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>antialiased</code> or <code>aa</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>axes</code></td>
<td>an <code>Axes</code> instance</td>
</tr>
<tr>
<td><code>clip_box</code></td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td><code>clip_on</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>clip_path</code></td>
<td>([Path, Transform]</td>
</tr>
<tr>
<td><code>color</code></td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td><code>contains</code></td>
<td>a callable function</td>
</tr>
<tr>
<td><code>edgecolor</code> or <code>ec</code></td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td><code>facecolor</code> or <code>fc</code></td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td><code>figure</code></td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td><code>fill</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>gid</code></td>
<td>an id string</td>
</tr>
<tr>
<td><code>hatch</code></td>
<td>[’\’</td>
</tr>
<tr>
<td><code>label</code></td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td><code>linestyle</code> or <code>ls</code></td>
<td>[’solid’</td>
</tr>
<tr>
<td><code>linewidth</code> or <code>lw</code></td>
<td>float or None for default</td>
</tr>
<tr>
<td><code>lod</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>path_effects</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>picker</code></td>
<td>[None</td>
</tr>
<tr>
<td><code>rasterized</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>sketch_params</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>snap</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>transform</code></td>
<td><code>Transform</code> instance</td>
</tr>
<tr>
<td><code>url</code></td>
<td>a url string</td>
</tr>
<tr>
<td><code>visible</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>zorder</code></td>
<td>any number</td>
</tr>
</tbody>
</table>

`get_patch_transform()`

`get_path()`
numvertices

orientation

radius

xy

class matplotlib.patches.Shadow(patch, ox, oy, props=None, **kwargs)

Bases: matplotlib.patches.Patch

Create a shadow of the given patch offset by ox, oy. props, if not None, is a patch property update dictionary. If None, the shadow will have have the same color as the face, but darkened.

kwargs are

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>([Path, Transform]</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>facecolor</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[’/’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle</td>
<td>[’solid’</td>
</tr>
<tr>
<td>linewidth</td>
<td>float or None for default</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>
**draw**(renderer)

**get_patch_transform()**

**get_path()**

class `matplotlib.patches.Wedge`(center, r, theta1, theta2, width=None, **kwargs)

Bases: `matplotlib.patches.Patch`

Wedge shaped patch.

Draw 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 kwargs are:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or ec</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>facecolor or fc</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[‘’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>[‘solid’</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float or None for default</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td><code>Transform</code> instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>
Matplotlib, Release 1.3.0

get_path()

set_center(center)

set_radius(radius)

set_theta1(theta1)

set_theta2(theta2)

set_width(width)

class matplotlib.patches.YAArrow(figure, xytip, xybase, width=4, frac=0.1, headwidth=12, **kwargs)

Bases: matplotlib.patches.Patch

Yet another arrow class.

This is an arrow that is defined in display space and has a tip at x1, y1 and a base at x2, y2.

Constructor arguments:

xytip (x, y) location of arrow tip

xybase (x, y) location the arrow base mid point

figure The Figure instance (fig.dpi)

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

Valid kwargs are:
### Property Description

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>([Path, Transform]</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or ec</td>
<td>mpl color spec, or None for default, or ‘none‘ for no color</td>
</tr>
<tr>
<td>facecolor or fc</td>
<td>mpl color spec, or None for default, or ‘none‘ for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
</tbody>
</table>
| hatch             | ['| | \| | |\-
+-\'-\"|\*\'] |
| label             | string or anything printable with ‘%s’ conversion. |
| linestyle or ls   | [‘solid’ | ‘dashed’ | ‘dashdot’ | ‘dotted’] |
| linewidth or lw   | float or None for default |
| lod               | [True | False]          |
| path_effects      | unknown              |
| picker            | [None|float|boolean|callable] |
| rasterized        | [True | False | None] |
| sketch_params     | unknown              |
| snap              | unknown              |
| transform         | Transform instance   |
| url               | a url string         |
| visible           | [True | False]          |
| zorder            | any number           |

**get_patch_transform()**

**get_path()**

**getpoints(x1, y1, x2, y2, k)**

For line segment defined by \((x1, y1)\) and \((x2, y2)\) return the points on the line that is perpendicular to the line and intersects \((x2, y2)\) and the distance from \((x2, y2)\) of the returned points is \(k\).

**matplotlib.patches.bbox_artist(artist, renderer, props=None, fill=True)**

This is a debug function to draw a rectangle around the bounding box returned by get_window_extent() of an artist, 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.

**matplotlib.patches.draw_bbox(bbox, renderer, color='k', trans=None)**

This is a debug function to draw a rectangle around the bounding box returned by
get_window_extent() of an artist, to test whether the artist is returning the correct bbox.

52.4 matplotlib.text

Classes for including text in a figure.

class matplotlib.text.Annotation(s, xy, xytext=None, xycoords='data', textcoords=None, arrowprops=None, annotation_clip=None, **kwargs)

Bases: matplotlib.text.Text, matplotlib.text._AnnotationBase

A Text class to make annotating things in the figure, such as Figure, Axes, Rectangle, etc., easier.

Annotate the x, y point xy with text s at x, y location xytext. (If xytext = None, defaults to xy, and if textcoords = None, defaults to xycoords).

arrowprops, if not None, is a dictionary of line properties (see matplotlib.lines.Line2D) for the arrow that connects annotation to the point.

If the dictionary has a key arrowstyle, a FancyArrowPatch instance is created with the given dictionary and is drawn. Otherwise, a YAArow patch instance is created and drawn. Valid keys for YAArow are

<table>
<thead>
<tr>
<th>Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>width</td>
<td>the width of the arrow in points</td>
</tr>
<tr>
<td>frac</td>
<td>the fraction of the arrow length occupied by the head</td>
</tr>
<tr>
<td>head-width</td>
<td>the width of the base of the arrow head in points</td>
</tr>
<tr>
<td>shrink</td>
<td>oftentimes it is convenient to have the arrow tip and base a bit away from the text and point being annotated. If d is the distance between the text and annotated point, shrink will shorten the arrow so the tip and base are shrink percent of the distance d away from the endpoints. ie, shrink=0.05 is 5%</td>
</tr>
</tbody>
</table>

? any key for matplotlib.patches.polygon

Valid keys for FancyArrowPatch are

<table>
<thead>
<tr>
<th>Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>arrowstyle</td>
<td>the arrow style</td>
</tr>
<tr>
<td>connectionstyle</td>
<td>the connection style</td>
</tr>
<tr>
<td>relpos</td>
<td>default is (0.5, 0.5)</td>
</tr>
<tr>
<td>patchA</td>
<td>default is bounding box of the text</td>
</tr>
<tr>
<td>patchB</td>
<td>default is None</td>
</tr>
<tr>
<td>shrinkA</td>
<td>default is 2 points</td>
</tr>
<tr>
<td>shrinkB</td>
<td>default is 2 points</td>
</tr>
<tr>
<td>mutation_scale</td>
<td>default is text size (in points)</td>
</tr>
<tr>
<td>mutation_aspect</td>
<td>default is 1.</td>
</tr>
</tbody>
</table>

? any key for matplotlib.patches.PathPatch

xycoords and textcoords are strings that indicate the coordinates of xy and xytext.
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘figure points’</td>
<td>points from the lower left corner of the figure</td>
</tr>
<tr>
<td>‘figure pixels’</td>
<td>pixels from the lower left corner of the figure</td>
</tr>
<tr>
<td>‘figure fraction’</td>
<td>0,0 is lower left of figure and 1,1 is upper right</td>
</tr>
<tr>
<td>‘axes points’</td>
<td>points from lower left corner of axes</td>
</tr>
<tr>
<td>‘axes pixels’</td>
<td>pixels from lower left corner of axes</td>
</tr>
<tr>
<td>‘axes fraction’</td>
<td>0,0 is lower left of axes and 1,1 is upper right</td>
</tr>
<tr>
<td>‘data’</td>
<td>use the coordinate system of the object being annotated (default)</td>
</tr>
<tr>
<td>‘offset points’</td>
<td>Specify an offset (in points) from the xy value</td>
</tr>
<tr>
<td>‘polar’</td>
<td>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.</td>
</tr>
</tbody>
</table>

If a ‘points’ or ‘pixels’ option is specified, values will be added to the bottom-left and if negative, values will be subtracted from the top-right. e.g.:

```python
# 10 points to the right of the left border of the axes and
# 5 points below the top border
xy=(10,-5), xycoords='axes points'
```

You may use an instance of Transform or Artist. See Annotating Axes for more details.

The annotation_clip attribute controls the visibility of the annotation when it goes outside the axes area. If True, the annotation will only be drawn when the xy is inside the axes. If False, the annotation will always be drawn regardless of its position. The default is None, which behave as True only if xycoords is “data”.

Additional kwargs are Text properties:
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>family or fontfamily or fontname or name</td>
<td>['serif'</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fontproperties or font_properties</td>
<td>a matplotlib.font_manager.FontProperties instance</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>horizontalalignment or ha</td>
<td>['center'</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with 's' conversion.</td>
</tr>
<tr>
<td>linespacing</td>
<td>float (multiple of font size)</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>multialignment</td>
<td>['left'</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>position</td>
<td>(x,y)</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>rotation</td>
<td>[ angle in degrees</td>
</tr>
<tr>
<td>rotation_mode</td>
<td>unknown</td>
</tr>
<tr>
<td>size or fontsize</td>
<td>[size in points</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>stretch or fontstretch</td>
<td>[a numeric value in range 0-1000</td>
</tr>
<tr>
<td>style or fontstyle</td>
<td>['normal'</td>
</tr>
<tr>
<td>text</td>
<td>string or anything printable with 's' conversion.</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>variant or fontvariant</td>
<td>['normal'</td>
</tr>
<tr>
<td>verticalalignment or va or ma</td>
<td>['center'</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>weight or fontweight</td>
<td>[a numeric value in range 0-1000</td>
</tr>
<tr>
<td>x</td>
<td>float</td>
</tr>
<tr>
<td>y</td>
<td>float</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

contains(event)

draw(artist, renderer, *args, **kwargs)

    Draw the Annotation object to the given renderer.

set_figure(fig)

update_bbox_position_size(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_positions(renderer)`

“Update the pixel positions of the annotated point and the text.

class matplotlib.text.OffsetFrom(artist, ref_coord, unit='points')
Bases: object

g_get_unit()

set_unit(unit)

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, **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 kwargs are

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>backgroundcolor</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>bbox</td>
<td>rectangle prop dict</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>family or fontfamily or fontname or name</td>
<td>[FONTNAME</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fontproperties or font_properties</td>
<td>a matplotlib.font_manager.FontProperties instance</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>horizontalalignment or ha</td>
<td>[ ‘center’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linespacing</td>
<td>float (multiple of font size)</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>multialignment</td>
<td>[’left’</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None]float]boolean]callable]</td>
</tr>
<tr>
<td>position</td>
<td>(x,y)</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>rotation</td>
<td>[ angle in degrees</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>rotation_mode</td>
<td>unknown</td>
</tr>
<tr>
<td>size or fontsize</td>
<td>[size in points</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>stretch or fontstretch</td>
<td>[a numeric value in range 0-1000</td>
</tr>
<tr>
<td>style or fontstyle</td>
<td>[ 'normal', 'italic', 'oblique']</td>
</tr>
<tr>
<td>text</td>
<td>string or anything printable with '%s' conversion.</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>variant or fontvariant</td>
<td>[ 'normal'</td>
</tr>
<tr>
<td>verticalalignment or va or ma</td>
<td>[ 'center', 'top', 'bottom', 'baseline' ]</td>
</tr>
<tr>
<td>visible</td>
<td>[True, False]</td>
</tr>
<tr>
<td>weight or fontweight</td>
<td>[a numeric value in range 0-1000</td>
</tr>
<tr>
<td>x</td>
<td>float</td>
</tr>
<tr>
<td>y</td>
<td>float</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

```python
cached = [(288.0, 134.39999999999998, 'simple', 'k', 'center', 'right', 7012663139591932583, None, None, 72, 10, 2.0)]
```
get_fontproperties()
    Return the FontProperties object

get_fontsize()
    alias for get_size

get_fontstretch()
    alias for get_stretch

get_fontstyle()
    alias for get_style

get_fontvariant()
    alias for get_variant

get_fontweight()
    alias for get_weight

get_ha()
    alias for get_horizontalalignment

get_horizontalalignment()
    Return the horizontal alignment as string. Will be one of ‘left’, ‘center’ or ‘right’.

get_name()
    Return the font name as string

get_position()
    Return the position of the text as a tuple (x, y)

get_prop_tup()
    Return a hashable tuple of properties.
    Not intended to be human readable, but useful for backends who want to cache derived information about text (eg layouts) and need to know if the text has changed.

get_rotation()
    return the text angle as float in degrees

get_rotation_mode()
    get text rotation mode

get_size()
    Return the font size as integer

get_stretch()
    Get the font stretch as a string or number

get_style()
    Return the font style as string

get_text()
    Get the text as string

get_va()
    alias for getverticalalignment()
get_variant()

Return the font variant as a string

general Verticalalignment()

Return the vertical alignment as string. Will be one of ‘top’, ‘center’, ‘bottom’ or ‘baseline’.

general Weight()

Get the font weight as string or number

general Window_extent(renderer=None, dpi=None)

Return a Bbox object 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.

renderer defaults to the _renderer attribute of the text object. This is not assigned until the first execution of draw(), so you must use this kwarg if you want to call get_window_extent() prior to the first draw(). For getting web page regions, it is simpler to call the method after saving the figure.

dpi defaults to self.figure.dpi; the renderer dpi is irrelevant. For the web application, if figure.dpi is not the value used when saving the figure, then the value that was used must be specified as the dpi argument.

static is_math_text(s)

Returns a cleaned string and a boolean flag. The flag indicates if the given string s contains any mathtext, determined by counting unescaped dollar signs. If no mathtext is present, the cleaned string has its dollar signs unescaped. If usetex is on, the flag always has the value “TeX”.

set_backgroundcolor(color)

Set the background color of the text by updating the bbox.

See also:

set_bbox() To change the position of the bounding box.

ACCEPTS: any matplotlib color

set_bbox(rectprops)

Draw a bounding box around self. rectprops are any settable properties for a rectangle, eg face-color=’red’, alpha=0.5.

t.set_bbox(dict(facecolor=’red’, alpha=0.5))

If rectprops has “boxstyle” key. A FancyBboxPatch is initialized with rectprops and will be drawn. The mutation scale of the FancyBboxPath is set to the fontsize.

ACCEPTS: rectangle prop dict

set_color(color)

Set the foreground color of the text

ACCEPTS: any matplotlib color

set_family(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 matplotlibrc file.
ACCEPES: [FONTNAME | ‘serif’ | ‘sans-serif’ | ‘cursive’ | ‘fantasy’ | ‘monospace’]

set_font_properties(fp)
    alias for set_fontproperties

set_fontname(fontname)
    alias for set_family

set_fontproperties(fp)
    Set the font properties that control the text. fp must be a matplotlib.font_manager.FontProperties object.
    ACCEPTS: a matplotlib.font_manager.FontProperties instance

set_fontsize(fontsize)
    alias for set_size

set_fontstretch(stretch)
    alias for set_stretch

set_fontstyle(fontstyle)
    alias for set_style

set_fontvariant(variant)
    alias for set_variant

set_fontweight(weight)
    alias for set_weight

set_ha(align)
    alias for set_horizontalalignment

set_horizontalalignment(align)
    Set the horizontal alignment to one of
    ACCEPTS: ['center' | 'right' | 'left']

set_linespacing(spacing)
    Set the line spacing as a multiple of the font size. Default is 1.2.
    ACCEPTS: float (multiple of font size)

set_ma(align)
    alias for set_verticalalignment

set_multialignment(align)
    Set the alignment for multiple lines layout. The layout of the bounding box of all the lines is
determined by the horizontalalignment and verticalalignment properties, but the multiline text
within that box can be
    ACCEPTS: ['left' | 'right' | 'center']

set_name(fontname)
    alias for set_family

set_position(xy)
    Set the (x, y) position of the text
`set_rotation(s)`  
Set the rotation of the text  

**ACCEPTS:** [ angle in degrees | ‘vertical’ | ‘horizontal’ ]

`set_rotation_mode(m)`  
Set text rotation mode. If “anchor”, the un-rotated text will first aligned according to their `ha` and `va`, and then will be rotated with the alignment reference point as a origin. If None (default), the text will be rotated first then will be aligned.

`set_size(fontsize)`  
Set the font size. May be either a size string, relative to the default font size, or an absolute font size in points.  

**ACCEPTS:** [size in points | ‘xx-small’ | ‘x-small’ | ‘small’ | ‘medium’ | ‘large’ | ‘x-large’ | ‘xx-large’ ]

`set_stretch(stretch)`  
Set the font stretch (horizontal condensation or expansion).  

**ACCEPTS:** [a numeric value in range 0-1000 | ‘ultra-condensed’ | ‘extra-condensed’ | ‘condensed’ | ‘semi-condensed’ | ‘normal’ | ‘semi-expanded’ | ‘expanded’ | ‘extra-expanded’ | ‘ultra-expanded’ ]

`set_style(fontstyle)`  
Set the font style.  

**ACCEPTS:** [ ‘normal’ | ‘italic’ | ‘oblique’ ]

`set_text(s)`  
Set the text string `s`  

It may contain newlines (`\n`) or math in LaTeX syntax.  

**ACCEPTS:** string or anything printable with ‘%s’ conversion.

`set_va(align)`  
alias for `set_verticalalignment`

`set_variant(variant)`  
Set the font variant, either ‘normal’ or ‘small-caps’.  

**ACCEPTS:** [ ‘normal’ | ‘small-caps’ ]

`set_verticalalignment(align)`  
Set the vertical alignment  

**ACCEPTS:** [ ‘center’ | ‘top’ | ‘bottom’ | ‘baseline’ ]

`set_weight(weight)`  
Set the font weight.  

set_x(x)
    Set the x position of the text
    ACCEPTS: float

set_y(y)
    Set the y position of the text
    ACCEPTS: float

update_bbox_position_size(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(other)
    Copy properties from other to self

zorder = 3

class matplotlib.text.TextWithDash(x=0, y=0, text='', color=None, verticalalignment='center', horizontalalignment='center', multialignment=None, fontproperties=None, rotation=None, linespacing=None, dashlength=0.0, dashdirection=0, dashrotation=None, dashpad=3, dashpush=0)

Bases: matplotlib.text.Text

This is basically a Text with a dash (drawn with a Line2D) before/after it. It is intended to be a drop-in
replacement for Text, and should behave identically to it when dashlength = 0.0.

The dash always comes between the point specified by set_position() and the text. When a dash
exists, the text alignment arguments (horizontalalignment, verticalalignment) are ignored.

dashlength is the length of the dash in canvas units. (default = 0.0).

dashdirection is one of 0 or 1, where 0 draws the dash after the text and 1 before. (default = 0).

dashrotation specifies the rotation of the dash, and should generally stay None. In this case
get_dashrotation() returns get_rotation(). (I.e., the dash takes its rotation from the text’s
rotation). Because the text center is projected onto the dash, major deviations in the rotation cause
what may be considered visually unappealing results. (default = None)

dashpad is a padding length to add (or subtract) space between the text and the dash, in canvas units.
(default = 3)

dashpush “pushes” the dash and text away from the point specified by set_position() by the amount
in canvas units. (default = 0)

Note: The alignment of the two objects is based on the bounding box of the Text, as obtained by
get_window_extent(). This, in turn, appears to depend on the font metrics as given by the rendering
backend. Hence the quality of the “centering” of the label text with respect to the dash varies depending
on the backend used.

Note: I’m not sure that I got the get_window_extent() right, or whether that’s sufficient for pro-
viding the object bounding box.

draw(renderer)

Draw the TextWithDash object to the given renderer.

get_dashdirection()

Get the direction dash. 1 is before the text and 0 is after.

get_dashlength()

Get the length of the dash.

get_dashpad()

Get the extra spacing between the dash and the text, in canvas units.

get_dashpush()

Get the extra spacing between the dash and the specified text position, in canvas units.

get_dashrotation()

Get the rotation of the dash in degrees.

get_figure()

return the figure instance the artist belongs to

get_position()

Return the position of the text as a tuple (x, y)

get_prop_tup()

Return a hashable tuple of properties.

Not intended to be human readable, but useful for backends who want to cache derived information about text (eg layouts) and need to know if the text has changed.

get_window_extent(renderer=None)

Return a Bbox object 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.

renderer defaults to the _renderer attribute of the text object. This is not assigned until the first execution of draw(), so you must use this kwarg if you want to call get_window_extent() prior to the first draw(). For getting web page regions, it is simpler to call the method after saving the figure.

set_dashdirection(dd)

Set the direction of the dash following the text. 1 is before the text and 0 is after. The default is 0, which is what you’d want for the typical case of ticks below and on the left of the figure.

ACCEPTS: int (1 is before, 0 is after)

set_dashlength(dl)

Set the length of the dash.

ACCEPTS: float (canvas units)
**set_dashpad(dp)**
Set the “pad” of the TextWithDash, which is the extra spacing between the dash and the text, in canvas units.

ACCEPTS: float (canvas units)

**set_dashpush(dp)**
Set the “push” of the TextWithDash, which is the extra spacing between the beginning of the dash and the specified position.

ACCEPTS: float (canvas units)

**set_dashrotation(dr)**
Set the rotation of the dash, in degrees

ACCEPTS: float (degrees)

**set_figure(fig)**
Set the figure instance the artist belong to.

ACCEPTS: a `matplotlib.figure.Figure` instance

**set_position(xy)**
Set the (x, y) position of the `TextWithDash`.

ACCEPTS: (x, y)

**set_transform(t)**
Set the `matplotlib.transforms.Transform` instance used by this artist.

ACCEPTS: a `matplotlib.transforms.Transform` instance

**set_x(x)**
Set the x position of the `TextWithDash`.

ACCEPTS: float

**set_y(y)**
Set the y position of the `TextWithDash`.

ACCEPTS: float

**update_coords(renderer)**
Computes the actual x, y coordinates for text based on the input x, y and the *dashlength*. Since the rotation is with respect to the actual canvas’s coordinates we need to map back and forth.

**matplotlib.text.get_rotation(rotation)**
Return the text angle as float.

*rotation* may be ‘horizontal’, ‘vertical’, or a numeric value in degrees.
53.1 matplotlib.axes

class matplotlib.axes.Axes(fig, rect, axisbg=None, frameon=True, sharex=None, sharey=None, label='', xscale=None, yscale=None, **kwargs)

Bases: matplotlib.artist.Artist

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.

acorr(x, **kwargs)

Plot the autocorrelation of x.

Call signature:

acorr(x, normed=True, detrend=mlab.detrend_none, usevlines=True, maxlags=10, **kwargs)

If normed = True, normalize the data by the autocorrelation at 0-th lag. x is detrended by the detrend callable (default no normalization).

Data are plotted as plot(lags, c, **kwargs)

Return value is a tuple (lags, c, **kwargs)

• lags are a length 2*maxlags+1 lag vector
• c is the 2*maxlags+1 auto correlation vector
• line is a Line2D instance returned by plot()

The default linestyle is None and the default marker is 'o', though these can be overridden with keyword args. The cross correlation is performed with numpy.correlate() with mode = 2.

If usevlines is True, vlines() rather than plot() is used to draw vertical lines from the origin to the acorr. Otherwise, the plot style is determined by the kwargs, which are Line2D properties.

maxlags is a positive integer detailing the number of lags to show. The default value of None will return all (2*len(x)-1) lags.

The return value is a tuple (lags, c, linecol, b) where
• `linecol` is the `LineCollection`

See also:

`plot()` or `vlines()` For documentation on valid kwargs.

Example:

`xcorr()` is top graph, and `acorr()` is bottom graph.

```
add_artist(a)
Add any Artist to the axes.

Returns the artist.

add_collection(collection, autolim=True)
Add a Collection instance to the axes.

Returns the collection.

add_container(container)
Add a Container instance to the axes.

Returns the collection.

add_line(line)
Add a Line2D to the list of plot lines
Returns the line.

**add_patch**(*p*)

Add a Patch *p* to the list of axes patches; the clipbox will be set to the Axes clipping box. If the transform is not set, it will be set to *transData*.

Returns the patch.

**add_table**(*tab*)

Add a Table instance to the list of axes tables

Returns the table.

**annotate**(*args, **kwargs*)

Create an annotation: a piece of text referring to a data point.

Call signature:

```python
annotate(s, xy, xytext=None, xycoords='data',
    textcoords='data', arrowprops=None, **kwargs)
```

Keyword arguments:

Annotate the *x*, *y* point *xy* with text *s* at *x*, *y* location *xytext*. (If *xytext = None*, defaults to *xy*, and if *textcoords = None*, defaults to *xycoords*).

*arrowprops*, if not *None*, is a dictionary of line properties (see matplotlib.lines.Line2D) for the arrow that connects annotation to the point.

If the dictionary has a key *arrowstyle*, a FancyArrowPatch instance is created with the given dictionary and is drawn. Otherwise, a YAArow patch instance is created and drawn. Valid keys for YAArow are

<table>
<thead>
<tr>
<th>Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>width</td>
<td>the width of the arrow in points</td>
</tr>
<tr>
<td>frac</td>
<td>the fraction of the arrow length occupied by the head</td>
</tr>
<tr>
<td>head-width</td>
<td>the width of the base of the arrow head in points</td>
</tr>
<tr>
<td>shrink</td>
<td>oftentimes it is convenient to have the arrowtip and base a bit away from the text and point being annotated. If <em>d</em> is the distance between the text and annotated point, shrink will shorten the arrow so the tip and base are shrink percent of the distance <em>d</em> away from the endpoints. ie, shrink=0.05 is 5%</td>
</tr>
<tr>
<td>?</td>
<td>any key for matplotlib.patches.polygon</td>
</tr>
</tbody>
</table>

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`

**xycoords and textcoords** are strings that indicate the coordinates of xy and xtext.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'figure points'</td>
<td>points from the lower left corner of the figure</td>
</tr>
<tr>
<td>'figure pixels'</td>
<td>pixels from the lower left corner of the figure</td>
</tr>
<tr>
<td>'figure fraction'</td>
<td>0,0 is lower left of figure and 1,1 is upper right</td>
</tr>
<tr>
<td>'axes points'</td>
<td>points from lower left corner of axes</td>
</tr>
<tr>
<td>'axes pixels'</td>
<td>pixels from lower left corner of axes</td>
</tr>
<tr>
<td>'axes fraction'</td>
<td>0,0 is lower left of axes and 1,1 is upper right</td>
</tr>
<tr>
<td>'data'</td>
<td>use the coordinate system of the object being annotated (default)</td>
</tr>
<tr>
<td>'offset points'</td>
<td>Specify an offset (in points) from the xy value</td>
</tr>
<tr>
<td>'polar'</td>
<td>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.</td>
</tr>
</tbody>
</table>

If a ‘points’ or ‘pixels’ option is specified, values will be added to the bottom-left and if negative, values will be subtracted from the top-right. e.g.:

```
# 10 points to the right of the left border of the axes and
# 5 points below the top border
xy=(10,-5), xycoords='axes points'
```

You may use an instance of `Transform` or `Artist`. See `Annotating Axes` for more details.

The `annotation_clip` attribute controls the visibility of the annotation when it goes outside the axes area. If True, the annotation will only be drawn when the xy is inside the axes. If False, the annotation will always be drawn regardless of its position. The default is `None`, which behave as True only if `xycoords` is “data”.

---

668 Chapter 53. axes
Additional kwargs are Text properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>backgroundcolor</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>bbox</td>
<td>rectangle prop dict</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>family or fontfamily or fontname or name</td>
<td>[FONTNAME</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fontproperties or font_properties</td>
<td>a matplotlib.font_manager.FontProperties instance</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>horizontalalignment or ha</td>
<td>[ ‘center’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linespacing</td>
<td>float (multiple of font size)</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>multialignment</td>
<td>[ ‘left’</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>position</td>
<td>(x,y)</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>rotation</td>
<td>[ angle in degrees</td>
</tr>
<tr>
<td>rotation_mode</td>
<td>unknown</td>
</tr>
<tr>
<td>size or fontsize</td>
<td>[size in points</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>stretch or fontstretch</td>
<td>[a numeric value in range 0-1000</td>
</tr>
<tr>
<td>style or fontstyle</td>
<td>[‘normal’</td>
</tr>
<tr>
<td>text</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>variant or fontvariant</td>
<td>[‘normal’</td>
</tr>
<tr>
<td>verticalalignment or va or ma</td>
<td>[ ‘center’</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>x</td>
<td>float</td>
</tr>
<tr>
<td>y</td>
<td>float</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>
apply_aspect(position=None)

Use _aspect() and _adjustable() to modify the axes box or the view limits.
arrow(x, y, dx, dy, **kwargs)

Add an arrow to the axes.

Call signature:

arrow(x, y, dx, dy, **kwargs)

Draws arrow on specified axis from (x, y) to (x + dx, y + dy). Uses FancyArrow patch to construct the arrow.

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().

Optional kwargs control the arrow construction and properties:

**Constructor arguments**

- **width**: float (default: 0.001) width of full arrow tail
- **length_includes_head**: [True | False] (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**: [True | False] (default: False) if True, the head starts being drawn at coordinate 0 instead of ending at coordinate 0.

Other valid kwargs (inherited from Patch) are:
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased  or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or ec</td>
<td>mpl color spec, or None for default, or 'none' for no color</td>
</tr>
<tr>
<td>facecolor or fc</td>
<td>mpl color spec, or None for default, or 'none' for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[’/’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ’%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>[’solid’</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float or None for default</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

**Example:**
autoscale\(\text{enable=True, axis='both', tight=None}\)

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.

**enable**: [True | False | None] True (default) turns autoscaling on, False turns it off. None leaves the autoscaling state unchanged.

**axis**: ['x' | 'y' | 'both'] which axis to operate on; default is 'both'

**tight**: [True | False | None] If True, set view limits to data limits; if False, let the locator and margins expand the view limits; if None, use tight scaling if the only artist is an image, otherwise treat tight as False. The tight setting is retained for future autoscaling until it is explicitly changed.

Returns None.

autoscale_view\(\text{tight=None, scalex=True, scaley=True}\)

Autoscale the view limits using the data limits. You can selectively autoscale only a single axis, eg, the xaxis by setting scaley to False. The autoscaling preserves any axis direction reversal that has already been done.

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.

axhline\(y=0, xmin=0, xmax=1, **kwargs\)

Add a horizontal line across the axis.

Call signature:
`axhline(y=0, xmin=0, xmax=1, **kwargs)`

Draw a horizontal line at $y$ from $xmin$ to $xmax$. With the default values of $xmin = 0$ and $xmax = 1$, this line will always span the horizontal extent of the axes, regardless of the xlim settings, even if you change them, e.g., with the `set_xlim()` command. That is, the horizontal extent is in axes coords: 0=left, 0.5=middle, 1.0=right but the $y$ location is in data coordinates.

Return value is the Line2D instance. kwargs are the same as kwargs to plot, and can be used to control the line properties. e.g.,

- draw a thick red hline at $y = 0$ that spans the xrange:

  ```python
  >>> axhline(linewidth=4, color='r')
  ```

- draw a default hline at $y = 1$ that spans the xrange:

  ```python
  >>> axhline(y=1)
  ```

- draw a default hline at $y = .5$ that spans the the middle half of the xrange:

  ```python
  >>> axhline(y=.5, xmin=0.25, xmax=0.75)
  ```

Valid kwargs are Line2D properties, with the exception of ‘transform’:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>agg_filter</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>alpha</code></td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td><code>animated</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>antialiased</code> or <code>aa</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>axes</code></td>
<td>an Axes instance</td>
</tr>
<tr>
<td><code>clip_box</code></td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td><code>clip_on</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>clip_path</code></td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td><code>color</code> or <code>c</code></td>
<td>any matplotlib color</td>
</tr>
<tr>
<td><code>contains</code></td>
<td>a callable function</td>
</tr>
<tr>
<td><code>dash_capstyle</code></td>
<td>[‘butt’</td>
</tr>
<tr>
<td><code>dash_joinstyle</code></td>
<td>[‘miter’</td>
</tr>
<tr>
<td><code>dashes</code></td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td><code>data</code></td>
<td>2D array (rows are x, y) or two 1D arrays</td>
</tr>
<tr>
<td><code>drawstyle</code></td>
<td>[‘default’</td>
</tr>
<tr>
<td><code>figure</code></td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td><code>fillstyle</code></td>
<td>[‘full’</td>
</tr>
<tr>
<td><code>gid</code></td>
<td>an id string</td>
</tr>
<tr>
<td><code>label</code></td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td><code>linestyle</code> or <code>ls</code></td>
<td>[‘-‘</td>
</tr>
<tr>
<td><code>linewidth</code> or <code>lw</code></td>
<td>float value in points</td>
</tr>
<tr>
<td><code>lod</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>marker</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>markeredgecolor</code> or <code>mec</code></td>
<td>any matplotlib color</td>
</tr>
</tbody>
</table>
### Table 53.2 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>markeredgewidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

See also:

- **axhspan()** for example plot and source code

```python
axhspan(ymin, ymax, xmin=0, xmax=1, **kwargs)
```

Add a horizontal span (rectangle) across the axis.

Call signature:

```python
axhspan(ymin, ymax, xmin=0, xmax=1, **kwargs)
```

y coords are in data units and x coords are in axes (relative 0-1) units.

Draw a horizontal span (rectangle) from ymin to ymax. With the default values of xmin = 0 and xmax = 1, this always spans the xrange, regardless of the xlim settings, even if you change them, e.g., with the `set_xlim()` command. That is, the horizontal extent is in axes coords: 0=left, 0.5=middle, 1.0=right but the y location is in data coordinates.

Return value is a `matplotlib.patches.Polygon` instance.

Examples:
- draw a gray rectangle from y = 0.25-0.75 that spans the horizontal extent of the axes:

  ```python
  >>> axhspan(0.25, 0.75, facecolor='0.5', alpha=0.5)
  ```

  Valid kwargs are `Polygon` properties:
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased  or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or ec</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>facecolor or fc</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[’/’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>[‘solid’</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float or None for default</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

Example:
**axis(**v**, **kwargs)**

Convenience method for manipulating the x and y view limits and the aspect ratio of the plot. For details, see `axis()`.

`kwargs` are passed on to `set_xlim()` and `set_ylim()`

**axvline**(x=0, ymin=0, ymax=1, **kwargs)

Add a vertical line across the axes.

Call signature:

```python
axvline(x=0, ymin=0, ymax=1, **kwargs)
```

Draw a vertical line at \(x\) from \(ymin\) to \(ymax\). With the default values of \(ymin = 0\) and \(ymax = 1\), this line will always span the vertical extent of the axes, regardless of the ylim settings, even if you change them, e.g., with the `set_ylim()` command. That is, the vertical extent is in axes coords: 0=bottom, 0.5=middle, 1.0=top but the \(x\) location is in data coordinates.

Return value is the Line2D instance. kwargs are the same as kwargs to plot, and can be used to control the line properties. e.g.,

- draw a thick red vline at \(x = 0\) that spans the yrange:

  ```python
  >>> axvline(linewidth=4, color='r')
  ```

- draw a default vline at \(x = 1\) that spans the yrange:
```python
>>> axvline(x=1)

• draw a default vline at \(x = .5\) that spans the the middle half of the yrange:

```python
>>> axvline(x=.5, ymin=0.25, ymax=0.75)
```

Valid kwargs are `Line2D` properties, with the exception of ‘transform’:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an <code>Axes</code> instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color or c</td>
<td>any <code>matplotlib</code> color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>dash_joistyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>data</td>
<td>2D array (rows are x, y) or two 1D arrays</td>
</tr>
<tr>
<td>drawstyle</td>
<td>['default'</td>
</tr>
<tr>
<td>figure</td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>['full'</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['-']</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>marker</td>
<td>unknown</td>
</tr>
<tr>
<td>markeredgewidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any <code>matplotlib</code> color</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any <code>matplotlib</code> color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>solid_joistyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>transform</td>
<td>a <code>matplotlib.transforms.Transform</code> instance</td>
</tr>
</tbody>
</table>
Table 53.3 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

See also:

axhspan() for example plot and source code

axvspan(xmin, xmax, ymin=0, ymax=1, **kwargs)
Add a vertical span (rectangle) across the axes.

Call signature:

axvspan(xmin, xmax, ymin=0, ymax=1, **kwargs)

x coords are in data units and y coords are in axes (relative 0-1) units.

Draw a vertical span (rectangle) from xmin to xmax. With the default values of ymin = 0 and ymax = 1, this always spans the yrange, regardless of the ylim settings, even if you change them, e.g., with the set_ylim() command. That is, the vertical extent is in axes coords: 0=bottom, 0.5=middle, 1.0=top but the y location is in data coordinates.

Return value is the matplotlib.patches.Polygon instance.

Examples:

- draw a vertical green translucent rectangle from x=1.25 to 1.55 that spans the yrange of the axes:

  >>> axvspan(1.25, 1.55, facecolor='g', alpha=0.5)

Valid kwargs are Polygon properties:
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)]</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or ec</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>facecolor or fc</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>['/']</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['solid’</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float or None for default</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

See also:

- `axhspan()` for example plot and source code

- `bar()` *(left, height, width=0.8, bottom=None, **kwargs)*

  Make a bar plot.

  Make a bar plot with rectangles bounded by:

  ```
  left, left + width, bottom, bottom + height
  ```

  **Parameters**

  - `left`: sequence of scalars
    - the x coordinates of the left sides of the bars
  - `height`: sequence of scalars
    - the heights of the bars
  - `width`: scalar or array-like, optional, default: 0.8
    - the width(s) of the bars
  - `bottom`: scalar or array-like, optional, default: None
    - the y coordinate(s) of the bars
  - `color`: scalar or array-like, optional
the colors of the bars

**edgecolor**: scalar or array-like, optional

the colors of the bar edges

**linewidth**: scalar or array-like, optional, default: None

width of bar edge(s). If None, use default linewidth; If 0, don’t draw edges.

**xerr**: scalar or array-like, optional, default: None

if not None, will be used to generate errorbar(s) on the bar chart

**yerr**: scalar or array-like, optional, default: None

if not None, will be used to generate errorbar(s) on the bar chart

**ecolor**: scalar or array-like, optional, default: None

specifies the color of errorbar(s)

**capsize**: integer, optional, default: 3

determines the length in points of the error bar caps

**error_kw**: dictionary of kwags to be passed to errorbar method. *ecolor* and *capsize* may be specified here rather than as independent kwags.

**align**: ['edge' | 'center'], optional, default: 'edge'

If 'edge', aligns bars by their left edges (for vertical bars) and by their bottom edges (for horizontal bars). If 'center', interpret the left argument as the coordinates of the centers of the bars.

**orientation**: 'vertical' | 'horizontal', optional, default: 'vertical'

The orientation of the bars.

**log**: boolean, optional, default: False

If true, sets the axis to be log scale

**Returns**: class:`matplotlib.patches.Rectangle` instances.

**Notes**

The optional arguments *color*, *edgecolor*, *linewidth*, *xerr*, and *yerr* can be either scalars or sequences of length equal to the number of bars. This enables you to use bar as the basis for stacked bar charts, or candlestick plots. Detail: *xerr* and *yerr* are passed directly to *errorbar()*(), so they can also have shape 2xN for independent specification of lower and upper errors.

Other optional kwargs:
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or ec</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>facecolor or fc</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[’/’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>[‘solid’</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float or None for default</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

**Example:** A stacked bar chart.
barbs(*args, **kw)
Plot a 2-D field of barbs.

Call signatures:

barb(U, V, **kw)
barb(U, V, C, **kw)
barb(X, Y, U, V, **kw)
barb(X, Y, U, V, C, **kw)

Arguments:

X, Y: The x and y coordinates of the barb locations (default is head of barb; see pivot kwarg)
U, V: Give the x and y components of the barb shaft
C: An optional array used to map colors to the bars

All arguments may be 1-D or 2-D arrays or sequences. If X and Y are absent, they will be generated as a uniform grid. If U and V are 2-D arrays but X and Y are 1-D, and if len(X) and len(Y) match the column and row dimensions of U, then X and Y will be expanded with numpy.meshgrid().

U, V, C may be masked arrays, but masked X, Y are not supported at present.

Keyword arguments:

length: Length of the barb in points; the other parts of the barb are scaled against this.
Default is 9
pivot: [ ‘tip’ | ‘middle’ ] The part of the arrow that is at the grid point; the arrow rotates about this point, hence the name pivot. Default is ‘tip’

barbcolor: [ color | color sequence ] Specifies the color all parts of the barb except any flags. This parameter is analogous to the edgecolor parameter for polygons, which can be used instead. However this parameter will override facecolor.

flagcolor: [ color | color sequence ] Specifies 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: 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: A flag on whether the empty barbs (circles) that are drawn should be filled with the flag color. If they are not filled, they will be drawn such that no color is applied to the center. Default is False

rounding: A flag to indicate 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. Default is True

barb_increments: 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: Either a single boolean flag or an array of booleans. Single boolean indicates whether the lines and flags should point opposite to normal for all barbs. An array (which should be the same size as the other data arrays) indicates whether to flip for each individual barb. Normal behavior is for the bars and lines to point right (comes from wind barbs having these features point towards low pressure in the Northern Hemisphere.) Default is False

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.

linewdths and edgecolors can be used to customize the barb. Additional PolyCollection keyword arguments:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or antialiaseds</td>
<td>Boolean or sequence of booleans</td>
</tr>
<tr>
<td>array</td>
<td>unknown</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clim</td>
<td>a length 2 sequence of floats</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[Path, Transform]</td>
</tr>
<tr>
<td>cmap</td>
<td>a colormap or registered colormap name</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>colorbar</td>
<td>unknown</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or edgecolors</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>facecolor or facecolors</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[’/’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or linestyles or dashes</td>
<td>[‘solid’</td>
</tr>
<tr>
<td>linewidth or lw or linewidths</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>norm</td>
<td>unknown</td>
</tr>
<tr>
<td>offset_position</td>
<td>unknown</td>
</tr>
<tr>
<td>offsets</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>paths</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>pickradius</td>
<td>unknown</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>urls</td>
<td>unknown</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
</tbody>
</table>

Continued on next page
Table 53.4 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

Example:
**barh**(*bottom*, *width*, **height**=0.8, *left**=None, **kwargs)

Make a horizontal bar plot.

Call signature:

```
barh(bottom, width, height=0.8, left=None, **kwargs)
```

Make a horizontal bar plot with rectangles bounded by:

- *left*, *left + width*, *bottom*, *bottom + height* (left, right, bottom and top edges)
- *bottom*, *width*, *height*, and *left* can be either scalars or sequences

Return value is a list of matplotlib.patches.Rectangle instances.

Required arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>bottom</em></td>
<td>the vertical positions of the bottom edges of the bars</td>
</tr>
<tr>
<td><em>width</em></td>
<td>the lengths of the bars</td>
</tr>
</tbody>
</table>

Optional keyword arguments:
<table>
<thead>
<tr>
<th>Key-word</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>height</strong></td>
<td>the heights (thicknesses) of the bars</td>
</tr>
<tr>
<td><strong>left</strong></td>
<td>the x coordinates of the left edges of the bars</td>
</tr>
<tr>
<td><strong>color</strong></td>
<td>the colors of the bars</td>
</tr>
<tr>
<td><strong>edge-color</strong></td>
<td>the colors of the bar edges</td>
</tr>
<tr>
<td><strong>linewidth</strong></td>
<td>width of bar edges; None means use default linewidth; 0 means don’t draw edges.</td>
</tr>
<tr>
<td><strong>xerr</strong></td>
<td>if not None, will be used to generate errorbars on the bar chart</td>
</tr>
<tr>
<td><strong>yerr</strong></td>
<td>if not None, will be used to generate errorbars on the bar chart</td>
</tr>
<tr>
<td><strong>ecolor</strong></td>
<td>specifies the color of any errorbar</td>
</tr>
<tr>
<td><strong>capsize</strong></td>
<td>(default 3) determines the length in points of the error bar caps</td>
</tr>
<tr>
<td><strong>align</strong></td>
<td>‘edge’ (default)</td>
</tr>
<tr>
<td><strong>log</strong></td>
<td>[False][True] False (default) leaves the horizontal axis as-is; True sets it to log scale</td>
</tr>
</tbody>
</table>

Setting `align` = ‘edge’ aligns bars by their bottom edges in bottom, while `align` = ‘center’ interprets these values as the y coordinates of the bar centers.

The optional arguments `color`, `edgecolor`, `linewidth`, `xerr`, and `yerr` can be either scalars or sequences of length equal to the number of bars. This enables you to use `barh` as the basis for stacked bar charts, or candlestick plots.

other optional kwargs:
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[ (Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or ec</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>facecolor or fc</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[’/’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>[‘solid’</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float or None for default</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

```
boxplot(x, notch=False, sym='b+', vert=True, whis=1.5, positions=None, widths=None,
        patch_artist=False, bootstrap=None, usermedians=None, conf_intervals=None)
```

Make a box and whisker plot.

Call signature:

```
boxplot(x, notch=False, sym='+', vert=True, whis=1.5,
        positions=None, widths=None, patch_artist=False,
        bootstrap=None, usermedians=None, conf_intervals=None)
```

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.

Function Arguments:

- **x**: Array or a sequence of vectors.
- **notch** [[ False (default) | True ]] If False (default), produces a rectangular box plot. If True, will produce a notched box plot
sym [[ default ‘b+’ ]] The default symbol for flier points. Enter an empty string (‘’) if you
don’t want to show fliers.

vert [[ False | True (default) ]] If True (default), makes the boxes vertical. If False, makes
horizontal boxes.

whis [[ default 1.5 ]] Defines the length of the whiskers as a function of the inner quartile
range. They extend to the most extreme data point within (whis*(75%-25%)) data
range.

bootstrap [[ None (default) | integer ]] Specifies whether to bootstrap the confidence in-
tervals around the median for notched boxplots. If bootstrap=None, no bootstrapping
is performed, and notches are calculated using a Gaussian-based asymptotic approxi-
mation (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 it’s 95% confidence intervals. Values between 1000 and 10000 are recom-
mended.

usermedians [[ default None ]] An array or sequence whose first dimension (or length)
is compatible with x. This overrides the medians computed by matplotlib for each
element of usermedians that is not None. When an element of usermedians == None, the
median will be computed directly as normal.

conf_intervals [[ default None ]] Array or sequence whose first dimension (or length)
is compatible with x and whose second dimension is 2. When the current element of
conf_intervals is not None, the notch locations computed by matplotlib are overrid-
den (assuming notch is True). When an element of conf_intervals is None, boxplot
compute notches the method specified by the other kwargs (e.g., bootstrap).

positions [[ default 1,2,...,n ]] Sets the horizontal positions of the boxes. The ticks and
limits are automatically set to match the positions.

widths [[ default 0.5 ]] Either a scalar or a vector and sets the width of each box. The de-
fault is 0.5, or 0.15*(distance between extreme positions) if that is smaller.

patch_artist [[ False (default) | True ]] If False produces boxes with the Line2D artist If
True produces boxes with the Patch artist

Returns a dictionary mapping each component of the boxplot to a list of the
matplotlib.pyplot.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 inter-
vals if enabled.
• medians: horizontal lines at the median of each box.
• whiskers: the vertical lines extending to the most extreme, n-outlier data points.
• caps: the horizontal lines at the ends of the whiskers.
• fliers: points representing data that extend beyone the whiskers (outliers).

Example:
Chapter 53. axes
**broken_barh**(xranges, yrange, **kwargs)

Plot horizontal bars.

Call signature:

```
broken_barh(self, xranges, yrange, **kwargs)
```

A collection of horizontal bars spanning `yrange` with a sequence of `xranges`.

Required arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>xranges</code></td>
<td>sequence of <code>(xmin, xwidth)</code></td>
</tr>
<tr>
<td><code>yrange</code></td>
<td>sequence of <code>(ymin, ywidth)</code></td>
</tr>
</tbody>
</table>

kwargs are `matplotlib.collections.BrokenBarHCollection` properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>agg_filter</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>alpha</code></td>
<td>float or None</td>
</tr>
<tr>
<td><code>animated</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>antialiased</code> or <code>antialiaseds</code></td>
<td>Boolean or sequence of booleans</td>
</tr>
<tr>
<td><code>array</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>axes</code></td>
<td>an <code>Axes</code> instance</td>
</tr>
<tr>
<td><code>clim</code></td>
<td>a length 2 sequence of floats</td>
</tr>
</tbody>
</table>

Continued on next page
Table 53.5 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>cmap</td>
<td>a colormap or registered colormap name</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>colorbar</td>
<td>unknown</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>facecolor</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[‘/’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle</td>
<td>[‘solid’</td>
</tr>
<tr>
<td>linewidth</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>offset_position</td>
<td>unknown</td>
</tr>
<tr>
<td>offsets</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>paths</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None]float</td>
</tr>
<tr>
<td>pickradius</td>
<td>unknown</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>urls</td>
<td>unknown</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

these can either be a single argument, ie:

    facecolors = 'black'

or a sequence of arguments for the various bars, ie:

    facecolors = ('black', 'red', 'green')

Example:
can_pan()
   Return True if this axes supports any pan/zoom button functionality.

can_zoom()
   Return True if this axes supports the zoom box button functionality.

cla()
   Clear the current axes.

clabel(CS, *args, **kwargs)
   Label a contour plot.

   Call signature:

   clabel(cs, **kwargs)

   Adds labels to line contours in cs, where cs is a ContourSet object returned by contour.

   clabel(cs, v, **kwargs)

   only labels contours listed in v.

   Optional keyword arguments:

   fontsize: size in points or relative size eg ’smaller’, ’x-large’
   colors:

      • if None, the color of each label matches the color of the corresponding contour
• if one string color, e.g., \texttt{colors = 'r'} or \texttt{colors = 'red'}, all labels will be plotted in this color
• if a tuple of matplotlib color args (string, float, rgb, etc), different labels will be plotted in different colors in the order specified

\texttt{inline}: controls whether the underlying contour is removed or not. Default is \texttt{True}.

\texttt{inline\_spacing}: space in pixels to leave on each side of label when placing inline. Defaults to 5. This spacing will be exact for labels at locations where the contour is straight, less so for labels on curved contours.

\texttt{fmt}: a format string for the label. Default is \texttt{'%1.3f'} Alternatively, this can be a dictionary matching contour levels with arbitrary strings to use for each contour level (i.e., \texttt{fmt[level]=string}), or it can be any callable, such as a \texttt{Formatter} instance, that returns a string when called with a numeric contour level.

\texttt{manual}: if \texttt{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).

\texttt{manual} can be an iterable object of x,y tuples. Contour labels will be created as if mouse is clicked at each x,y positions.

\texttt{rightside\_up}: if \texttt{True} (default), label rotations will always be plus or minus 90 degrees from level.

\texttt{use\_clabeltext}: if \texttt{True} (default is \texttt{False}), \texttt{ClabelText} class (instead of \texttt{matplotlib.Text}) is used to create labels. \texttt{ClabelText} recalculates rotation angles of texts during the drawing time, therefore this can be used if aspect of the axes changes.
labels at selected locations
Single color - negative contours solid
Crazy lines
clear()
clear the axes

**cohere**

```python
cohere(x, y, NFFT=256, Fs=2, Fc=0, detrend=<function detrend_none at 0x24627d0>,
        window=<function window_hanning at 0x2462578>, noverlap=0, pad_to=0,
        sides='default', scale_by_freq=None, **kwargs)
```

Plot the coherence between $x$ and $y$.

Call signature:

```python
cohere(x, y, NFFT=256, Fs=2, Fc=0, detrend = mlab.detrend_none,
        window = mlab.window_hanning, noverlap=0, pad_to=None,
        sides='default', scale_by_freq=None, **kwargs)
```

Plot the coherence between $x$ and $y$. Coherence is the normalized cross spectral density:

$$C_{xy} = \frac{|P_{xy}|^2}{P_{xx}P_{yy}}$$

(53.1)

Keyword arguments:

**NFFT**: integer The number of data points used in each block for the FFT. Must be even; a power 2 is most efficient. The default value is 256. This should NOT be used to get zero padding, or the scaling of the result will be incorrect. Use *pad_to* for this instead.

**Fs**: scalar The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, freqs, in cycles per time unit. The default value is 2.
**detrend**: callable 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 it is a function. The `pylab` module defines `detrend_none()`, `detrend_mean()`, and `detrend_linear()`, but you can use a custom function as well.

**window**: callable or ndarray 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()`, etc. The default is `window_hanning()`. 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.

**pad_to**: integer 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 psd (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`.

**sides**: [‘default’ | ‘onesided’ | ‘twosided’] Specifies which sides of the PSD to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. ‘onesided’ forces the return of a one-sided PSD, while ‘twosided’ forces two-sided.

**scale_by_freq**: boolean Specifies 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**: integer The number of points of overlap between blocks. The default value is 0 (no overlap).

**Fc**: integer The center frequency of `x` (defaults to 0), which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsamped to baseband.

The return value is a tuple `(Cxy, f)`, where `f` are the frequencies of the coherence vector.

kwargs are applied to the lines.

References:

kwargs control the Line2D properties of the coherence plot:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
</tbody>
</table>
Table 53.6 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>data</td>
<td>2D array (rows are x, y) or two 1D arrays</td>
</tr>
<tr>
<td>drawstyle</td>
<td>['default'</td>
</tr>
<tr>
<td>figure</td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>['full'</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with '%s' conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['-.'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>marker</td>
<td>unknown</td>
</tr>
<tr>
<td>markeredgecolor or mec</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markeredgewidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function <code>fn(artist, event)</code></td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>transform</td>
<td>a <code>matplotlib.transforms.Transform</code> instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

Example:
contains(mouseevent)
Test whether the mouse event occurred in the axes.

Returns True / False, {}  

contains_point(point)
Returns True if the point (tuple of x,y) is inside the axes (the area defined by the its patch). A pixel coordinate is required.

contour(*args, **kwargs)
Plot contours.

contour() and contourf() draw contour lines and filled contours, respectively. Except as noted, function signatures and return values are the same for both versions.

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().

Call signatures:

contour(Z)
make a contour plot of an array Z. The level values are chosen automatically.

contour(X,Y,Z)
$X, Y$ specify the (x, y) coordinates of the surface

```python
contour(Z,N)
contour(X,Y,Z,N)
```

contour $N$ automatically-chosen levels.

```python
contour(Z,V)
contour(X,Y,Z,V)
```

draw contour lines at the values specified in sequence $V$

```python
contourf(..., V)
```

fill the $\text{len}(V)-1$ regions between the values in $V$

```python
contour(Z, **kwargs)
```

Use keyword args to control colors, linewidth, origin, cmap ... see below for more details.

$X$ and $Y$ must both be 2-D with the same shape as $Z$, or they must both be 1-D such that len($X$) is the number of columns in $Z$ and len($Y$) is the number of rows in $Z$.

$C = \text{contour(...)}$ returns a QuadContourSet object.

Optional keyword arguments:

- **colors**: [None | string | mpl_colors] If None, the colormap specified by cmap will be used.
  - If a string, like ‘r’ or ‘red’, all levels will be plotted in this color.
  - If a tuple of matplotlib color args (string, float, rgb, etc), different levels will be plotted in different colors in the order specified.
- **alpha**: float The alpha blending value
- **cmap**: [None | Colormap] A cm Colormap instance or None. If cmap is None and colors is None, a default Colormap is used.
- **norm**: [None | Normalize] A matplotlib.colors.Normalize instance for scaling data values to colors. If norm is None and colors is None, the default linear scaling is used.
- **vmin, vmax**: [None | scalar] If not None, either or both of these values will be supplied to the matplotlib.colors.Normalize instance, overriding the default color scaling based on levels.
- **levels**: [level0, level1, ..., leveln] A list of floating point numbers indicating the level curves to draw; eg to draw just the zero contour pass levels=[0]
- **origin**: [None | ‘upper’ | ‘lower’ | ‘image’] If None, the first value of $Z$ will correspond to the lower left corner, location (0,0). If ‘image’, the rc value for image.origin will be used.
  - This keyword is not active if $X$ and $Y$ are specified in the call to contour.
- **extent**: [None | (x0,x1,y0,y1)]
If `origin` is not `None`, then `extent` is interpreted as in `matplotlib.pyplot.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 keyword is not active if X and Y are specified in the call to `contour`.

**locator**: `[None | ticker.Locator subclass]` If `locator` is `None`, the default `MaxNLocator` is used. The locator is used to determine the contour levels if they are not given explicitly via the `V` argument.

**extend**: `[‘neither’ | ‘both’ | ‘min’ | ‘max’]` Unless this is ‘neither’, contour levels are automatically added to one or both ends of the range so that all data are included. These added ranges are then mapped to the special colormap values which default to the ends of the colormap range, but can be set via `matplotlib.colors.Colormap.set_under()` and `matplotlib.colors.Colormap.set_over()` methods.

**units, yunits**: `[None | registered units]` Override axis units by specifying an instance of a `matplotlib.units.ConversionInterface`.

**antialiased**: `[True | False]` enable antialiasing, overriding the defaults. For filled contours, the default is `True`. For line contours, it is taken from `rcParams[‘lines.antialiased’]`.

**contour-only keyword arguments**:

**linewidths**: `[None | number | tuple of numbers]` If `linewidths` is `None`, the default width in `lines.linewidth` in `matplotlibrc` is used.

If a number, all levels will be plotted with this linewidth.

If a tuple, different levels will be plotted with different linewidths in the order specified.

**linestyles**: `[None | ‘solid’ | ‘dashed’ | ‘dashdot’ | ‘dotted’]` If `linestyles` is `None`, the default is ‘solid’ unless the lines are monochrome. In that case, negative contours will take their linestyle from the `matplotlibrc` `contour.negative_linestyle` 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.

**contourf-only keyword arguments**:

**nchunk**: `[0 | integer]` If 0, no subdivision of the domain. Specify a positive integer to divide the domain into subdomains of roughly `nchunk` by `nchunk` points. This may never actually be advantageous, so this option may be removed. Chunking introduces artifacts at the chunk boundaries unless `antialiased` is `False`.

**hatches**: 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.

Note: `contourf` fills intervals that are closed at the top; that is, for boundaries z1 and z2, the filled region is:

\[ z1 < z \leq z2 \]

There is one exception: if the lowest boundary coincides with the minimum value of the z array, then that minimum value will be included in the lowest interval.
Examples:
labels at selected locations
Nonsense (3 masked regions)
Listed colors (3 masked regions)
**contourf**(*args, **kwargs)

Plot contours.

*contour()* and *contourf()* draw contour lines and filled contours, respectively. Except as noted, function signatures and return values are the same for both versions.

*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()*.

Call signatures:

contour(Z)

make a contour plot of an array Z. The level values are chosen automatically.

contour(X,Y,Z)

X, Y specify the (x, y) coordinates of the surface

contour(Z,N)

contour Z,N automatically-chosen levels.
contour(Z,V)
contour(X,Y,Z,V)

draw contour lines at the values specified in sequence V

contourf(..., V)

fill the len(V)−1 regions between the values in V

contour(Z, **kwargs)

Use keyword args to control colors, linewidth, origin, cmap ... see below for more details.

X and Y must both be 2-D with the same shape as Z, or they must both be 1-D such that len(X) is the number of columns in Z and len(Y) is the number of rows in Z.

C = contour(...) returns a QuadContourSet object.

Optional keyword arguments:

- **colors**: [ None | string | (mpl_colors) ] If None, the colormap specified by cmap will be used. If a string, like ‘r’ or ‘red’, all levels will be plotted in this color. If a tuple of matplotlib color args (string, float, rgb, etc), different levels will be plotted in different colors in the order specified.
- **alpha**: float The alpha blending value
- **cmap**: [ None | Colormap ] A cmap Colormap instance or None. If cmap is None and colors is None, a default Colormap is used.
- **norm**: [ None | Normalize ] A matplotlib.colors.Normalize instance for scaling data values to colors. If norm is None and colors is None, the default linear scaling is used.
- **vmin**, **vmax**: [ None | scalar ] If not None, either or both of these values will be supplied to the matplotlib.colors.Normalize instance, overriding the default color scaling based on levels.
- **levels**: [level0, level1, ..., leveln] A list of floating point numbers indicating the level curves to draw; eg to draw just the zero contour pass levels=[0]
- **origin**: [ None | ‘upper’ | ‘lower’ | ‘image’ ] If None, the first value of Z will correspond to the lower left corner, location (0,0). If ‘image’, the rc value for image.origin will be used.

This keyword is not active if X and Y are specified in the call to contour.

- **extent**: [ None | (x0,x1,y0,y1) ]

  If origin is not None, then extent is interpreted as in matplotlib.pyplot.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 keyword is not active if X and Y are specified in the call to contour.
locator: [ None | ticker.Locator subclass ] If locator is None, the default MaxNLocator is used. The locator is used to determine the contour levels if they are not given explicitly via the V argument.

extend: [ ‘neither’ | ‘both’ | ‘min’ | ‘max’ ] Unless this is ‘neither’, contour levels are automatically added to one or both ends of the range so that all data are included. These added ranges are then mapped to the special colormap values which default to the ends of the colormap range, but can be set via matplotlib.colors.Colormap.set_under() and matplotlib.colors.Colormap.set_over() methods.

xunits, yunits: [ None | registered units ] Override axis units by specifying an instance of matplotlib.units.ConversionInterface.

antialiased: [ True | False ] enable antialiasing, overriding the defaults. For filled contours, the default is True. For line contours, it is taken from rc-Params[‘lines.antialiased’].

contour-only keyword arguments:
linewdths: [ None | number | tuple of numbers ] If linewidths is None, the default width in lines.linewidth in matplotlibrc is used.
If a number, all levels will be plotted with this linewidth. If a tuple, different levels will be plotted with different linewidths in the order specified

linestyles: [ None | ‘solid’ | ‘dashed’ | ‘dashdot’ | ‘dotted’ ] If linestyles is None, the default is ‘solid’ unless the lines are monochrome. In that case, negative contours will take their linestyle from the matplotlibrc contour.negative_linestyle 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.

contourf-only keyword arguments:

ncbunk: [ 0 | integer ] If 0, no subdivision of the domain. Specify a positive integer to divide the domain into subdomains of roughly ncbunk by ncbunk points. This may never actually be advantageous, so this option may be removed. Chunking introduces artifacts at the chunk boundaries unless antialiased is False.

hatches: 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.

Note: contourf fills intervals that are closed at the top; that is, for boundaries z1 and z2, the filled region is:

\[ z1 < z \leq z2 \]

There is one exception: if the lowest boundary coincides with the minimum value of the z array, then that minimum value will be included in the lowest interval.

Examples:
Simplest default with labels
labels at selected locations
matplotlib.axes
Single color - negative contours solid
Nonsense (3 masked regions)

word length anomaly

sentence length anomaly

verbosity coefficient

Matplotlib, Release 1.3.0
Listed colors (3 masked regions)
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 $\text{detrend}$ and windowed by function $\text{window}$. 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.

Returns the tuple $(P_{xy}, \text{freqs})$. $P$ is the cross spectrum (complex valued), and $10 \log_{10} |P_{xy}|$ is plotted.

Keyword arguments:

- $NFFT$: integer The number of data points used in each block for the FFT. Must be even; a power 2 is most efficient. The default value is 256. This should NOT be used to get zero padding, or the scaling of the result will be incorrect. Use pad_to for this instead.
- $Fs$: scalar The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, freqs, in cycles per time unit. The default value is 2.
detrend: callable 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 it is a function. The pylab module defines detrend_none(), detrend_mean(), and detrend_linear(), but you can use a custom function as well.

window: callable or ndarray 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. The default is window_hanning(). 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.

pad_to: integer 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 psd (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.

sides: [‘default’ | ‘onesided’ | ‘twosided’] Specifies which sides of the PSD to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. ‘onesided’ forces the return of a one-sided PSD, while ‘twosided’ forces two-sided.

scale_by_freq: boolean Specifies 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: integer The number of points of overlap between blocks. The default value is 0 (no overlap).

Fc: integer The center frequency of x (defaults to 0), 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.


kwargs control the Line2D properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>[‘butt’</td>
</tr>
</tbody>
</table>
Table 53.7 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dash_joinstyle</td>
<td>[‘miter’</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>data</td>
<td>2D array (rows are x, y) or two 1D arrays</td>
</tr>
<tr>
<td>drawstyle</td>
<td>[‘default’</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>[‘full’</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['-'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>marker</td>
<td>unknown</td>
</tr>
<tr>
<td>markeredgecolor or mec</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markeredgewidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>[‘butt’</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>[‘miter’</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

Example:
The following methods are defined in the `Axes` class:

- **drag_pan**(button, key, x, y)
  Called when the mouse moves during a pan operation.

  
  
  *button* is the mouse button number:
  - 1: LEFT
  - 2: MIDDLE
  - 3: RIGHT

  *key* is a “shift” key

  *x, y* are the mouse coordinates in display coords.

  
  **Note:** Intended to be overridden by new projection types.

- **draw**(artist, renderer, *args, **kwargs)
  Draw everything (plot lines, axes, labels)

- **draw_artist**(a)
  This method can only be used after an initial draw which caches the renderer. It is used to efficiently update Axes data (axis ticks, labels, etc are not updated)

- **end_pan**()
  Called when a pan operation completes (when the mouse button is up.)

  
  **Note:** Intended to be overridden by new projection types.
**errorbar**(*x*, *y*, *yerr=None*, *xerr=None*, *fmt='-'*, *ecolor=None*, *elinewidth=None*, *capsize=3*, *barsabove=False*, *lolims=False*, *uplims=False*, *xlolims=False*, *xuplims=False*, *errorevery=1*, *caphick=None*, **kwargs)

Plot an errorbar graph.

Call signature:

```python
errorbar(x, y, yerr=None, xerr=None, fmt='-', ecolor=None, elinewidth=None, capsize=3, barsabove=False, lolims=False, uplims=False, xlolims=False, xuplims=False, errorevery=1, caphick=None, **kwargs)
```

Plot *x* versus *y* with error deltas in *yerr* and *xerr*. Vertical errorbars are plotted if *yerr* is not *None*. Horizontal errorbars are plotted if *xerr* is not *None*.

*x*, *y*, *xerr*, and *yerr* can all be scalars, which plots a single error bar at *x*, *y*.

Optional keyword arguments:

- **xerr/yerr**: [scalar | N, Nx1, or 2xN array-like] If a scalar number, len(N) array-like object, or an Nx1 array-like object, errorbars are drawn at +/-value relative to the data.
  - If a sequence of shape 2xN, errorbars are drawn at -row1 and +row2 relative to the data.
- **fmt**: '•' The plot format symbol. If *fmt* is *None*, only the errorbars are plotted. This is used for adding errorbars to a bar plot, for example.
- **ecolor**: [None | mpl color] A matplotlib color arg which gives the color the errorbar lines; if *None*, use the marker color.
- **elinewidth**: scalar The linewidth of the errorbar lines. If *None*, use the linewidth.
- **capsize**: scalar The length of the error bar caps in points
- **caphick**: scalar An alias kwarg to 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 caphick. This may change in future releases.
- **barsabove**: [True | False] if *True*, will plot the errorbars above the plot symbols. Default is below.
- **lolims / uplims / xlolims / xuplims**: [False | True] 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 of the same type as *xerr* and *yerr*.
- **errorevery**: positive integer subsamples the errorbars. e.g., if everyerror=5, errorbars for every 5-th datapoint will be plotted. The data plot itself still shows all data points.

All other keyword arguments are passed on to the plot command for the markers. For example, this code makes big red squares with thick green edges:

```python
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

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>data</td>
<td>2D array (rows are x, y) or two 1D arrays</td>
</tr>
<tr>
<td>drawstyle</td>
<td>['default'</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>['full'</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with '%s' conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['-']</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>marker</td>
<td>unknown</td>
</tr>
<tr>
<td>markeredgecolor or mec</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markeredgewidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>
Returns (plotline, caplines, barlinecols):

plotline: Line2D instance x, y plot markers and/or line
caplines: list of error bar cap Line2D instances
barlinecols: list of LineCollection instances for the horizontal and vertical error
ranges.

Example:

![Event plot example](image)

```
eventplot(positions, orientation='horizontal', lineoffsets=1, linelengths=1, linewidths=None, colors=None, linestyles='solid', **kwargs)
```

Plot identical parallel lines at specific positions.

Call signature:

```
eventplot(positions, orientation='horizontal', lineoffsets=0, linelengths=1, linewidths=None, color=None, linestyles='solid')
```

Plot parallel lines at the given positions. positions should be a 1D or 2D array-like object, with each row corresponding to a row or column of lines.

This type of plot is commonly used in neuroscience for representing neural events, where it is commonly 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.

**orientation** [‘horizontal’ | ‘vertical’] ‘horizontal’: the lines will be vertical and arranged in rows
‘vertical’: lines will be horizontal and arranged in columns

**lineoffsets** : A float or array-like containing floats.
**linelengths** : A float or array-like containing floats.
**linewidths** : A float or array-like containing floats.

**colors** must be a sequence of RGBA tuples (eg arbitrary color strings, etc, not allowed) or a list of
such sequences

**linestyles** : [‘solid’ | ‘dashed’ | ‘dashdot’ | ‘dotted’] or an array of these values
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 or column in positions.

Returns a list of :class:`matplotlib.collections.EventCollection` objects that were added.

kwargs are :class:`LineCollection` properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or antialiased</td>
<td>Boolean or sequence of booleans</td>
</tr>
<tr>
<td>array</td>
<td>unknown</td>
</tr>
<tr>
<td>axes</td>
<td>an :class:<code>Axes</code> instance</td>
</tr>
<tr>
<td>clim</td>
<td>a length 2 sequence of floats</td>
</tr>
<tr>
<td>clip_box</td>
<td>a :class:<code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>cmap</td>
<td>a colormap or registered colormap name</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>colorbar</td>
<td>unknown</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or edgecolors</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>facecolor or facecolors</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>figure</td>
<td>a :class:<code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[‘/’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or linestyles or dashes</td>
<td>[‘solid’</td>
</tr>
<tr>
<td>linewidth or lw or linewidths</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>norm</td>
<td>unknown</td>
</tr>
<tr>
<td>offset_position</td>
<td>unknown</td>
</tr>
<tr>
<td>offsets</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>paths</td>
<td>unknown</td>
</tr>
</tbody>
</table>

Continued on next page
Table 53.9 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>picker</td>
<td>[None, float, bool, callable]</td>
</tr>
<tr>
<td>pickradius</td>
<td>unknown</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True, False, None]</td>
</tr>
<tr>
<td>segments</td>
<td>unknown</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>urls</td>
<td>unknown</td>
</tr>
<tr>
<td>verts</td>
<td>unknown</td>
</tr>
<tr>
<td>visible</td>
<td>[True, False]</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

Example:

```
fill(*args, **kwargs)
Plot filled polygons.
Call signature:
```
fill(*args, **kwargs)

*args is a variable length argument, allowing for multiple x, y pairs with an optional color format string; see `plot()` for details on the argument parsing. For example, to plot a polygon with vertices at \( x, y \) in blue:

```python
ax.fill(x, y, 'b')
```

An arbitrary number of \( x, y \), *color* groups can be specified:

```python
ax.fill(x1, y1, 'g', x2, y2, 'r')
```

Return value is a list of `Patch` instances that were added.

The same color strings that `plot()` supports are supported by the fill format string.

If you would like to fill below a curve, e.g., shade a region between 0 and \( y \) along \( x \), use `fill_between()`

The `closed` kwarg will close the polygon when `True` (default).

kwargs control the `Polygon` properties:
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or ec</td>
<td>mpl color spec, or None for default, or 'none' for no color</td>
</tr>
<tr>
<td>facecolor or fc</td>
<td>mpl color spec, or None for default, or 'none' for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[‘/’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>[‘solid’</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float or None for default</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

Example:
**fill_between**($x$, $y_1$, $y_2=0$, $where=None$, $interpolate=False$, **kwargs)

Make filled polygons between two curves.

Call signature:

```python
fill_between($x$, $y_1$, $y_2=0$, $where=None$, **kwargs)
```

Create a *PolyCollection* filling the regions between $y_1$ and $y_2$ where $where==True$

- $x$ : An N-length array of the x data
- $y_1$ : An N-length array (or scalar) of the y data
- $y_2$ : An N-length array (or scalar) of the y data
- $where$ : If *None*, default to fill between everywhere. If not *None*, it is an N-length numpy boolean array and the fill will only happen over the regions where $where==True$.
- $interpolate$ : If *True*, interpolate between the two lines to find the precise point of intersection. Otherwise, the start and end points of the filled region will only occur on explicit values in the $x$ array.
- $kwargs$ : Keyword args passed on to the *PolyCollection*.

$kwargs$ control the *Polygon* properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>antialiased or antialiaseds</td>
<td>Boolean or sequence of booleans</td>
</tr>
<tr>
<td>array</td>
<td>unknown</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clim</td>
<td>a length 2 sequence of floats</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>cmap</td>
<td>a colormap or registered colormap name</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>colorbar</td>
<td>unknown</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or edgecolors</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>facecolor or facecolors</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>['/','','</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with '%s' conversion.</td>
</tr>
<tr>
<td>linestyle or linestyles</td>
<td>['solid', 'dashed', 'dashdot', 'dotted', (offset, on-off-dash-seq)]</td>
</tr>
<tr>
<td>linewidth or lw or linewidths</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>norm</td>
<td>unknown</td>
</tr>
<tr>
<td>offset_position</td>
<td>unknown</td>
</tr>
<tr>
<td>offsets</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>paths</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None]float[boolean][callable]</td>
</tr>
<tr>
<td>pickradius</td>
<td>unknown</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>urls</td>
<td>unknown</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>
between y1 and 0

between y1 and y2

between y1 and 1
fill between where
0.0 0.5 1.0 1.5 2.0
Now regions with y2>1 are masked
See also:

fill_betweenx() for filling between two sets of x-values

fill_betweenx(y, x1, x2=0, where=None, **kwargs)

Make filled polygons between two horizontal curves.

Call signature:

fill_betweenx(y, x1, x2=0, where=None, **kwargs)

Create a PolyCollection filling the regions between x1 and x2 where where==True

y : An N-length array of the y data
x1 : An N-length array (or scalar) of the x data
x2 : An N-length array (or scalar) of the x data
where : If None, default to fill between everywhere. If not None, it is a N length numpy
    boolean array and the fill will only happen over the regions where where==True
kwarg : keyword args passed on to the PolyCollection

kwarg controls the Polygon properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>antialiased</td>
<td>Boolean or sequence of booleans</td>
</tr>
<tr>
<td>array</td>
<td>unknown</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clim</td>
<td>a length 2 sequence of floats</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>cmap</td>
<td>a colormap or registered colormap name</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>colorbar</td>
<td>unknown</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>facecolor</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>['/'</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle</td>
<td>['solid'</td>
</tr>
<tr>
<td>linewidth</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>norm</td>
<td>unknown</td>
</tr>
<tr>
<td>offset_position</td>
<td>unknown</td>
</tr>
<tr>
<td>offsets</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>paths</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None]float[boolean]calllable</td>
</tr>
<tr>
<td>pickradius</td>
<td>unknown</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>urls</td>
<td>unknown</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>
between $y_1$ and 0

between $y_1$ and 1

between $y_1$ and $y_2$
Now regions with $y_2 > 1$ are masked

See also:

- `fill_between()` for filling between two sets of y-values
- `format_coord(x, y)`
  - Return a format string formatting the x, y coord
- `format_xdata(x)`
  - Return x string formatted. This function will use the attribute `self.fmt_xdata` if it is callable, else will fall back on the xaxis major formatter
- `format_ydata(y)`
  - Return y string formatted. This function will use the `fmt_ydata` attribute if it is callable, else will fall back on the yaxis major formatter
- `get_adjustable()`
- `get_anchor()`
- `get_aspect()`
- `get_autoscale_on()`
  - Get whether autoscaling is applied for both axes on plot commands
get_autoscalex_on()
Get whether autoscaling for the x-axis is applied on plot commands

get_autoscaley_on()
Get whether autoscaling for the y-axis is applied on plot commands

get_axes_locator()
return axes_locator

get_axis_bgcolor()
Return the axis background color

get_axisbelow()
Get whether axis below is true or not

get_children()
return a list of child artists

get_cursor_props()
Return the cursor properties as a (linewidth, color) tuple, where linewidth is a float and color is an RGBA tuple

get_data_ratio()
Returns the aspect ratio of the raw data.
This method is intended to be overridden by new projection types.

get_data_ratio_log()
Returns the aspect ratio of the raw data in log scale. Will be used when both axis scales are in log.

get_default_bbox_extra_artists()

get_frame_on()
Get whether the axes rectangle patch is drawn

get_images()
return a list of Axes images contained by the Axes

get_legend()
Return the legend.Legend instance, or None if no legend is defined

get_legend_handles_labels(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)

get_lines()
Return a list of lines contained by the Axes

get_navigate()
Get whether the axes responds to navigation commands
get_navigate_mode()
Get the navigation toolbar button status: ‘PAN’, ‘ZOOM’, or None

get_position(original=False)
Return the a copy of the axes rectangle as a Bbox

get_rasterization_zorder()
Get zorder value below which artists will be rasterized

get_renderer_cache()

get_shared_x_axes()
Return a copy of the shared axes Grouper object for x axes

get_shared_y_axes()
Return a copy of the shared axes Grouper object for y axes

get_tightbbox(renderer, call_axes_locator=True)
Return the tight bounding box of the axes. The dimension of the Bbox in canvas coordinate.
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.

get_title(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, optional
Which title to get, defaults to ‘center’

Returns  title: str :
The title text string.

get_window_extent(*args, **kwargs)
get the axes bounding box in display space; args and kwargs are empty

get_xaxis()
Return the XAxis instance

get_xaxis_text1_transform(pad_points)
Get the transformation used for drawing x-axis labels, which will add the given amount of padding
(in points) between the axes and the label. The x-direction is in data coordinates and the y-
direction is in axis coordinates. Returns a 3-tuple of the form:

(transform, valign, halign)

where valign and halign are requested alignments for the text.

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_xaxis_text2_transform(pad_points)
Get the transformation used for drawing the secondary x-axis labels, which will add the given amount of padding (in points) between the axes and the label. The x-direction is in data coordinates and the y-direction is in axis coordinates. Returns a 3-tuple of the form:

(transform, valign, halign)

where valign and halign are requested alignments for the text.

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_xaxis_transform(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_xbound()
Returns the x-axis numerical bounds where:

lowerBound < upperBound

get_xgridlines()
Get the x grid lines as a list of Line2D instances

get_xlabel()
Get the xlabel text string.

get_xlim()
Get the x-axis range [left, right]

get_xmajor ticklabels()
Get the xtick labels as a list of Text instances.

get_xminorticklabels()
Get the x minor tick labels as a list of matplotlib.text.Text instances.

get_xscale()
Return the xaxis scale string: linear, log, symlog

get_xticklabels(minor=False)
Get the x tick labels as a list of Text instances.

get_xticklines()
Get the xtick lines as a list of Line2D instances

get_xticks(minor=False)
Return the x ticks as a list of locations
get_yaxis()
Return the YAxis instance

gget_yaxis_text1_transform(pad_points)
Get the transformation used for drawing y-axis labels, which will add the given amount of padding (in points) between the axes and the label. The x-direction is in axis coordinates and the y-direction is in data coordinates. Returns a 3-tuple of the form:

(transform, valign, halign)

where valign and halign are requested alignments for the text.

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_text2_transform(pad_points)
Get the transformation used for drawing the secondary y-axis labels, which will add the given amount of padding (in points) between the axes and the label. The x-direction is in axis coordinates and the y-direction is in data coordinates. Returns a 3-tuple of the form:

(transform, valign, halign)

where valign and halign are requested alignments for the text.

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_transform(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.

get_ybound()
Return y-axis numerical bounds in the form of lowerBound < upperBound

get_ygridlines()
Get the y grid lines as a list of Line2D instances

get_ylabel()
Get the ylabel text string.

get_ylim()
Get the y-axis range [bottom, top]

get_ymajorticklabels()
Get the major y tick labels as a list of Text instances.
get_yminorticklabels()
Get the minor y tick labels as a list of Text instances.

get_yscale()
Return the yaxis scale string: linear, log, symlog

get_yticklabels(minor=False)
Get the y tick labels as a list of Text instances

get_yticklines()
Get the ytick lines as a list of Line2D instances

get_yticks(minor=False)
Return the y ticks as a list of locations

grid(b=None, which='major', axis='both', **kwargs)
Turn the axes grids on or off.

    Call signature:

    grid(self, b=None, which='major', axis='both', **kwargs)

Set the axes grids on or off; b is a boolean. (For MATLAB compatibility, b may also be a string, ‘on’ or ‘off’.)

If b is None and len(kwargs)==0, toggle the grid state. If kwargs are supplied, it is assumed that you want a grid and b is thus set to True.

which can be ‘major’ (default), ‘minor’, or ‘both’ to control whether major tick grids, minor tick grids, or both are affected.

axis can be ‘both’ (default), ‘x’, or ‘y’ to control which set of gridlines are drawn.

kwargs are used to set the grid line properties, eg:

    ax.grid(color='r', linestyle='-', linewidth=2)

Valid Line2D kwargs are

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>['butt'</td>
</tr>
</tbody>
</table>
| dash_joinstyle         | ['miter' | 'round' | 'bevel']
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>data</td>
<td>2D array (rows are x, y) or two 1D arrays</td>
</tr>
<tr>
<td>drawstyle</td>
<td>['default'</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>['full'</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with '%s' conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['-']</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>marker</td>
<td>unknown</td>
</tr>
<tr>
<td>markeredgecolor or mec</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markeredgewidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

**has_data()**

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.

**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='none', reduce_C_function=<function mean at 0x14e86e0>, mincnt=None, marginals=False, **kwargs)

Make a hexagonal binning plot.
Call signature:

```python
hexbin(x, y, C = None, gridsize = 100, bins = None,
    xscale = 'linear', yscale = 'linear',
    cmap=None, norm=None, vmin=None, vmax=None,
    alpha=None, linewidths=None, edgecolors='none'
    reduce_C_function = np.mean, mincnt=None, marginals=True
    **kwargs)
```

Make a hexagonal binning plot of $x$ versus $y$, where $x$, $y$ are 1-D sequences of the same length, $N$. If $C$ is `None` (the default), this is a histogram of the number of occurences of the observations at $(x[i],y[i])$.

If $C$ is specified, it specifies values at the coordinate $(x[i],y[i])$. These values are accumulated for each hexagonal bin and then reduced according to `reduce_C_function`, which defaults to numpy’s mean function (`np.mean`). (If $C$ is specified, it must also be a 1-D sequence of the same length as $x$ and $y$.)

$x$, $y$ and/or $C$ may be masked arrays, in which case only unmasked points will be plotted.

Optional keyword arguments:

- **gridsize**: `[ 100 | integer ]` The number of hexagons in the $x$-direction, default is 100. The corresponding number of hexagons in the $y$-direction is chosen such that the hexagons are approximately regular. Alternatively, `gridsize` can be a tuple with two elements specifying the number of hexagons in the $x$-direction and the $y$-direction.

- **bins**: `[ None | 'log' | integer | sequence ]` 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.

  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' ]` Use a linear or log10 scale on the horizontal axis.

- **yscale**: `[ 'linear' | 'log' ]` Use a linear or log10 scale on the vertical axis.

- **mincnt**: `[ None | a positive integer ]` If not `None`, only display cells with more than `mincnt` number of points in the cell.

- **marginals**: `[ True | 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**: `[ None | scalars (left, right, bottom, top) ]` The limits of the bins. The default assigns the limits based on `gridsize`, $x$, $y$, `xscale` and `yscale`.

Other keyword arguments controlling color mapping and normalization arguments:

- **cmap**: `[ None | Colormap ]` a `matplotlib.colors.Colormap` instance. If `None`, defaults to rc `image.cmap`.

- **norm**: `[ None | Normalize ]` `matplotlib.colors.Normalize` instance is used to scale luminance data to 0,1.

- **vmin / vmax**: scalar `vmin` and `vmax` are used in conjunction with `norm` to normalize luminance data. If either are `None`, the min and max of the color array $C$ is used. Note if you pass a norm instance, your settings for `vmin` and `vmax` will be ignored.
**alpha**: scalar between 0 and 1, or `None` the alpha value for the patches

**linewidths**: [ `None` | scalar ] If `None`, defaults to rc lines.linewidth. Note that this is a tuple, and if you set the linewidths argument you must set it as a sequence of floats, as required by `RegularPolyCollection`.

Other keyword arguments controlling the Collection properties:

**edgecolors**: [ `None` | ’none’ | `mpl` color | color sequence ] If ’none’, draws the edges in the same color as the fill color. This is the default, as it avoids unsightly unpainted pixels between the hexagons.

If `None`, draws the outlines in the default color.

If a matplotlib color arg or sequence of rgba tuples, draws the outlines in the specified color.

Here are the standard descriptions of all the `Collection` kwargs:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or antialiaseds</td>
<td>Boolean or sequence of booleans</td>
</tr>
<tr>
<td>array</td>
<td>unknown</td>
</tr>
<tr>
<td>axes</td>
<td>an <code>Axes</code> instance</td>
</tr>
<tr>
<td>clim</td>
<td>a length 2 sequence of floats</td>
</tr>
<tr>
<td>clip_box</td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[ (Path, Transform)</td>
</tr>
<tr>
<td>cmap</td>
<td>a colormap or registered colormap name</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>colorbar</td>
<td>unknown</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or edgecolors</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>facecolor or facecolors</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>figure</td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[ ’/’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or linestyles or dashes</td>
<td>[’solid’</td>
</tr>
<tr>
<td>linewidth or lw or linewidths</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>norm</td>
<td>unknown</td>
</tr>
<tr>
<td>offset_position</td>
<td>unknown</td>
</tr>
<tr>
<td>offsets</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>paths</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None]float</td>
</tr>
<tr>
<td>pickradius</td>
<td>unknown</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
</tbody>
</table>

Continued on next page
Table 53.13 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>urls</td>
<td>unknown</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

The return value is a PolyCollection instance; use get_array() on this PolyCollection to get the counts in each hexagon. If marginals is True, horizontal bar and vertical bar (both PolyCollections) will be attached to the return collection as attributes hbar and vbar.

Example:

```python
hist(x, bins=10, range=None, normed=False, weights=None, cumulative=False, bottom=None, histtype='bar', align='mid', orientation='vertical', rwidth=None, log=False, color=None, label=None, stacked=False, **kwargs)
```

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.
Multiple data can be provided via \texttt{x} as a list of datasets of potentially different length ([\texttt{x0}, \texttt{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 at present.

**Parameters**

- \texttt{x} : array_like, shape (n,)
  Input values.
- \texttt{bins} : integer or array_like, optional, default: 10
  If an integer is given, \texttt{bins + 1} bin edges are returned, consistently with \texttt{numpy.histogram()} for numpy version \texttt{>= 1.3}.

Unequally spaced bins are supported if \texttt{bins} is a sequence.

- \texttt{range} : tuple, optional, default: None
  The lower and upper range of the bins. Lower and upper outliers are ignored.
  If not provided, \texttt{range} is (\texttt{x.min()}, \texttt{x.max()}). Range has no effect if \texttt{bins} is a sequence.

If \texttt{bins} is a sequence or \texttt{range} is specified, autoscaling is based on the specified bin range instead of the range of \texttt{x}.

- \texttt{normed} : boolean, optional, default: False
  If \texttt{True}, the first element of the return tuple will be the counts normalized to form a probability density, i.e., \( n/(\text{len(x)} \cdot \text{dbin}) \), i.e, the integral of the histogram will sum to 1. If \texttt{stacked} is also \texttt{True}, the sum of the histograms is normalized to 1.

- \texttt{weights} : array_like, shape (n,), optional, default: None
  An array of weights, of the same shape as \texttt{x}. Each value in \texttt{x} only contributes its associated weight towards the bin count (instead of 1). If \texttt{normed} is True, the weights are normalized, so that the integral of the density over the range remains 1.

- \texttt{cumulative} : boolean, optional, default
  If \texttt{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 \texttt{normed} is also \texttt{True} then the histogram is normalized such that the last bin equals 1. If \texttt{cumulative} evaluates to less than 0 (e.g., -1), the direction of accumulation is reversed. In this case, if \texttt{normed} is also \texttt{True}, then the histogram is normalized such that the first bin equals 1.

- \texttt{histtype} : ['bar' | 'barstacked' | 'step' | 'stepfilled'], optional
  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.

- \texttt{align} : ['left' | 'mid' | 'right'], optional, default: 'mid'
  Controls how the histogram is plotted.
  - '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.

- \texttt{orientation} : ['horizontal' | 'vertical'], optional
If ‘horizontal’, barh will be used for bar-type histograms and the bottom kwarg will be the left edges.

**rwidth** : scalar, optional, default: None
The relative width of the bars as a fraction of the bin width. If None, automatically compute the width. Ignored if histtype = ‘step’ or ‘stepfilled’.

**log** : boolean, optional, default
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, optional, default: None
Color spec or sequence of color specs, one per dataset. Default (None) uses the standard line color sequence.

**label** : string, optional, default: ‘’
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 the legend command will work as expected.

**stacked** : boolean, optional, default
If True, multiple data are stacked on top of each other If False multiple data are aranged side by side if histtype is ‘bar’ or on top of each other if histtype is ‘step’

**Returns** : tuple : (n, bins, patches) or ([n0, n1, ...], bins, [patches0, patches1,...])

**See also**:

hist2d 2D histograms

**Notes**

Until numpy release 1.5, the underlying numpy histogram function was incorrect with normed = ‘True if bin sizes were unequal. MPL inherited that error. It is now corrected within MPL when using earlier numpy versions.
Examples

**hist2d** *(x, y, bins=10, range=None, normed=False, weights=None, cmin=None, cmax=None, **kwargs)*

Make a 2D histogram plot.

**Parameters**

- **x, y**: array_like, shape (n, )
  - Input values

- **bins**: [None | int | [int, int] | array_like | [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, default: None
  - 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.

- **normed**: boolean, optional, default: False
  - Normalize histogram.

- **weights**: array_like, shape (n, ), optional, default: None
  - An array of values w_i weighing each sample (x_i, y_i).

- **cmin**: scalar, optional, default: None
All bins that have count less than cmin will not be displayed and these count values in the return value count histogram will also be set to nan upon return.

cmax : scalar, optional, default: None
All bins that have count more than cmax will not be displayed (set to none before passing to imshow) and these count values in the return value count histogram will also be set to nan upon return.

Returns
The return value is ``(counts, xedges, yedges, Image)``.

Other Parameters
``kwargs``:
``pcolorfast()`` properties.

See also:
``hist`` 1D histogram

Notes

Rendering the histogram with a logarithmic color scale is accomplished by passing a `colors.LogNorm` instance to the `norm` keyword argument.

Examples

```python
hlines(y, xmin, xmax, colors='k', linestyles='solid', label='', **kwargs)
```

Plot horizontal lines.
Plot horizontal lines at each y from xmin to xmax.

**Parameters**
- **y**: scalar or 1D array_like
  - y-indexes where to plot the lines.
- **xmin, xmax**: scalar or 1D array_like
  - Respective beginning and end of each line. If scalars are provided, all lines will have same length.
- **colors**: array_like of colors, optional, default: ‘k’
- **linestyles**: ['solid' | 'dashed' | 'dashdot' | 'dotted'], optional
- **label**: string, optional, default: ‘’

**Returns**
- **lines**: LineCollection

**Other Parameters**
- **kwargs**: LineCollection properties.

See also:
- **vlines** vertical lines

**Examples**

```
hold(b=None)
```

Call signature:

```
hold(b=None)
```

Set the hold state. If hold is None (default), toggle the hold state. Else set the hold state to boolean value b.

Examples:

```
# toggle hold
del hold
hold()

# turn hold on
```
hold(True)

# turn hold off
hold(False)

When hold is True, subsequent plot commands will be added to the current axes. When hold is False, the current axes and figure will be cleared on the next plot command.

```python
imshow(X, cmap=None, norm=None, aspect=None, interpolation=None, alpha=None, vmin=None, vmax=None, origin=None, extent=None, shape=None, filternorm=1, filterrad=4.0, imlim=None, resample=None, url=None, **kwargs)
```

Display an image on the axes.

**Parameters**

- **X**: array_like, shape (n, m) or (n, m, 3) or (n, m, 4)
  
  Display the image in X to current axes. X may be a float array, a uint8 array or a PIL image. If X is an array, it can have the following shapes:
  - MxN – luminance (grayscale, float array only)
  - MxNx3 – RGB (float or uint8 array)
  - MxNx4 – RGBA (float or uint8 array)
  
  The value for each component of MxNx3 and MxNx4 float arrays should be in the range 0.0 to 1.0; MxN float arrays may be normalised.

- **cmap** : Colormap, optional, default: None
  
  If None, default to rc image.cmap value. cmap is ignored when X has RGB(A) information.

- **aspect** : ['auto' | 'equal' | scalar], optional, default: None
  
  If ‘auto’, changes the image aspect ratio to match that of the axes.
  
  If ‘equal’, and extent is None, changes the axes aspect ratio to match that of the image. If extent is not None, the axes aspect ratio is changed to match that of the extent.
  
  If None, default to rc image.aspect value.

- **interpolation** : string, optional, default: None
  
  
  If interpolation is None, default to rc image.interpolation. See also the filternorm and filterrad parameters. If interpolation is ‘none’, then no interpolation is performed on the Agg, ps and pdf backends. Other backends will fall back to ‘nearest’.

- **norm** : Normalize, optional, default: None
  
  A Normalize instance is used to scale luminance data to 0, 1. If None, use the default function: norm is only used if X is an array of floats.

- **vmin, vmax** : scalar, optional, default: None
  
  vmin and vmax are used in conjunction with norm to normalize luminance data. Note if you pass a norm instance, your settings for vmin and vmax will be ignored.

- **alpha** : scalar, optional, default: None
  
  The alpha blending value, between 0 (transparent) and 1 (opaque)

- **origin** : ['upper' | 'lower'], optional, default: None
  
  Place the [0,0] index of the array in the upper left or lower left corner of the axes.
If None, default to rc `image.origin`.

**extent**: scalars (left, right, bottom, top), optional, default: None

Data limits for the axes. The default assigns zero-based row, column indices to the x, y centers of the pixels.

**shape**: scalars (columns, rows), optional, default: None

For raw buffer images

**filternorm**: scalar, optional, default: 1

A parameter for the antigrain image resize filter. From the antigrain documentation, if `filternorm = 1`, 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**: scalar, optional, default: 4.0

The filter radius for filters that have a radius parameter, i.e. when interpolation is one of: ‘sinc’, ‘lanczos’ or ‘blackman’

**Returns**

**image**: `AxesImage`

**Other Parameters**

**kwargs**: `Artist` properties.

**See also**:

- `matshow`: Plot a matrix or an array as an image.
Examples

```
in_axes(mouseevent)
    Return True if the given mouseevent (in display coords) is in the Axes

invert_xaxis()
    Invert the x-axis.

invert_yaxis()
    Invert the y-axis.

ishold()
    return the HOLD status of the axes

legend(*args, **kwargs)
    Place a legend on the current axes.

    Call signature:

    legend(*args, **kwargs)

    Places legend at location loc. Labels are a sequence of strings and loc can be a string or an integer specifying the legend location.

    To make a legend with existing lines:
```
legend()

`legend()` by itself will try and build a legend using the label property of the lines/patches/collections. You can set the label of a line by doing:

```python
plot(x, y, label='my data')
```

or:

```python
line.set_label('my data').
```

If label is set to `'_nolegend_'`, the item will not be shown in legend.

To automatically generate the legend from labels:

```python
legend( ('label1', 'label2', 'label3') )
```

To make a legend for a list of lines and labels:

```python
legend( (line1, line2, line3), ('label1', 'label2', 'label3') )
```

To make a legend at a given location, using a location argument:

```python
legend( ('label1', 'label2', 'label3'), loc='upper left')
```

or:

```python
legend((line1, line2, line3), ('label1', 'label2', 'label3'), loc=2)
```

The location codes are

<table>
<thead>
<tr>
<th>Location String</th>
<th>Location Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>'best'</td>
<td>0</td>
</tr>
<tr>
<td>'upper right'</td>
<td>1</td>
</tr>
<tr>
<td>'upper left'</td>
<td>2</td>
</tr>
<tr>
<td>'lower left'</td>
<td>3</td>
</tr>
<tr>
<td>'lower right'</td>
<td>4</td>
</tr>
<tr>
<td>'right'</td>
<td>5</td>
</tr>
<tr>
<td>'center left'</td>
<td>6</td>
</tr>
<tr>
<td>'center right'</td>
<td>7</td>
</tr>
<tr>
<td>'lower center'</td>
<td>8</td>
</tr>
<tr>
<td>'upper center'</td>
<td>9</td>
</tr>
<tr>
<td>'center'</td>
<td>10</td>
</tr>
</tbody>
</table>

Users can specify any arbitrary location for the legend using the `bbox_to_anchor` keyword argument. `bbox_to_anchor` can be an instance of BboxBase (or its derivatives) or a tuple of 2 or 4 floats. For example:

```python
loc = 'upper right', bbox_to_anchor = (0.5, 0.5)
```
Matplotlib, Release 1.3.0

will place the legend so that the upper right corner of the legend at the center of the axes.

The legend location can be specified in other coordinate, by using the `bbox_transform` keyword.

The loc itself can be a 2-tuple giving x,y of the lower-left corner of the legend in axes coords
(`bbox_to_anchor` is ignored).

Keyword arguments:

- **prop**: [None | FontProperties | dict] A matplotlib.font_manager.FontProperties instance. If `prop` is a dictionary, a new instance will be created with `prop`. If None, use rc settings.
- **fontsize**: [size in points | ‘xx-small’ | ‘x-small’ | ‘small’ | ‘medium’ | ‘large’ | ‘x-large’ | ‘xx-large’ ]
  Set the font size. May be either a size string, relative to the default font size, or an absolute font size in points. This argument is only used if prop is not specified.
- **numpoints**: integer The number of points in the legend for line
- **scatterpoints**: integer The number of points in the legend for scatter plot
- **scatteryoffsets**: list of floats a list of yoffsets for scatter symbols in legend
- **markerscale**: [None | scalar ] The relative size of legend markers vs. original. If None, use rc settings.
- **frameon**: [True | False] if True, draw a frame around the legend. The default is set by the rcParam ‘legend.frameon’
- **fancybox**: [None | False | True ] if True, draw a frame with a round fancybox. If None, use rc settings.
- **shadow**: [None | False | True ] If True, draw a shadow behind legend. If None, use rc settings.
- **framealpha**: [None | float] If not None, alpha channel for legend frame. Default None.
- **ncol**: integer] number of columns. default is 1
- **mode**: [[“expand” | None ]] if mode is “expand”, the legend will be horizontally expanded to fill the axes area (or `bbox_to_anchor`)
- **bbox_to_anchor**: an instance of BboxBase or a tuple of 2 or 4 floats the bbox that the legend will be anchored.
- **bbox_transform** [[ an instance of Transform | None ]] the transform for the bbox. transAxes if None.
- **title**: [string] the legend title

Padding and spacing between various elements use following keywords parameters. These values are measure in font-size units. e.g., a fontsize of 10 points and a handlelength=5 implies a handlelength of 50 points. Values from rcParams will be used if None.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>borderpad</td>
<td>the fractional whitespace inside the legend border</td>
</tr>
<tr>
<td>labelspacing</td>
<td>the vertical whitespace between the legend entries</td>
</tr>
<tr>
<td>handlelength</td>
<td>the length of the legend handles</td>
</tr>
<tr>
<td>handletextpad</td>
<td>the pad between the legend handle and text</td>
</tr>
<tr>
<td>borderaxespad</td>
<td>the pad between the axes and legend border</td>
</tr>
<tr>
<td>columnspacing</td>
<td>the spacing between columns</td>
</tr>
</tbody>
</table>

Note: Not all kinds of artist are supported by the legend command. See Legend guide for details.

Example:
See also:

Legend guide.

**locator_params** *(axis='both', tight=None, **kwargs)*

Control behavior of tick locators.

Keyword arguments:

*axis* ['x' | 'y' | 'both'] Axis on which to operate; default is 'both'.

*tight* [True | False | None] Parameter passed to autoscale_view(). Default is None, for no change.

Remaining keyword arguments are passed to directly to the set_params() method.

Typically one might want to reduce the maximum number of ticks and use tight bounds when plotting small subplots, for example:

```
ax.locator_params(tight=True, nbins=4)
```

Because the locator is involved in autoscaling, autoscale_view() is called automatically after the parameters are changed.

This presently works only for the MaxNLocator used by default on linear axes, but it may be generalized.

**loglog** (*args, **kwargs)*

Make a plot with log scaling on both the x and y axis.
Call signature:

```
loglog(*args, **kwargs)
```


Notable keyword arguments:

- **`basex/basey`: scalar > 1** Base of the x/y logarithm
- **`subsx/subsy`: [None | sequence]** The location of the minor x/y ticks; `None` defaults to autosubs, which depend on the number of decades in the plot; see `matplotlib.axes.Axes.set_xscale()` / `matplotlib.axes.Axes.set_yscale()` for details
- **`nonposx/nonposy`: ['mask' | 'clip']** Non-positive values in x or y can be masked as invalid, or clipped to a very small positive number

The remaining valid kwargs are `Line2D` properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an <code>Axes</code> instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[ (Path, Transform)</td>
</tr>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>data</td>
<td>2D array (rows are x, y) or two 1D arrays</td>
</tr>
<tr>
<td>drawstyle</td>
<td>['default'</td>
</tr>
<tr>
<td>figure</td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>['full'</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['-.'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>marker</td>
<td>unknown</td>
</tr>
<tr>
<td>markeredgecolor or mec</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markeredgewidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>None</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

Example:

```
semilogy

semilogx

loglog base 4 on x

Errorbars go negative

margins(*args, **kw)
Set or retrieve autoscaling margins.
```
signatures:

margins()
returns xmargin, ymargin

margins(margin)

margins(xmargin, ymargin)

margins(x=xmargin, y=ymargin)

margins(..., tight=False)

All three forms above set the xmargin and ymargin parameters. All keyword parameters are optional. A single argument specifies both xmargin and ymargin. 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.

matshow(Z, **kwargs)
Plot a matrix or array as an 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 shape (n, m)
The matrix to be displayed.

Returns image : AxesImage

Other Parameters kwargs : imshow arguments
Sets origin to ‘upper’, ‘interpolation’ to ‘nearest’ and ‘aspect’ to equal.

See also:

imshow plot an image
minorticks_off()
    Remove minor ticks from the axes.

minorticks_on()
    Add autoscaling minor ticks to the axes.

name = 'rectilinear'

pcolor(*args, **kwargs)
    Create a pseudocolor plot of a 2-D array.

**Note:** pcolor can be very slow for large arrays; consider using the similar but much faster pcolormesh() instead.

Call signatures:

    pcolor(C, **kwargs)
    pcolor(X, Y, C, **kwargs)

    C is the array of color values.

    X and Y, if given, specify the (x, y) coordinates of the colored quadrilaterals; the quadrilateral for C[i,j] has corners at:
(X[i, j], Y[i, j]),
(X[i, j+1], Y[i, j+1]),
(X[i+1, j], Y[i+1, j]),
(X[i+1, j+1], Y[i+1, j+1]).

Ideally the dimensions of X and Y should be one greater than those of C; if the dimensions are the same, then the last row and column of C will be ignored.

Note that the column index corresponds to the x-coordinate, and the row index corresponds to y; for details, see the Grid Orientation section below.

If either or both of X and Y are 1-D arrays or column vectors, they will be expanded as needed into the appropriate 2-D arrays, making a rectangular grid.

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.

Keyword arguments:

- **cmap:** [None | Colormap] A `matplotlib.colors.Colormap` instance. If None, use rc settings.
- **norm:** [None | Normalize] An `matplotlib.colors.Normalize` instance is used to scale luminance data to 0,1. If None, defaults to `normalize()`.
- **vmin/vmax:** [None | scalar] vmin and vmax are used in conjunction with norm to normalize luminance data. If either is None, it is autoscaled to the respective min or max of the color array C. If not None, vmin or vmax passed in here override any pre-existing values supplied in the norm instance.
- **shading:** ['flat' | 'faceted'] If ‘faceted’, a black grid is drawn around each rectangle; if ‘flat’, edges are not drawn. Default is ‘flat’, contrary to MATLAB.

This kwarg is deprecated; please use ‘edgecolors’ instead:

- shading=’flat’ – edgecolors=’none’
- shading=’faceted’ – edgecolors=’k’

- **edgecolors:** [None | ’none’ | color | color sequence] If None, the rc setting is used by default.

If ‘none’, edges will not be visible.

An mpl color or sequence of colors will set the edge color

- **alpha:** 0 <= scalar <= 1 or None the alpha blending value

Return value is a `matplotlib.collections.Collection` instance. The grid orientation follows the MATLAB convention: an array C with shape `(nrows, ncolumns)` is plotted with the column number as X and the row number as Y, increasing up; hence it is plotted the way the array would be printed, except that the Y axis is reversed. That is, C is taken as `C*(y,x)`.

Similarly for `meshgrid()`:

```python
x = np.arange(5)
y = np.arange(3)
X, Y = np.meshgrid(x, y)
```

is equivalent to:
X = array([[0, 1, 2, 3, 4],
          [0, 1, 2, 3, 4],
          [0, 1, 2, 3, 4]])

Y = array([[0, 0, 0, 0, 0],
          [1, 1, 1, 1, 1],
          [2, 2, 2, 2, 2]])

so if you have:

C = rand(len(x), len(y))

then you need to transpose C:

pcolor(X, Y, C.T)

or:

pcolor(C.T)

MATLAB `pcolor()` always discards the last row and column of C, but matplotlib displays the last row and column if X and Y are not specified, or if X and Y have one more row and column than C.

kwarg can be used to control the PolyCollection properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or antialiased</td>
<td>Boolean or sequence of booleans</td>
</tr>
<tr>
<td>array</td>
<td>unknown</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clim</td>
<td>a length 2 sequence of floats</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>cmap</td>
<td>a colormap or registered colormap name</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>colorbar</td>
<td>unknown</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgcolor or edgecolors</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>facecolor or facecolors</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[ ‘/’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
</tbody>
</table>

Continued on next page
Table 53.15 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>linestyle or linestyles or dashes</td>
<td>['solid', 'dashed', 'dashdot', 'dotted'] (offset, on-off-dash-seq)</td>
</tr>
<tr>
<td>linewidth or lw or linewidths</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>norm</td>
<td>unknown</td>
</tr>
<tr>
<td>offset_position</td>
<td>unknown</td>
</tr>
<tr>
<td>offsets</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>paths</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>pickradius</td>
<td>unknown</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>urls</td>
<td>unknown</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

Note: The default antialiased 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 antialiased is taken from rcParams['patch.antialiased'], which defaults to True. Stroking the edges may be preferred if alpha is 1, but will cause artifacts otherwise.

See also:

pcolormesh() For an explanation of the differences between pcolor and pcolormesh.

pcolorfast(*args, **kwargs)
pseudocolor plot of a 2-D array

Experimental; this is a pcolor-type method that provides the fastest possible rendering with the Agg backend, and that can handle any quadrilateral grid. It supports only flat shading (no outlines), it lacks support for log scaling of the axes, and it does not have a pyplot wrapper.

Call signatures:

ax.pcolorfast(C, **kwargs)
ax.pcolorfast(xr, yr, C, **kwargs)
ax.pcolorfast(x, y, C, **kwargs)
ax.pcolorfast(X, Y, C, **kwargs)

C is the 2D array of color values corresponding to quadrilateral cells. Let (nr, nc) be its shape. C may be a masked array.
ax.pcolorfast(C, **kwargs) is equivalent to ax.pcolorfast([0,nc], [0,nr], C, **kwargs)

xr, yr specify the ranges of x and y corresponding to the rectangular region bounding C. If:

xr = [x0, x1]

and:

yr = [y0, y1]

then x goes from x0 to x1 as the second index of C goes from 0 to nc, etc. (x0, y0) is the outermost corner of cell (0,0), and (x1, y1) is the outermost corner of cell (nr-1, nc-1). All cells are rectangles of the same size. This is the fastest version.

x, y are 1D arrays of length nc +1 and nr +1, respectively, giving the x and y boundaries of the cells. Hence the cells are rectangular but the grid may be nonuniform. The speed is intermediate. (The grid is checked, and if found to be uniform the fast version is used.)

X and Y are 2D arrays with shape (nr +1, nc +1) that specify the (x,y) coordinates of the corners of the colored quadrilaterals; the quadrilateral for C[i,j] has corners at (X[i,j],Y[i,j]), (X[i,j+1],Y[i,j+1]), (X[i+1,j],Y[i+1,j]), (X[i+1,j+1],Y[i+1,j+1]). The cells need not be rectangular. 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.

Note that the column index corresponds to the x-coordinate, and the row index corresponds to y; for details, see the “Grid Orientation” section below.

Optional keyword arguments:

- **cmap**: [ None | Colormap ] A matplotlib.colors.Colormap instance from cm. If None, use rc settings.
- **norm**: [ None | Normalize ] A matplotlib.colors.Normalize instance is used to scale luminance data to 0,1. If None, defaults to normalize()
- **vmin/vmax**: [ None | scalar ] vmin and vmax are used in conjunction with norm to normalize luminance data. If either are None, the min and max of the color array C is used. If you pass a norm instance, vmin and vmax will be None.
- **alpha**: 0 <= scalar <= 1 or None the alpha blending value

Return value is an image if a regular or rectangular grid is specified, and a QuadMesh collection in the general quadrilateral case.

pcolormesh(*args, **kwargs)
Plot a quadrilateral mesh.

Call signatures:

pcolormesh(C)
pcolormesh(X, Y, C)
pcolormesh(C, **kwargs)

Create a pseudocolor plot of a 2-D array.
pcolormesh is similar to pcolor(), but uses a different mechanism and returns a different object; pcolor returns a PolyCollection but pcolormesh returns a QuadMesh. It is much faster, so it is almost always preferred for large arrays.

C may be a masked array, but X and Y may not. Masked array support is implemented via cmap and norm; in contrast, pcolor() simply does not draw quadrilaterals with masked colors or vertices.

Keyword arguments:

- **cmap**: [None | Colormap] A matplotlib.colors.Colormap instance. If None, use rc settings.
- **norm**: [None | Normalize] A matplotlib.colors.Normalize instance is used to scale luminance data to 0,1. If None, defaults to normalize().
- **vmin/vmax**: [None | scalar] vmin and vmax are used in conjunction with norm to normalize luminance data. If either is None, it is autoscaled to the respective min or max of the color array $C$. If not None, vmin or vmax passed in here override any pre-existing values supplied in the norm instance.
- **shading**: [‘flat‘ | ‘gouraud‘] ‘flat‘ indicates a solid color for each quad. When ‘gouraud’, each quad will be Gouraud shaded. When gouraud shading, edgecolors is ignored.
- **edgecolors**: [None | ‘None‘ | ‘face‘ | color | color sequence] If None, the rc setting is used by default.
  
  If ‘None”, edges will not be visible.
  
  If ‘face’, edges will have the same color as the faces.

An mpl color or sequence of colors will set the edge color

- **alpha**: 0 <= scalar <= 1 or None the alpha blending value

Return value is a matplotlib.collections.QuadMesh object.

kwargs can be used to control the matplotlib.collections.QuadMesh properties:
Table 53.16 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>facecolor or facecolors</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>['/', '', '</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with '%s' conversion.</td>
</tr>
<tr>
<td>linestyle or linestyle or dashes</td>
<td>['solid', 'dashed', 'dashdot', 'dotted'] [offset, on-off-dash-seq]</td>
</tr>
<tr>
<td>linewidth or lw or linewidths</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>norm</td>
<td>unknown</td>
</tr>
<tr>
<td>offset_position</td>
<td>unknown</td>
</tr>
<tr>
<td>offsets</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>paths</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None][float][boolean][callable]</td>
</tr>
<tr>
<td>pickradius</td>
<td>unknown</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>urls</td>
<td>unknown</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

See also:

pcolor() For an explanation of the grid orientation and the expansion of 1-D X and/or Y to 2-D arrays.

pick(*args)
Call signature:

pick(mouseevent)

each child artist will fire a pick event if mouseevent is over the artist and the artist has picker set

pie(x, explode=None, labels=None, colors=None, autopct=None, pctdistance=0.6, shadow=False, labeldistance=1.1, startangle=None, radius=None)
Plot a pie chart.

Call signature:

pie(x, explode=None, labels=None, colors=\('b', 'g', 'r', 'c', 'm', 'y', 'k', 'w'\), autopct=None, pctdistance=0.6, shadow=False, labeldistance=1.1, startangle=None, radius=None)
Make a pie chart of array \( x \). The fractional area of each wedge is given by \( x / \text{sum}(x) \). If \( \text{sum}(x) \leq 1 \), then the values of \( x \) give the fractional area directly and the array will not be normalized. The wedges are plotted counterclockwise, by default starting from the x-axis.

Keyword arguments:
- \( \text{explode} \): [None | len(x) sequence] If not None, is a len(x) array which specifies the fraction of the radius with which to offset each wedge.
- \( \text{colors} \): [None | color sequence] A sequence of matplotlib color args through which the pie chart will cycle.
- \( \text{labels} \): [None | len(x) sequence of strings] A sequence of strings providing the labels for each wedge.
- \( \text{autopct} \): [None | format string | format function] 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 \( \text{fmt}\%\text{pct} \). If it is a function, it will be called.
- \( \text{pctdistance} \): scalar The ratio between the center of each pie slice and the start of the text generated by \( \text{autopct} \). Ignored if \( \text{autopct} \) is None; default is 0.6.
- \( \text{labeldistance} \): scalar The radial distance at which the pie labels are drawn.
- \( \text{shadow} \): [False | True] Draw a shadow beneath the pie.
- \( \text{startangle} \): [None | Offset angle] If not None, rotates the start of the pie chart by angle degrees counterclockwise from the x-axis.
- \( \text{radius} \): [None | scalar] The radius of the pie, if \( \text{radius} \) is None it will be set to 1.

The pie chart will probably look best if the figure and axes are square, or the Axes aspect is equal. e.g.:

```python
figure(figsize=(8, 8))
ax = axes([0.1, 0.1, 0.8, 0.8])
```

or:

```python
axes(aspect=1)
```

**Return value:** If \( \text{autopct} \) is None, return the tuple (\( \text{patches} \), \( \text{texts} \)):
- \( \text{patches} \) is a sequence of matplotlib.patches.Wedge instances
- \( \text{texts} \) is a list of the label matplotlib.text.Text instances.

If \( \text{autopct} \) is not None, return the tuple (\( \text{patches} \), \( \text{texts} \), \( \text{autotexts} \)), where \( \text{patches} \) and \( \text{texts} \) are as above, and \( \text{autotexts} \) is a list of Text instances for the numeric labels.

**plot(**args, **kwargs)**
Plot lines and/or markers to the Axes. args is a variable length argument, allowing for multiple \( x \), \( y \) pairs with an optional format string. For example, each of the following is legal:

```python
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
```

If \( x \) and/or \( y \) is 2-dimensional, then the corresponding columns will be plotted.
An arbitrary number of x, y, fmt groups can be specified, as in:

```python
a.plot(x1, y1, 'g^', x2, y2, 'g-')
```

Return value is a list of lines that were added.

By default, each line is assigned a different color specified by a ‘color cycle’. To change this behavior, you can edit the axes.color_cycle rcParam. Alternatively, you can use `set_default_color_cycle()`.

The following format string characters are accepted to control the line style or marker:

<table>
<thead>
<tr>
<th>character</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'-'</td>
<td>solid line style</td>
</tr>
<tr>
<td>'--'</td>
<td>dashed line style</td>
</tr>
<tr>
<td>'-.'</td>
<td>dash-dot line style</td>
</tr>
<tr>
<td>':'</td>
<td>dotted line style</td>
</tr>
<tr>
<td>'.'</td>
<td>point marker</td>
</tr>
<tr>
<td>'o'</td>
<td>circle marker</td>
</tr>
<tr>
<td>'v'</td>
<td>triangle_down marker</td>
</tr>
<tr>
<td>'^'</td>
<td>triangle_up marker</td>
</tr>
<tr>
<td>'&lt;'</td>
<td>triangle_left marker</td>
</tr>
<tr>
<td>'&gt;'</td>
<td>triangle_right marker</td>
</tr>
<tr>
<td>'1'</td>
<td>tri_down marker</td>
</tr>
<tr>
<td>'2'</td>
<td>tri_up marker</td>
</tr>
<tr>
<td>'3'</td>
<td>tri_left marker</td>
</tr>
<tr>
<td>'4'</td>
<td>tri_right marker</td>
</tr>
<tr>
<td>'s'</td>
<td>square marker</td>
</tr>
<tr>
<td>'p'</td>
<td>pentagon marker</td>
</tr>
<tr>
<td>'x'</td>
<td>star marker</td>
</tr>
<tr>
<td>'h'</td>
<td>hexagon1 marker</td>
</tr>
<tr>
<td>'H'</td>
<td>hexagon2 marker</td>
</tr>
<tr>
<td>'+'</td>
<td>plus marker</td>
</tr>
<tr>
<td>'x'</td>
<td>x marker</td>
</tr>
<tr>
<td>'D'</td>
<td>diamond marker</td>
</tr>
<tr>
<td>'d'</td>
<td>thin_diamond marker</td>
</tr>
<tr>
<td>'['</td>
<td>vline marker</td>
</tr>
<tr>
<td>'_'</td>
<td>hline marker</td>
</tr>
</tbody>
</table>

The following color abbreviations are supported:
In addition, you can specify colors in many weird and wonderful ways, including full names ('green'), hex strings ('#008000'), RGB or RGBA tuples ((0, 1, 0, 1)) or grayscale intensities as a string ('0.8'). Of these, the string specifications can be used in place of a fmt group, but the tuple forms can be used only as kwargs.

Line styles and colors are combined in a single format string, as in 'bo' for blue circles.

The kwargs can be used to set line properties (any property that has a set_* method). You can use this to set a line label (for auto legends), linewidth, antialising, marker face color, etc. Here is an 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')
axis([0, 4, 0, 10])
legend()
```

If you make multiple lines with one plot command, the kwargs apply to all those lines, e.g.:

```
plot(x1, y1, x2, y2, antialiased=False)
```

Neither line will be antialiased.

You do not need to use format strings, which are just abbreviations. All of the line properties can be controlled by keyword arguments. For example, you can set the color, marker, linestyle, and markerfacecolor with:

```
plot(x, y, color='green', linestyle='dashed', marker='o',
     markerfacecolor='blue', markersize=12).
```

See Line2D for details.

The kwargs are Line2D properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
</tbody>
</table>
### Table 53.17 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>([Path, Transform]</td>
</tr>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>data</td>
<td>2D array (rows are x, y) or two 1D arrays</td>
</tr>
<tr>
<td>drawstyle</td>
<td>['default'</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>['full'</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['-']</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>marker</td>
<td>unknown</td>
</tr>
<tr>
<td>markeredgecolor or mec</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markeredgewidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

**kwargs** `scalex` and `scaley`, if defined, are passed on to `autoscale_view()` to determine whether the x and y axes are autoscaled; the default is `True`.

**plot_date**(x, y, fmt='bo', tz=None, xdate=True, ydate=False, **kwargs)

Plot with data with dates.
Call signature:

```
plot_date(x, y, fmt='bo', tz=None, xdate=True, ydate=False, **kwargs)
```

Similar to the `plot()` command, except the x or y (or both) data is considered to be dates, and the axis is labeled accordingly.

x and/or y can be a sequence of dates represented as float days since 0001-01-01 UTC.

Keyword arguments:

- **fmt**: string The plot format string.
- **tz**: [None | timezone string | tzinfo instance] The time zone to use in labeling dates. If None, defaults to rc value.
- **xdate**: [True | False] If True, the x-axis will be labeled with dates.
- **ydate**: [False | True] If True, the y-axis will be labeled with dates.

Note if you are using custom date tickers and formatters, it may be necessary to set the formatters/locators after the call to `plot_date()` since `plot_date()` will set the default tick locator to `matplotlib.dates.AutoDateLocator` (if the tick locator is not already set to a `matplotlib.dates.DateLocator` instance) and the default tick formatter to `matplotlib.dates.AutoDateFormatter` (if the tick formatter is not already set to a `matplotlib.dates DateFormatter` instance).

Valid kwargs are `Line2D` properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>data</td>
<td>2D array (rows are x, y) or two 1D arrays</td>
</tr>
<tr>
<td>drawstyle</td>
<td>['default'</td>
</tr>
<tr>
<td>figure</td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>['full'</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with '%s' conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['-']</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
</tbody>
</table>
### Table 53.18 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>marker</td>
<td>unknown</td>
</tr>
<tr>
<td>markeredgecolor or mec</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markeredgewidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

See also:

- `dates` for helper functions
- `date2num()`, `num2date()` and `drange()` for help on creating the required floating point dates.

```python
psd(x, NFFT=256, Fs=2, Fc=0, detrend=mlab.detrend_none, window=mlab.window_hanning, noverlap=0, pad_to=None, sides='default', scale_by_freq=None, **kwargs)
```

Plot the power spectral density.

Call signature:

```python
psd(x, NFFT=256, Fs=2, Fc=0, detrend=mlab.detrend_none,
window=mlab.window_hanning, noverlap=0, pad_to=None,
sides='default', scale_by_freq=None, **kwargs)
```

The power spectral density 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 `|fft(i)|^2` of each segment `i` are averaged to compute `P_{xx}`, with a scaling to correct for power loss due to windowing. `Fs` is the sampling frequency.

Keyword arguments:
**NFFT**: integer  The number of data points used in each block for the FFT. Must be even; a power 2 is most efficient. The default value is 256. This should *NOT* be used to get zero padding, or the scaling of the result will be incorrect. Use *pad_to* for this instead.

**Fs**: scalar  The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, freqs, in cycles per time unit. The default value is 2.

**detrend**: callable  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 it is a function. The *pylab* module defines *detrend_none*(), *detrend_mean*(), and *detrend_linear*(), but you can use a custom function as well.

**window**: callable or ndarray  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. The default is *window_hanning*(). 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.

**pad_to**: integer  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 psd (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*

**sides**: ['default' | 'onesided' | 'twosided']  Specifies which sides of the PSD to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. ‘onesided’ forces the return of a one-sided PSD, while ‘twosided’ forces two-sided.

**scale_by_freq**: boolean  Specifies 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**: integer  The number of points of overlap between blocks. The default value is 0 (no overlap).

**Fc**: integer  The center frequency of *x* (defaults to 0), which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsamped to baseband.

Returns the tuple (*Pxx*, *freqs*).

For plotting, the power is plotted as $10 \log_{10}(P_{xx})$ for decibels, though $P_{xx}$ itself is returned.


**kwargs** control the *Line2D* properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
</tbody>
</table>
Table 53.19 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>axes</td>
<td>an <code>Axes</code> instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color or c</td>
<td>any <code>matplotlib</code> color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>data</td>
<td>2D array (rows are x, y) or two 1D arrays</td>
</tr>
<tr>
<td>drawstyle</td>
<td>['default'</td>
</tr>
<tr>
<td>figure</td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>['full'</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with '%s' conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['-'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>marker</td>
<td>unknown</td>
</tr>
<tr>
<td>markeredgcolor or mec</td>
<td>any <code>matplotlib</code> color</td>
</tr>
<tr>
<td>markeredgwidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any <code>matplotlib</code> color</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any <code>matplotlib</code> color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>transform</td>
<td>a <code>matplotlib.transforms.Transform</code> instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

Example:
**quiver(***args, **kw*)**

Plot a 2-D field of arrows.

call signatures:

- `quiver(U, V, **kw)`
- `quiver(U, V, C, **kw)`
- `quiver(X, Y, U, V, **kw)`
- `quiver(X, Y, U, V, C, **kw)`

Arguments:
- `X, Y`: The x and y coordinates of the arrow locations (default is tail of arrow; see `pivot` kwarg)
- `U, V`: Give the x and y components of the arrow vectors
- `C`: An optional array used to map colors to the arrows

All arguments may be 1-D or 2-D arrays or sequences. If `X` and `Y` are absent, they will be generated as a uniform grid. If `U` and `V` are 2-D arrays but `X` and `Y` are 1-D, and if `len(X)` and `len(Y)` match the column and row dimensions of `U`, then `X` and `Y` will be expanded with `numpy.meshgrid()`.

`U, V, C` may be masked arrays, but masked `X, Y` are not supported at present.

Keyword arguments:
- `units`: [‘width’ | ‘height’ | ‘dots’ | ‘inches’ | ‘x’ | ‘y’ | ‘xy’] Arrow units; the arrow dimensions except for length are in multiples of this unit.
• ‘width’ or ‘height’: the width or height of the axes
• ‘dots’ or ‘inches’: pixels or inches, based on the figure dpi
• ‘x’, ‘y’, or ‘xy’: X, Y, or sqrt(X^2+Y^2) 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 the window is resized; for ‘dots’ or ‘inches’, resizing does not change the arrows.

`angles`: [‘uv’ | ‘xy’ | array] With the default ‘uv’, the arrow aspect ratio is 1, so that if \( U*V = V * U \) the angle of the arrow on the plot is 45 degrees CCW from the x-axis. With ‘xy’, the arrow points from \((x,y)\) to \((x+u, y+v)\). Alternatively, arbitrary angles may be specified as an array of values in degrees, CCW from the x-axis.

`scale`: [ None | float ] Data units per arrow length unit, e.g., m/s per plot width; a smaller scale parameter makes the arrow longer. 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`: None, or any of the units options. For example, if `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’, then the vector will be half the width 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`: 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`: scalar Head width as multiple of shaft width, default is 3

`headlength`: scalar Head length as multiple of shaft width, default is 5

`headaxislength`: scalar Head length at shaft intersection, default is 4.5

`minshaft`: scalar Length below which arrow scales, in units of head length. Do not set this to less than 1, or small arrows will look terrible! Default is 1

`minlength`: scalar Minimum length as a multiple of shaft width; if an arrow length is less than this, plot a dot (hexagon) of this diameter instead. Default is 1.

`pivot`: [‘tail’ | ‘middle’ | ‘tip’] The part of the arrow that is at the grid point; the arrow rotates about this point, hence the name pivot.

`color`: [ color | color sequence ] This is a synonym for the PolyCollection facecolor kwarg. If \(C\) has been set, color has no effect.

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.

Linewidths and edgecolors can be used to customize the arrow outlines. Additional PolyCollection keyword arguments:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
</tbody>
</table>

Continued on next page
### Table 53.20 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>antialiased</td>
<td>Boolean or sequence of booleans</td>
</tr>
<tr>
<td>array</td>
<td>unknown</td>
</tr>
<tr>
<td>axes</td>
<td>an <code>Axes</code> instance</td>
</tr>
<tr>
<td>clim</td>
<td>a length 2 sequence of floats</td>
</tr>
<tr>
<td>clip_box</td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>cmap</td>
<td>a colormap or registered colormap name</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>colorbar</td>
<td>unknown</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>facecolor</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>figure</td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[ / \</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion</td>
</tr>
<tr>
<td>linestyle</td>
<td>['solid'</td>
</tr>
<tr>
<td>linewidth</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>norm</td>
<td>unknown</td>
</tr>
<tr>
<td>offset_position</td>
<td>unknown</td>
</tr>
<tr>
<td>offsets</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>paths</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None]float[boolean]call]</td>
</tr>
<tr>
<td>pickradius</td>
<td>unknown</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>urls</td>
<td>unknown</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

**quiverkey(**args, **kw)**

Add a key to a quiver plot.

Call signature:

quiverkey(Q, X, Y, U, label, **kw)
Arguments:

\(Q\): The Quiver instance returned by a call to quiver.
\(X, Y\): The location of the key; additional explanation follows.
\(U\): The length of the key

**label**: A string with the length and units of the key

Keyword arguments:

\(coordinates\) = [‘axes’ | ‘figure’ | ‘data’ | ‘inches’] 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.

\(color\): 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.

\(labelsep\): Distance in inches between the arrow and the label. Default is 0.1

\(labelcolor\): defaults to default Text color.

\(fontproperties\): A dictionary with keyword arguments accepted by the FontProperties initializer: family, style, variant, size, weight

Any additional keyword arguments are used to override vector properties taken from \(Q\).

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.

\(redraw_in_frame()\)

This method can only be used after an initial draw which caches the renderer. It is used to efficiently update Axes data (axis ticks, labels, etc are not updated)

\(relim()\)

Recompute the data limits based on current artists.

At present, Collection instances are not supported.

\(reset_position()\)

Make the original position the active position

\(scatter(x, y, s=20, c='b', marker='o', cmap=None, norm=None, vmin=None, vmax=None, alpha=None, linewidths=None, verts=None, **kwargs)\)

Make a scatter plot of \(x vs y\), where \(x\) and \(y\) are sequence like objects of the same lengths.

**Parameters**

\(x, y\) : array_like, shape (n, )

Input data

\(s\) : scalar or array_like, shape (n, ), optional, default: 20

size in points^2.

\(c\) : color or sequence of color, optional, default

\(c\) can be a single color format string, or a sequence of color specifications of length \(N\), or a sequence of \(N\) numbers to be mapped to colors using the cmap and norm specified via kwargs (see below). 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. \(c\) can be a 2-D array in which the rows are RGB or RGBA, however.
**marker**: MarkerStyle, optional, default: ‘o’
See markers for more information on the different styles of markers scatter supports.

**cmap**: Colormap, optional, default: None
A Colormap instance or registered name. cmap is only used if c is an array of floats. If None, defaults to rc image.cmap.

**norm**: Normalize, optional, default: None
A Normalize instance is used to scale luminance data to 0, 1. norm is only used if c is an array of floats. If None, use the default normalize().

**vmin, vmax**: scalar, optional, default: None
vmin and vmax are used in conjunction with norm to normalize luminance data. If either are None, the min and max of the color array is used. Note if you pass a norm instance, your settings for vmin and vmax will be ignored.

**alpha**: scalar, optional, default: None
The alpha blending value, between 0 (transparent) and 1 (opaque)

**linewidths**: scalar or array_like, optional, default: None
If None, defaults to (lines.linewidth,). Note that this is a tuple, and if you set the linewidths argument you must set it as a sequence of floats, as required by RegularPolyCollection.

**Returns paths**: PathCollection

**Other Parameters** kwargs: Collection properties

**Notes**

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.
semilogx(*args, **kwargs)

Make a plot with log scaling on the x axis.

Call signature:

```
semilogx(*args, **kwargs)
```

semilogx() supports all the keyword arguments of plot() and matplotlib.axes.Axes.set_xscale().

Notable keyword arguments:

- **basex**: scalar > 1 Base of the x logarithm
- **subsx**: [None | sequence] The location of the minor xticks; None defaults to autosubs, which depend on the number of decades in the plot; see set_xscale() for details.
- **nonposx**: [‘mask’ | ‘clip’] Non-positive values in x can be masked as invalid, or clipped to a very small positive number

The remaining valid kwargs are Line2D properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>data</td>
<td>2D array (rows are x, y) or two 1D arrays</td>
</tr>
<tr>
<td>drawstyle</td>
<td>['default'</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>['full'</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['-.'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>marker</td>
<td>unknown</td>
</tr>
<tr>
<td>markeredgewidth or mec</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markeredgewidth or mec</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

See also:

53.1. matplotlib.axes 795
loglog() For example code and figure

semilogy(*args, **kwargs)
Make a plot with log scaling on the y axis.
call signature:

semilogy(*args, **kwargs)

semilogy() supports all the keyword arguments of plot() and matplotlib.axes.Axes.set_yscale().

Notable keyword arguments:

- **basey**: scalar > 1 Base of the y logarithm
- **subsy**: [None | sequence] The location of the minor ticks; None defaults to autosubs, which depend on the number of decades in the plot; see set_yscale() for details.
- **nonposy**: ['mask' | 'clip'] Non-positive values in y can be masked as invalid, or clipped to a very small positive number

The remaining valid kwargs are Line2D properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>data</td>
<td>2D array (rows are x, y) or two 1D arrays</td>
</tr>
<tr>
<td>drawstyle</td>
<td>['default'</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>['full'</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['-']</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>marker</td>
<td>unknown</td>
</tr>
<tr>
<td>markeredgedcolor or mec</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markeredgewidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
</tbody>
</table>
Table 53.22 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

See also:

loglog() For example code and figure

set_adjustable(adjustable)

ACCEPSTS: ['box' | 'datalim' | 'box-forced']

set_anchor(anchor)

<table>
<thead>
<tr>
<th>value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘C’</td>
<td>Center</td>
</tr>
<tr>
<td>‘SW’</td>
<td>bottom left</td>
</tr>
<tr>
<td>‘S’</td>
<td>bottom</td>
</tr>
<tr>
<td>‘SE’</td>
<td>bottom right</td>
</tr>
<tr>
<td>‘E’</td>
<td>right</td>
</tr>
<tr>
<td>‘NE’</td>
<td>top right</td>
</tr>
<tr>
<td>‘N’</td>
<td>top</td>
</tr>
<tr>
<td>‘NW’</td>
<td>top left</td>
</tr>
<tr>
<td>‘W’</td>
<td>left</td>
</tr>
</tbody>
</table>

set_aspect(aspect, adjustable=None, anchor=None)

<table>
<thead>
<tr>
<th>value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘auto’</td>
<td>automatic; fill position rectangle with data</td>
</tr>
<tr>
<td>‘normal’</td>
<td>same as ‘auto’; deprecated</td>
</tr>
<tr>
<td>‘equal’</td>
<td>same scaling from data to plot units for x and y</td>
</tr>
<tr>
<td>num</td>
<td>a circle will be stretched such that the height is num times the width. aspect=1 is the same as aspect=’equal’.</td>
</tr>
</tbody>
</table>
### Adjustable

<table>
<thead>
<tr>
<th>value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'box'</td>
<td>change physical size of axes</td>
</tr>
<tr>
<td>'datalim'</td>
<td>change xlim or ylim</td>
</tr>
<tr>
<td>'box-forced'</td>
<td>same as 'box', but axes can be shared</td>
</tr>
</tbody>
</table>

'box' does not allow axes sharing, as this can cause unintended side effect. For cases when sharing axes is fine, use 'box-forced'.

### Anchor

<table>
<thead>
<tr>
<th>value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'C'</td>
<td>centered</td>
</tr>
<tr>
<td>'SW'</td>
<td>lower left corner</td>
</tr>
<tr>
<td>'S'</td>
<td>middle of bottom edge</td>
</tr>
<tr>
<td>'SE'</td>
<td>lower right corner</td>
</tr>
<tr>
<td>etc.</td>
<td></td>
</tr>
</tbody>
</table>

Deprecated since version 1.2: the option ‘normal’ for aspect is deprecated. Use ‘auto’ instead.

#### set_autoscale_on(b)
Set whether autoscaling is applied on plot commands

accepts: [ True | False ]

#### set_autoscalex_on(b)
Set whether autoscaling for the x-axis is applied on plot commands

accepts: [ True | False ]

#### set_autoscaley_on(b)
Set whether autoscaling for the y-axis is applied on plot commands

accepts: [ True | False ]

#### set_axes_locator(locator)
set_axes_locator

ACCEPT: a callable object which takes an axes instance and renderer and returns a bbox.

#### set_axisbgcolor(color)
set the axes background color

ACCEPTS: any matplotlib color - see colors()

#### set_axis_off()
turn off the axis

#### set_axis_on()
turn on the axis

#### set_axisbelow(b)
Set whether the axis ticks and gridlines are above or below most artists

ACCEPTS: [ True | False ]

#### set_color_cycle(clist)
Set the color cycle for any future plot commands on this Axes.

clist is a list of mpl color specifiers.
set_cursor_props(*args)
    Set the cursor property as:
    
    ax.set_cursor_props(linewidth, color)
    
    or:
    
    ax.set_cursor_props((linewidth, color))
    
    ACCEPTS: a (float, color) tuple

set_figure(fig)
    Set the class: Axes figure
    accepts a class: Figure instance

set_frame_on(b)
    Set whether the axes rectangle patch is drawn
    ACCEPTS: [ True | False ]

set_navigate(b)
    Set whether the axes responds to navigation toolbar commands
    ACCEPTS: [ True | False ]

set_navigate_mode(b)
    Set the navigation toolbar button status;
    
    Warning: this is not a user-API function.

set_position(pos, which='both')
    Set the axes position with:
    
    pos = [left, bottom, width, height]
    
    in relative 0,1 coords, or pos can be a Bbox
    
    There are two position variables: one which is ultimately used, but which may be modified by
    apply_aspect(), and a second which is the starting point for apply_aspect().

    Optional keyword arguments: which
    
    | value   | description          |
    |---------|----------------------|
    | 'active'| to change the first  |
    | 'original'| to change the second|
    | 'both'  | to change both       |

set_rasterization_zorder(z)
    Set zorder value below which artists will be rasterized. Set to None to disable rasterizing of artists below a particular zorder.

set_title(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'],
   'verticalalignment': 'baseline',
   'horizontalalignment': loc}
  ```

  - **loc** : {'center', 'left', 'right'}, str, optional
    Which title to set, defaults to 'center'

**Returns**

- **text** : Text
  The matplotlib text instance representing the title

**Other Parameters**

Other keyword arguments are text properties, see:

- `~matplotlib.text.Text` for a list of valid text properties.

**set_xbound(upper=None, lower=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 _autoscaleXOn attribute.

**set_xlabel(xlabel, fontdict=None, labelpad=None, **kwargs)**

Set the label for the xaxis.

- **xlabel** : string
- **labelpad** : scalar, optional, default: None
  spacing in points between the label and the x-axis

**Other Parameters**

- **kwargs** : Text properties

See also:

- `text` for information on how to override and the optional args work

**set_xlim(emit=True, auto=False, **kw)**

Call signature:

```
set_xlim(self, *args, **kw):
```

Set the data limits for the xaxis

Examples:

```
set_xlim((left, right))
set_xlim(left, right)
set_xlim(left=1) # right unchanged
set_xlim(right=1) # left unchanged
```

Keyword arguments:

- **left** : scalar
  The left xlim; `xmin`, the previous name, may still be used
The right $\texttt{xmax}$, the previous name, may still be used

$\texttt{emit}$: [ $\texttt{True}$ | $\texttt{False}$ ] Notify observers of limit change

$\texttt{auto}$: [ $\texttt{True}$ | $\texttt{False}$ | $\texttt{None}$ ] Turn $x$ autoscaling on ($\texttt{True}$), off ($\texttt{False}$; default), or leave un-changed ($\texttt{None}$)

Note, the left (formerly $\texttt{xmin}$) value may be greater than the right (formerly $\texttt{xmax}$). For example, suppose $x$ is years before present. Then one might use:

```python
set_ylim(5000, 0)
```

so 5000 years ago is on the left of the plot and the present is on the right.

Returns the current xlimits as a length 2 tuple

ACCEPTS: length 2 sequence of floats

$\texttt{set_xmargin}(m)$

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.

accepts: float in range 0 to 1

$\texttt{set_xscale}(value, **\texttt{kwargs})$

Call signature:

```python
set_xscale(value)
```

Set the scaling of the x-axis: ‘linear’ | ‘log’ | ‘symlog’

ACCEPTS: [‘linear’ | ‘log’ | ‘symlog’]

Different kwargs are accepted, depending on the scale: ‘linear’

‘log’

$\texttt{basex}/\texttt{basey}$: The base of the logarithm

$\texttt{nonposx}/\texttt{nonposy}$: [‘$\texttt{mask}$’ | ‘$\texttt{clip}$’ ] non-positive values in $x$ or $y$ can be masked as invalid, or clipped to a very small positive number

$\texttt{subsx}/\texttt{subsy}$: Where to place the subticks between each major tick. Should be a sequence of integers. 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.

‘symlog’

$\texttt{basex}/\texttt{basey}$: The base of the logarithm

$\texttt{linthreshx}/\texttt{linthreshy}$: The range (-$x$, $x$) within which the plot is linear (to avoid having the plot go to infinity around zero).

$\texttt{subsx}/\texttt{subsy}$: Where to place the subticks between each major tick. Should be a sequence of integers. 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.

$\texttt{linscalex}/\texttt{linscaley}$: This allows the linear range (-$\texttt{linthresh}$ to $\texttt{linthresh}$) 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 $\texttt{linscale} == 1.0$ (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.
**set_xticklabels**(labels, fontdict=None, minor=False, **kwargs)

Call signature:

```python
set_xticklabels(labels, fontdict=None, minor=False, **kwargs)
```

Set the xtick labels with list of strings `labels`. Return a list of axis text instances.

`kwargs` set the `Text` properties. Valid properties are:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an <code>Axes</code> instance</td>
</tr>
<tr>
<td>backgroundcolor</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>bbox</td>
<td>rectangle prop dict</td>
</tr>
<tr>
<td>clip_box</td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>family or fontfamily or fontname or name</td>
<td>[FONTNAME</td>
</tr>
<tr>
<td>figure</td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td>fontproperties or font_properties</td>
<td>a <code>matplotlib.font_manager.FontProperties</code> instance</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>horizontalalignment or ha</td>
<td>[‘center’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linespacing</td>
<td>float (multiple of font size)</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>multialignment</td>
<td>[‘left’</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>position</td>
<td>(x,y)</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>rotation</td>
<td>[ angle in degrees</td>
</tr>
<tr>
<td>rotation_mode</td>
<td>unknown</td>
</tr>
<tr>
<td>size or fontsize</td>
<td>[size in points</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>stretch or fontstretch</td>
<td>[a numeric value in range 0-1000</td>
</tr>
<tr>
<td>style or fontstyle</td>
<td>[‘normal’</td>
</tr>
<tr>
<td>text</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>transform</td>
<td><code>Transform</code> instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>variant or fontvariant</td>
<td>[‘normal’</td>
</tr>
<tr>
<td>verticalalignment or va or ma</td>
<td>[‘center’</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
</tbody>
</table>
### Table 53.23 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>x</strong></td>
<td>float</td>
</tr>
<tr>
<td><strong>y</strong></td>
<td>float</td>
</tr>
<tr>
<td><strong>zorder</strong></td>
<td>any number</td>
</tr>
</tbody>
</table>

ACCEPETS: sequence of strings

**set_xticks** *(ticks, minor=*

Set the x ticks with list of *ticks*

ACCEPETS: sequence of floats

**set_ybound** *(lower=*

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 _autoscaleYon attribute.

**set_ylabel** *(ylabel, fontdict=None, labelpad=None, **kwargs)*

Set the label for the yaxis

**Parameters**

- ylabel : string
  y label
- labelpad : scalar, optional, default: None
  spacing in points between the label and the x-axis

**Other Parameters**

- kwargs : Text properties

See also:

**text** for information on how override and the optional args work

**set_ylim** *(bottom=None, top=None, emit=True, auto=False, **kw)*

Call signature:

**set_ylim** *(self, *args, **kwargs):*

Set the data limits for the yaxis

Examples:

```python
set_ylim((bottom, top))
set_ylim(bottom, top)
set_ylim(bottom=1)  # top unchanged
set_ylim(top=1)     # bottom unchanged
```

Keyword arguments:

- **bottom**: scalar The bottom ylim; the previous name, *ymin*, may still be used
- **top**: scalar The top ylim; the previous name, *ymax*, may still be used
- **emit**: [ **True** | **False** ] Notify observers of limit change
- **auto**: [ **True** | **False** | **None** ] Turn y autoscaling on (**True**), off (**False**: default), or leave unchanged (**None**)
Note, the bottom (formerly ymin) value may be greater than the top (formerly ymax). For example, suppose y is depth in the ocean. Then one might use:

```python
set_ylim(5000, 0)
```

so 5000 m depth is at the bottom of the plot and the surface, 0 m, is at the top.

Returns the current ylims as a length 2 tuple

ACCEPTS: length 2 sequence of floats

**set_ymargin**(*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.

accepts: float in range 0 to 1

**set_yscale**(*value*, **kwargs*)

Call signature:

```python
set_yscale(value)
```

Set the scaling of the y-axis: ‘linear’ | ‘log’ | ‘symlog’

ACCEPTS: [‘linear’ | ‘log’ | ‘symlog’]

Different kwargs are accepted, depending on the scale: ‘linear’

‘log’

- **base**: The base of the logarithm
- **nonposx/nonposy**: [‘mask’ | ‘clip’] non-positive values in x or y can be masked as invalid, or clipped to a very small positive number
- **subx/suby**: Where to place the subticks between each major tick. Should be a sequence of integers. 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.

‘symlog’

- **base**: The base of the logarithm
- **linthreshx/linthreshy**: The range (-x, x) within which the plot is linear (to avoid having the plot go to infinity around zero).
- **subx/suby**: Where to place the subticks between each major tick. Should be a sequence of integers. 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.

- **linscalex/linscaley**: This allows the linear range (-linthresh to 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.

**set_yticklabels**(*labels*, **fontdict=None, minor=False, **kwargs*)

Call signature:
set_yticklabels(labels, fontdict=None, minor=False, **kwargs)

Set the y tick labels with list of strings labels. Return a list of Text instances.

kwargs set Text properties for the labels. Valid properties are

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>backgroundcolor</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>bbox</td>
<td>rectangle prop dict</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>family or fontfamily or fontname or name</td>
<td>[FONTNAME</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fontproperties or font_properties</td>
<td>a matplotlib.font_manager.FontProperties instance</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>horizontalalignment or ha</td>
<td>[‘center’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linespacing</td>
<td>float (multiple of font size)</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>multialignment</td>
<td>[‘left’</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>position</td>
<td>(x,y)</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>rotation</td>
<td>[ angle in degrees</td>
</tr>
<tr>
<td>rotation_mode</td>
<td>unknown</td>
</tr>
<tr>
<td>size or fontsize</td>
<td>[size in points</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>stretch or fontstretch</td>
<td>[a numeric value in range 0-1000</td>
</tr>
<tr>
<td>style or fontstyle</td>
<td>[ ‘normal’</td>
</tr>
<tr>
<td>text</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>variant or fontvariant</td>
<td>[‘normal’</td>
</tr>
<tr>
<td>verticalalignment or va or ma</td>
<td>[‘center’</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>weight or fontweight</td>
<td>[a numeric value in range 0-1000</td>
</tr>
<tr>
<td>x</td>
<td>float</td>
</tr>
</tbody>
</table>
Table 53.24 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>float</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

ACCEPTS: sequence of strings

```python
set_yticks(ticks, minor=False)
```
Set the y ticks with list of ticks

ACCEPTS: sequence of floats

Keyword arguments:

```python
minor: [False | True] Sets the minor ticks if True
```

```python
specgram(x, NFFT=256, Fs=2, Fc=0, detrend=<function detrend_none at 0x24627d0>, window=<function window_hanning at 0x2462578>, noverlap=128, cmap=None, xextent=None, pad_to=None, sides='default', scale_by_freq=None, **kwargs)
```

Plot a spectrogram.

Call signature:

```python
specgram(x, NFFT=256, Fs=2, Fc=0, detrend=mlab.detrend_none, window=mlab.window_hanning, noverlap=128, cmap=None, xextent=None, pad_to=None, sides='default', scale_by_freq=None, **kwargs)
```

Compute and plot a spectrogram of data in x. Data are split into NFFT length segments and the PSD 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 in decibels as a colormap (using imshow).

Keyword arguments:

```python
NFFT: integer The number of data points used in each block for the FFT. Must be even; a power 2 is most efficient. The default value is 256. This should NOT be used to get zero padding, or the scaling of the result will be incorrect. Use pad_to for this instead.
```

```python
Fs: scalar The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, freqs, in cycles per time unit. The default value is 2.
```

```python
detrend: callable 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 `pylab` module defines detrend_none(), detrend_mean(), and detrend_linear(), but you can use a custom function as well.
```

```python
window: callable or ndarray A function or a vector of length NFFT. To create window vectors see window_hanning(), window_none(), numpy.blackman(), numpy.hanning(), numpy.bartlett(), scipy.signal(). scipy.signal.get_window(), etc. The default is window_hanning(). 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.
```

```python
pad_to: integer 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 psd (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
sides: [ ‘default’ | ‘onesided’ | ‘twosided’ ] Specifies which sides of the PSD to return. De-
default gives the default behavior, which returns one-sided for real data and both for complex
data. ‘onesided’ forces the return of a one-sided PSD, while ‘twosided’ forces two-sided.
scale_by_freq: boolean Specifies 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: integer The number of points of overlap between blocks. The default value is 128.
Fc: integer The center frequency of x (defaults to 0), which offsets the y extents of the plot
to reflect the frequency range used when a signal is acquired and then filtered and down-
sampled to baseband.
cmap: A matplotlib.colors.Colormap instance; if None, use default determined by rc
xextent: The image extent along the x-axis. xextent = (xmin,xmax) The default is
(0,max(bins)), where bins is the return value from specgram()
kwarg:
  Additional kwarg is passed on to imshow which makes the specgram image
Return value is (Pxx, freqs, bins, im):
  **bins** are the time points the spectrogram is calculated over
  **freqs** is an array of frequencies
  **Pxx** is an array of shape (len(times), len(freqs)) of power
  **im** is a AxesImage instance

Note: If x is real (i.e. non-complex), only the positive spectrum is shown. If x is complex, both
positive and negative parts of the spectrum are shown. This can be overridden using the sides
keyword argument.

Also note that while the plot is in dB, the Pxx array returned is linear in power.

Example:
spy(Z, precision=0, marker=None, markersize=None, aspect='equal', **kwargs)

Plot the sparsity pattern on a 2-D array.

Call signature:

spy(Z, precision=0, marker=None, markersize=None, aspect='equal', **kwargs)

spy(Z) plots the sparsity pattern of the 2-D array Z.

If `precision` is 0, any non-zero value will be plotted; else, values of $|Z| > precision$ will be plotted.

For `scipy.sparse.spmatrix` instances, there is a special case: if `precision` is ‘present’, any value present in the array will be plotted, even if it is identically zero.

The array will be plotted as it would be printed, with the first index (row) increasing down and the second index (column) increasing to the right.

By default aspect is ‘equal’, so that each array element occupies a square space; set the aspect kwarg to ‘auto’ to allow the plot to fill the plot box, or to any scalar number to specify the aspect ratio of an array element directly.

Two plotting styles are available: image or marker. Both are available for full arrays, but only the marker style works for `scipy.sparse.spmatrix` instances.

If `marker` and `markersize` are `None`, an image will be returned and any remaining kwargs are passed to `imshow()`; else, a `Line2D` object will be returned with the value of marker determining
the marker type, and any remaining kwargs passed to the plot() method.

If marker and markersize are None, useful kwargs include:
  • cmap
  • alpha

See also:

imshow() For image options.

For controlling colors, e.g., cyan background and red marks, use:

cmap = mcolors.ListedColormap(['c', 'r'])

If marker or markersize is not None, useful kwargs include:
  • marker
  • markersize
  • color

Useful values for marker include:
  • ‘s’ square (default)
  • ‘o’ circle
  • ‘.’ point
  • ‘,’ pixel

See also:

plot() For plotting options

stackplot(x, *args, **kwargs)

Draws a stacked area plot.

x : 1d array of dimension N
y : [2d array of dimension MxN, OR any number 1d arrays each of 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

Keyword arguments:

baseline [['zero', 'sym', 'wiggle', 'weighted_wiggle']] Method used to calculate the baseline.
  ‘zero’ is just a simple stacked plot. ‘sym’ is 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://www.leebyron.com/else/streamgraph/

colors [A list or tuple of colors. These will be cycled through and] used to colour the stacked areas. All other keyword arguments are passed to fill_between()

Returns r : A list of PolyCollection, one for each element in the stacked area plot.

Note that Legend does not support PolyCollection objects. To create a legend on a stackplot, use a proxy artist: http://matplotlib.org/users/legend_guide.html#using-proxy-artist

start_pan(x, y, button)

Called when a pan operation has started.

x, y are the mouse coordinates in display coords. button is the mouse button number:
stem(*args, **kwargs)
Create a stem plot.

Call signatures:

stem(y, linefmt='b-', markerfmt='bo', basefmt='r-')
stem(x, y, linefmt='b-', markerfmt='bo', basefmt='r-')

A stem plot plots vertical lines (using linefmt) at each x location from the baseline to y, and places a marker there using markerfmt. A horizontal line at 0 is is plotted using basefmt.

If no x values are provided, the default is (0, 1, ..., len(y) - 1)

Return value is a tuple (markerline, stemlines, baseline).

See also:
This document for details.

Example:
step(x, y, *args, **kwargs)
Make a step plot.

Call signature:

```
step(x, y, *args, **kwargs)
```

Additional keyword args to step() are the same as those for plot().

x and y must be 1-D sequences, and it is assumed, but not checked, that x is uniformly increasing.

Keyword arguments:

- **where**: ['pre' | 'post' | 'mid'] If 'pre', the interval from x[i] to x[i+1] has level y[i+1]
  - If 'post', that interval has level y[i]
  - If 'mid', the jumps in y occur half-way between the x-values.

streamplot(x, y, u, v, density=1, linewidth=None, color=None, cmap=None, norm=None, arrowsize=1, arrowstyle='->', minlength=0.1, transform=None)
Draws streamlines of a vector flow.

- x, y [1d arrays] an evenly spaced grid.
- u, v [2d arrays] x and y-velocities. Number of rows should match length of y, and the number of columns should match x.
- **density**: [float or 2-tuple] Controls the closeness of streamlines. When density = 1, the domain is divided into a 25x25 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 [density_x, density_y].
- **linewidth**: [numeric or 2d array] vary linewidth when given a 2d array with the same shape as velocities.
- **color**: [matplotlib color code, or 2d array] Streamline color. When given an array with the same shape as velocities, color values are converted to colors using cmap.
- **cmap**: [Colormap] Colormap used to plot streamlines and arrows. Only necessary when using an array input for color.
- **norm**: [Normalize] Normalize object used to scale luminance data to 0, 1. If None, stretch (min, max) to (0, 1). Only necessary when color is an array.
- **arrowsize**: [float] Factor scale arrow size.
- **minlength**: [float] Minimum length of streamline in axes coordinates.

Returns:

- **stream_container**: [StreamplotSet] Container object with attributes
  - lines: matplotlib.collections.LineCollection of streamlines
  - arrows: collection of matplotlib.patches.FancyArrowPatch objects representing 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.

table(**kwargs)
Add a table to the current axes.

Call signature:
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):

Returns a matplotlib.table.Table instance. For finer grained control over tables, use the Table class and add it to the axes with add_table().

Thanks to John Gill for providing the class and table.

text(x, y, s, fontdict=None, withdash=False, **kwargs)

Add text to the axes.

Add text in string s to axis at location x, y, data coordinates.

Parameters
s : string
text
x, y : scalars
data coordinates
fontdict : dictionary, optional, default: None
A dictionary to override the default text properties. If fontdict is None, the defaults are determined by your rc parameters.
withdash : boolean, optional, default: False
Creates a TextWithDash instance instead of a Text instance.
Other Parameters `kwargs`:

- **Text properties.**
- Other miscellaneous text parameters.

### Examples

Individual keyword arguments can be used to override any given parameter:

```python
>>> 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:

```python
>>> 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:

```python
>>> text(x, y, s, bbox=dict(facecolor='red', alpha=0.5))
```

#### tick_params

- **axis** = ['x' | 'y' | 'both'] Axis on which to operate; default is ‘both’.
- **reset** = [True | False] If True, set all parameters to defaults before processing other keyword arguments. Default is False.
- **which** = ['major' | 'minor' | 'both'] Default is ‘major’; apply arguments to which ticks.
- **direction** = ['in' | 'out' | 'inout'] Puts ticks inside the axes, outside the axes, or both.
- **length** = Tick length in points.
- **width** = Tick width in points.
- **color** = Tick color; accepts any mpl color spec.
- **pad** = Distance in points between tick and label.
- **labelsize** = Tick label font size in points or as a string (e.g., ‘large’).
- **labelcolor** = Tick label color; mpl color spec.
- **colors** = Changes the tick color and the label color to the same value: mpl color spec.
- **zorder** = Tick and label zorder.
- **bottom, top, left, right** = [bool | ‘on’ | ‘off’] controls whether to draw the respective ticks.
- **labelbottom, labeltop, labelleft, labelright** = Boolean or [‘on’ | ‘off’], controls whether to draw the respective tick labels.

Example:

```python
taxx.tick_params(direction='out', length=6, width=2, colors='r')
```

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.
**ticklabel_format(**kwargs\)**

Change the ScalarFormatter used by default for linear axes.

Optional keyword arguments:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>style</strong></td>
<td>[ 'sci' (or 'scientific')</td>
</tr>
<tr>
<td><strong>scilimits</strong></td>
<td>(m, n), pair of integers; if <strong>style</strong> is 'sci', scientific notation will be used for numbers outside the range $10^m$ to $10^n$. Use (0,0) to include all numbers.</td>
</tr>
<tr>
<td><strong>useOffset</strong></td>
<td>[ True</td>
</tr>
<tr>
<td><strong>axis</strong></td>
<td>[ 'x'</td>
</tr>
</tbody>
</table>

Only the major ticks are affected. If the method is called when the **ScalarFormatter** is not the **Formatter** being used, an **AttributeError** will be raised.

**tricontour(***args, **kwargs\)**

Draw contours on an unstructured triangular grid. **tricontour()** and **tricontourf()** draw contour lines and filled contours, respectively. Except as noted, function signatures and return values are the same for both versions.

The triangulation can be specified in one of two ways; either:

**tricontour(triangulation, ...)**

where triangulation is a **matplotlib.tri.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 **Triangulation** for an explanation of these possibilities.

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, N)**
contour $N$ automatically-chosen levels.

```python
trcountour(..., Z, V)
```
draw contour lines at the values specified in sequence $V$

```python
trcountourf(..., Z, V)
```
fill the (len($V$)-1) regions between the values in $V$

```python
trcountour(Z, **kwargs)
```
Use keyword args to control colors, linewidth, origin, cmap ... see below for more details.

$C = t\text{rcountour}(\ldots)$ returns a TriContourSet object.

Optional keyword arguments:

- `colors`: [None | string | (mpl_colors)] If None, the colormap specified by cmap will be used.
  - If a string, like ‘r’ or ‘red’, all levels will be plotted in this color.
  - If a tuple of matplotlib color args (string, float, rgb, etc), different levels will be plotted in different colors in the order specified.
- `alpha`: float The alpha blending value
- `cmap`: [None | Colormap] A cm Colormap instance or None. If cmap is None and colors is None, a default Colormap is used.
- `norm`: [None | Normalize] A matplotlib.colors.Normalize instance for scaling data values to colors. If norm is None and colors is None, the default linear scaling is used.
- `levels`: [level0, level1, ..., leveln] A list of floating point numbers indicating the level curves to draw; eg to draw just the zero contour pass levels=[0]
- `origin`: [None | ‘upper’ | ‘lower’ | ‘image’] If None, the first value of $Z$ will correspond to the lower left corner, location (0,0). If ‘image’, the rc value for image.origin will be used.
  This keyword is not active if $X$ and $Y$ are specified in the call to contour.
- `extent`: [None | (x0,x1,y0,y1)]
  - If origin is not None, then extent is interpreted as in matplotlib.pyplot.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 keyword is not active if $X$ and $Y$ are specified in the call to contour.
- `locator`: [None | ticker.Locator subclass] If locator is None, the default MaxNLocator is used. The locator is used to determine the contour levels if they are not given explicitly via the $V$ argument.
extend: ['neither' | 'both' | 'min' | 'max'] Unless this is 'neither', contour levels are automatically added to one or both ends of the range so that all data are included. These added ranges are then mapped to the special colormap values which default to the ends of the colormap range, but can be set via matplotlib.colors.Colormap.set_under() and matplotlib.colors.Colormap.set_over() methods.

xunits, yunits: [None | registered units] Override axis units by specifying an instance of a matplotlib.units.ConversionInterface.

tricontour-only keyword arguments:

linewdths: [None | number | tuple of numbers] If linewdths is None, the default width in lines.linewidth in matplotlibrc is used.

If a number, all levels will be plotted with this linewidth.

If a tuple, different levels will be plotted with different linewidths in the order specified.

linestyles: [None | 'solid' | 'dashed' | 'dashdot' | 'dotted'] If linestyles is None, the 'solid' is used.

linewdths 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.

If contour is using a monochrome colormap and the contour level is less than 0, then the linestyle specified in contour.negative_linestyle in matplotlibrc will be used.

tricontourf-only keyword arguments:

antialiased: [True | False] enable antialiasing

nchunk: [0 | integer] If 0, no subdivision of the domain. Specify a positive integer to divide the domain into subdomains of roughly nchunk by nchunk points. This may never actually be advantageous, so this option may be removed. Chunking introduces artifacts at the chunk boundaries unless antialiased is False.

Note: tricontourf fills intervals that are closed at the top; that is, for boundaries z1 and z2, the filled region is:

z1 < z <= z2

There is one exception: if the lowest boundary coincides with the minimum value of the z array, then that minimum value will be included in the lowest interval.

Examples:
**tricontourf**(*args, **kwargs)

Draw contours on an unstructured triangular grid. **tricontour()** and **tricontourf()** draw contour lines and filled contours, respectively. Except as noted, function signatures and return values are the same for both versions.

The triangulation can be specified in one of two ways; either:

**tricontour**(triangulation, ...)

where triangulation is a :class:`matplotlib.tri.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 :class:`Triangulation` for a explanation of these possibilities.

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.

\[
\text{tricontour}(\ldots, Z, N)
\]

contour \( N \) automatically-chosen levels.

\[
\text{tricontour}(\ldots, Z, V)
\]
draw contour lines at the values specified in sequence \( V \)

\[
\text{tricontourf}(\ldots, Z, V)
\]
fill the \((\text{len}(V)-1)\) regions between the values in \( V \)

\[
\text{tricontour}(Z, \text{**kwargs})
\]
Use keyword args to control colors, linewidth, origin, cmap ... see below for more details.

\[
C = \text{tricontour}(\ldots) \text{ returns a TriContourSet object.}
\]

Optional keyword arguments:

- **colors**: [None | string | (mpl_colors)] If None, the colormap specified by cmap will be used.
  If a string, like ‘r’ or ‘red’, all levels will be plotted in this color.
  If a tuple of matplotlib color args (string, float, rgb, etc),
  different levels will be plotted in different colors in the order specified.

- **alpha**: float The alpha blending value

- **cmap**: [None | Colormap] A cm Colormap instance or None. If cmap is None and colors is None, a default Colormap is used.

- **norm**: [None | Normalize] A matplotlib.colors.Normalize instance for scaling data values to colors. If norm is None and colors is None, the default linear scaling is used.

- **levels**: [level0, level1, ..., leveln] A list of floating point numbers indicating the level curves to draw; eg to draw just the zero contour pass \( \text{levels}=[0] \)

- **origin**: [None | ‘upper’ | ‘lower’ | ‘image’] If None, the first value of \( Z \) will correspond to the lower left corner, location \((0,0)\). If ‘image’, the rc value for image.origin will be used.
  This keyword is not active if \( X \) and \( Y \) are specified in the call to contour.

- **extent**: [None | (x0,x1,y0,y1)]
  If origin is not None, then extent is interpreted as in matplotlib.pyplot.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 keyword is not active if \( X \) and \( Y \) are specified in the call to contour.
locator: [ None | ticker.Locator subclass ] If locator is None, the default MaxNLocator is used. The locator is used to determine the contour levels if they are not given explicitly via the V argument.

extend: [ ‘neither’ | ‘both’ | ‘min’ | ‘max’ ] Unless this is ‘neither’, contour levels are automatically added to one or both ends of the range so that all data are included. These added ranges are then mapped to the special colormap values which default to the ends of the colormap range, but can be set via matplotlib.colors.Colormap.set_under() and matplotlib.colors.Colormap.set_over() methods.

xunits, yunits: [ None | registered units ] Override axis units by specifying an instance of a matplotlib.units.ConversionInterface.

tricontour-only keyword arguments:

linewdiths: [ None | number | tuple of numbers ] If linewidths is None, the default width in lines.linewidth in matplotlibrc is used.

If a number, all levels will be plotted with this linewidth.

If a tuple, different levels will be plotted with different linewidths in the order specified.

linestyles: [ None | ‘solid’ | ‘dashed’ | ‘dashdot’ | ‘dotted’ ] If linestyles is None, the ‘solid’ is used.

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.

If contour is using a monochrome colormap and the contour level is less than 0, then the linestyle specified in contour.negative_linestyle in matplotlibrc will be used.

tricontourf-only keyword arguments:

antialiased: [ True | False ] enable antialiasing

ncunk: [ 0 | integer ] If 0, no subdivision of the domain. Specify a positive integer to divide the domain into subdomains of roughly nchunk by nchunk points. This may never actually be advantageous, so this option may be removed. Chunking introduces artifacts at the chunk boundaries unless antialiased is False.

Note: tricontourf fills intervals that are closed at the top; that is, for boundaries z1 and z2, the filled region is:

z1 < z <= z2

There is one exception: if the lowest boundary coincides with the minimum value of the z array, then that minimum value will be included in the lowest interval.

Examples:
Contour plot of Delaunay triangulation
**tripcolor(***args, **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 `matplotlib.tri.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`. 

---

Chapter 53. axes
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. shading of ‘faceted’ is deprecated; please use edgecolors instead.

The remaining kwargs are the same as for pcolor().

Example:

```
tripcolor of Delaunay triangulation, flat shading
```

![Image of tripcolor of Delaunay triangulation, flat shading](image_url)
Matplotlib, Release 1.3.0

Tripcolor of Delaunay triangulation, gouraud shading

Tripcolor of user-specified triangulation

Chapter 53. axes
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 matplotlib.tri.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 kwars are the same as for plot().

Example:
**twinx()**

Call signature:

```python
ax = twinx()
```

create a twin of Axes for generating a plot with a share x-axis but independent y axis. The y-axis of self will have ticks on left and the returned axes will have ticks on the right.

**Note:** For those who are ‘picking’ artists while using twinx, pick events are only called for the artists in the top-most axes.

---

**twiny()**

Call signature:

```python
ax = twiny()
```

create a twin of Axes for generating a plot with a shared y-axis but independent x axis. The x-axis of self will have ticks on bottom and the returned axes will have ticks on the top.

**Note:** For those who are ‘picking’ artists while using twiny, pick events are only called for the artists in the top-most axes.
**update_datalim**(*xys*, *updatex=True, updatey=True*)
Update the data lim bbox with seq of xy tups or equiv. 2-D array

**update_datalim_bounds**(*bounds*)
Update the data lim to include the given Bbox bounds

**update_datalim_numerix**(*x, y*)
Update the data lim bbox with seq of xy tups

**vlines**(*x, ymin, ymax, colors='k', linestyles='solid', label='', **kwargs*)
Plot vertical lines.

Plot vertical lines at each *x* from *ymin* to *ymax*.

**Parameters**
- **x**: scalar or 1D array_like
  x-indexes where to plot the lines.
- **xmin, xmax**: scalar or 1D array_like
  Respective beginning and end of each line. If scalars are provided, all lines will have same length.
- **colors**: array_like of colors, optional, default: ‘k’
- **linestyles**: ['solid' | 'dashed' | 'dashdot' | 'dotted'], optional
- **label**: string, optional, default: ‘’

**Returns**
- **lines**: LineCollection

**Other Parameters**
- **kwargs**: LineCollection properties.

**See also:**
- **hlines** horizontal lines

**Examples**

![Vertical lines demo](image_1)

**xaxis_date**(*tz=None*)
Sets up x-axis ticks and labels that treat the x data as dates.

*tz* is a timezone string or tzinfo instance. Defaults to rc value.
**xaxis_inverted()**

Returns `True` if the x-axis is inverted.

**xcorr(x, y, normed=True, detrend=<function detrend_none at 0x24627d0>, usevlines=True, maxlags=10, **kwargs)**

Plot the cross correlation between `x` and `y`.

Call signature:

```
xcorr(self, x, y, normed=True, detrend=mlab.detrend_none,
       usevlines=True, maxlags=10, **kwargs)
```

If `normed = True`, normalize the data by the cross correlation at 0-th lag. `x` and `y` are detrended by the `detrend` callable (default no normalization). `x` and `y` must be equal length.

Data are plotted as `plot(lags, c, **kwargs)`

Return value is a tuple `(lags, c, **kwargs)`

- `lags` are a length `2*maxlags+1` lag vector
- `c` is the `2*maxlags+1` auto correlation vector
- `line` is a `Line2D` instance returned by `plot()`.

The default `linestyle` is `None` and the default `marker` is ‘o’, though these can be overridden with keyword args. The cross correlation is performed with `numpy.correlate()` with `mode = 2`.

If `usevlines` is `True`:

- `vlines()` rather than `plot()` is used to draw vertical lines from the origin to the xcorr. Otherwise the plotstyle is determined by the kwargs, which are `Line2D` properties.

The return value is a tuple `(lags, c, linecol, b)` where `linecol` is the `matplotlib.collections.LineCollection` instance and `b` is the x-axis.

`maxlags` is a positive integer detailing the number of lags to show. The default value of `None` will return all `(2*len(x)-1)` lags.

**Example:**

`xcorr()` is top graph, and `acorr()` is bottom graph.
**yaxis_date**(*tz=None*)

Sets up y-axis ticks and labels that treat the y data as dates.

*tz* is a timezone string or *tzinfo* instance. Defaults to rc value.

**yaxis_inverted()**

Returns True if the y-axis is inverted.

`matplotlib.axes.Subplot`

alias of *AxesSubplot*

class `matplotlib.axes.SubplotBase(fig, *args, **kwargs)`

Base class for subplots, which are *Axes* instances with additional methods to facilitate generating and manipulating a set of *Axes* within a figure.

*fig* is a `matplotlib.figure.Figure` instance.

*args* is the tuple (*numRows*, *numCols*, *plotNum*), where the array of subplots in the figure has dimensions *numRows*, *numCols*, and where *plotNum* is the number of the subplot being created. *plotNum* starts at 1 in the upper left corner and increases to the right.

If *numRows* <= *numCols* <= *plotNum* < 10, *args* can be the decimal integer *numRows* * 100 + *numCols* * 10 + *plotNum*.

**change_geometry**(numrows, numcols, num)

change subplot geometry, e.g., from 1,1,1 to 2,2,3
getGeometry()
get the subplot geometry, eg 2,2,3

getSubplotspec()
get the SubplotSpec instance associated with the subplot

is_first_col()

is_first_row()

is_last_col()

is_last_row()

label_outer()
set the visible property on ticklabels so xticklabels are visible only if the subplot is in the last row and yticklabels are visible only if the subplot is in the first column

setSubplotspec(subplotspec)
set the SubplotSpec instance associated with the subplot

update_params()
update the subplot position from fig.subplotpars

matplotlib.axes.subplot_class_factory(axces_class=\nNone)
54.1 `matplotlib.axis`

Classes for the ticks and x and y axis

```python
class matplotlib.axis.Axis(axes, pickradius=15):
    Bases: matplotlib.artist.Artist
```

Public attributes
- `axes.transData` - transform data coords to display coords
- `axes.transAxes` - transform axis coords to display coords
- `labelpad` - number of points between the axis and its label

Init the axis with the parent Axes instance

```python
OFFSETTEXTPAD = 3
```

```python
axis_date(tz=None)
    Sets up x-axis ticks and labels that treat the x data as dates. `tz` is a `tzinfo` instance or a timezone string. This timezone is used to create date labels.
```

```python
cla()
    clear the current axis
```

```python
convert_units(x)
```

```python
draw(artist, renderer, *args, **kwargs)
    Draw the axis lines, grid lines, tick lines and labels
```

```python
get_children()
```

```python
get_data_interval()
    return the Interval instance for this axis data limits
```

```python
get_gridlines()
    Return the grid lines as a list of Line2D instance
```

```python
get_label()
    Return the axis label as a Text instance
```
get_label_text()
    Get the text of the label

get_major_formatter()
    Get the formatter of the major ticker

get_major_locator()
    Get the locator of the major ticker

get_major_ticks(numticks=None)
    get the tick instances; grow as necessary

get_major_ticklabels()
    Return a list of Text instances for the major ticklabels

get_major_ticklines()
    Return the major tick lines as a list of Line2D instances

get_major_ticklocs()
    Get the major tick locations in data coordinates as a numpy array

get_minor_formatter()
    Get the formatter of the minor ticker

get_minor_locator()
    Get the locator of the minor ticker

get_minor_ticks(numticks=None)
    get the minor tick instances; grow as necessary

get_minor_ticklabels()
    Return a list of Text instances for the minor ticklabels

get_minor_ticklines()
    Return the minor tick lines as a list of Line2D instances

get_minor_ticklocs()
    Get the minor tick locations in data coordinates as a numpy array

get_offset_text()
    Return the axis offsetText as a Text instance

get_pickradius()
    Return the depth of the axis used by the picker

get_scale()

get_smart_bounds()
    get whether the axis has smart bounds

get_ticklabel_extents(renderer)
    Get the extents of the tick labels on either side of the axes.

get_ticklabels(minor=False)
    Return a list of Text instances for ticklabels
get_ticklines(minor=False)
    Return the tick lines as a list of Line2D instances

gGet_ticklocs(minor=False)
    Get the tick locations in data coordinates as a numpy array

get_tightbbox(renderer)
    Return a bounding box that encloses the axis. It only accounts tick labels, axis label, and
    offsetText.

get_transform()

gGet_units()
    return the units for axis

get_view_interval()
    return the Interval instance for this axis view limits

grid(b=None, which='major', **kwargs)
    Set the axis grid on or off; b is a boolean. Use which = ‘major’ | ‘minor’ | ‘both’ to set the grid
    for major or minor ticks.
    If b is None and len(kwargs)==0, toggle the grid state. If kwargs are supplied, it is assumed
    you want the grid on and b will be set to True.
    
    kwargs are used to set the line properties of the grids, eg,
    xax.grid(color='r', linestyle='-', linewidth=2)

have_units()

iter_ticks()
    Iterate through all of the major and minor ticks.

limit_range_for_scale(vmin, vmax)

pan(numsteps)
    Pan numsteps (can be positive or negative)

reset_ticks()

set_clip_path(clippath, transform=None)

set_data_interval()
    set the axis data limits

set_default_intervals()
    set the default limits for the axis data and view interval if they are not mutated

set_label_coords(x, y, transform=None)
    Set the coordinates of the label. By default, the x coordinate of the y label is determined by the
tick label bounding boxes, but this can lead to poor alignment of multiple ylabels if there are multiple axes. Ditto for the y coordinate of the x label.

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 (left,bottom), (0.5, 0.5) is middle, etc

`set_label_text(label, fontdict=None, **kwargs)`
Sets the text value of the axis label

ACCEPTS: A string value for the label

`set_major_formatter(formatter)`
Set the formatter of the major ticker

ACCEPTS: A `Formatter` instance

`set_major_locator(locator)`
Set the locator of the major ticker

ACCEPTS: a `Locator` instance

`set_minor_formatter(formatter)`
Set the formatter of the minor ticker

ACCEPTS: A `Formatter` instance

`set_minor_locator(locator)`
Set the locator of the minor ticker

ACCEPTS: a `Locator` instance

`set_pickradius(pickradius)`
Set the depth of the axis used by the picker

ACCEPTS: a distance in points

`set_scale(*args, **kwargs)`
Deprecated since version 1.3: The `set_scale` function was deprecated in version 1.3.

This should be a private function (moved to `_set_scale`)

`set_smart_bounds(value)`
set the axis to have smart bounds

`set_tick_params(which='major', reset=False, **kw)`
Set appearance parameters for ticks and ticklabels.

For documentation of keyword arguments, see `matplotlib.axes.Axes.tick_params()`.

`set_ticklabels(ticklabels, *args, **kwargs)`
Set the text values of the tick labels. Return a list of Text instances. Use `kwarg minor=True` to select minor ticks. All other kwargs are used to update the text object properties. As for `get_ticklabels`, label1 (left or bottom) is affected for a given tick only if its label1On attribute is True, and similarly for label2. The list of returned label text objects consists of all such label1 objects followed by all such label2 objects.
The input *ticklabels* is assumed to match the set of tick locations, regardless of the state of label1On and label2On.

**ACCEPTS:** sequence of strings

**set_ticks**(ticks, minor=False)
Set the locations of the tick marks from sequence ticks

**ACCEPTS:** sequence of floats

**set_units**(u)
set the units for axis

**ACCEPTS:** a units tag

**set_view_interval**(vmin, vmax, ignore=False)

**update_units**(data)
introspect *data* for units converter and update the axis.converter instance if necessary. Return *True* if *data* is registered for unit conversion.

**zoom**(direction)
Zoom in/out on axis; if *direction* is >0 zoom in, else zoom out

**class matplotlib.axis.Tick**(axes, loc, label, 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)

**Bases:** matplotlib.artist.Artist

Abstract base class for the axis ticks, grid lines and labels

1 refers to the bottom of the plot for xticks and the left for yticks 2 refers to the top of the plot for xticks and the right for yticks

Publicly accessible attributes:

- **tick1line** a Line2D instance
- **tick2line** a Line2D instance
- **gridline** a Line2D instance
- **label1** a Text instance
- **label2** a Text instance
- **gridOn** a boolean which determines whether to draw the tickline
- **tick1On** a boolean which determines whether to draw the 1st tickline
- **tick2On** a boolean which determines whether to draw the 2nd tickline
- **label1On** a boolean which determines whether to draw tick label
- **label2On** a boolean which determines whether to draw 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

**apply_tickdir**(tickdir)
Calculate self._pad and self._tickmarkers

**contains**(mouseevent)
Test whether the mouse event occurred in the Tick marks.
This function always returns false. It is more useful to test if the axis as a whole contains the mouse rather than the set of tick marks.

\texttt{draw}(\texttt{artist}, \texttt{renderer}, \texttt{*args}, \texttt{**kwargs})

\texttt{get\_children}()

\texttt{get\_loc}()
   Return the tick location (data coords) as a scalar

\texttt{get\_pad}()
   Get the value of the tick label pad in points

\texttt{get\_pad\_pixels}()

\texttt{get\_view\_interval}()
   return the view interval instance for the axis this tick is ticking

\texttt{set\_clip\_path}(\texttt{clippath}, \texttt{transform=\texttt{None}})
   Set the artist’s clip path, which may be:
   \begin{itemize}
   \item a \texttt{Patch} (or subclass) instance
   \item a \texttt{Path} instance, \texttt{in which case} an optional \texttt{Transform} instance may be provided, which will be applied to the path before using it for clipping.
   \item \texttt{None}, to remove the clipping path
   \end{itemize}
   For efficiency, if the path happens to be an axis-aligned rectangle, this method will set the clipping box to the corresponding rectangle and set the clipping path to \texttt{None}.

   ACCEPTS: [(\texttt{Path}, \texttt{Transform}) | \texttt{Patch} | \texttt{None}]

\texttt{set\_label}(s)
   Set the text of ticklabel

   ACCEPTS: \texttt{str}

\texttt{set\_label1}(s)
   Set the text of ticklabel

   ACCEPTS: \texttt{str}

\texttt{set\_label2}(s)
   Set the text of ticklabel2

   ACCEPTS: \texttt{str}

\texttt{set\_pad}(\texttt{val})
   Set the tick label pad in points

   ACCEPTS: \texttt{float}

\texttt{class \texttt{matplotlib.axis.Ticker}}

   \texttt{formatter} = \texttt{None}
locator = None

class matplotlib.axis.XAxis(axes, pickradius=15):
    Bases: matplotlib.axis.Axis

    Init the axis with the parent Axes instance

    axis_name = ‘x’

    contains(mouseevent)
        Test whether the mouse event occured in the x axis.

    get_data_interval()
        return the Interval instance for this axis data limits

    get_label_position()
        Return the label position (top or bottom)

    get_minpos()

    get_text_heights(renderer)
        Returns the amount of space one should reserve for text above and below the axes. Returns a
tuple (above, below)

    get_ticks_position()
        Return the ticks position (top, bottom, default or unknown)

    get_view_interval()
        return the Interval instance for this axis view limits

    set_data_interval(vmin, vmax, ignore=False)
        set the axis data limits

    set_default_intervals()
        set the default limits for the axis interval if they are not mutated

    set_label_position(position)
        Set the label position (top or bottom)

        ACCEPTS: [ ‘top’ | ‘bottom’ ]

    set_ticks_position(position)
        Set the ticks position (top, bottom, both, default or 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.

        ACCEPTS: [ ‘top’ | ‘bottom’ | ‘both’ | ‘default’ | ‘none’ ]

    set_view_interval(vmin, vmax, ignore=False)
        If ignore is False, the order of vmin, vmax does not matter; the original axis orientation will be
preserved. In addition, the view limits can be expanded, but will not be reduced. This method
is for mpl internal use; for normal use, see set_xlim().

54.1. matplotlib.axis
**tick_bottom()**
use ticks only on bottom

**tick_top()**
use ticks only on top

**class matplotlib.axis.XTick(axes, loc, label, 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)**

Bases: matplotlib.axis.Tick

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

**apply_tickdir(tickdir)**

**get_view_interval()**
return the Interval instance for this axis view limits

**update_position(loc)**
Set the location of tick in data coords with scalar **loc**

**class matplotlib.axis.YAxis(axes, pickradius=15)**

Bases: matplotlib.axis.Axis

Init the axis with the parent Axes instance

**axis_name = 'y'**

**contains(mouseevent)**
Test whether the mouse event occurred in the y axis.

Returns True | False

**get_data_interval()**
return the Interval instance for this axis data limits

**get_label_position()**
Return the label position (left or right)

**get_minpos()**

**get_text_widths(renderer)**

**get_ticks_position()**
Return the ticks position (left, right, both or unknown)

**get_view_interval()**
return the Interval instance for this axis view limits
set_data_interval(vmin, vmax, ignore=False)
    set the axis data limits

set_default_intervals()
    set the default limits for the axis interval if they are not mutated

set_label_position(position)
    Set the label position (left or right)
    ACCEPTS: [ ‘left’ | ‘right’ ]

set_offset_position(position)

set_ticks_position(position)
    Set the ticks position (left, right, both, default or 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.
    ACCEPTS: [ ‘left’ | ‘right’ | ‘both’ | ‘default’ | ‘none’ ]

set_view_interval(vmin, vmax, ignore=False)
    If ignore is False, the order of vmin, vmax does not matter; the original axis orientation will be
    preserved. In addition, the view limits can be expanded, but will not be reduced. This method
    is for mpl internal use; for normal use, see set_ylim().

tick_left()
    use ticks only on left

tick_right()
    use ticks only on right

class matplotlib.axis.YTick(axes, loc, label, size=None, width=None, color=None, tick-
dir=None, pad=None, labelsize=None, labelcolor=None, zorder=None, gridOn=None, tick1On=True, tick2On=True, label1On=True, label2On=False, major=True)

Bases: matplotlib.axis.Tick

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

apply_tickdir(tickdir)

get_view_interval()
    return the Interval instance for this axis view limits

update_position(loc)
    Set the location of tick in data coords with scalar loc
55.1 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 `matplotlib.figure.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__`, inherited from `ShowBase`.

```python
class matplotlib.backend_bases.CloseEvent(name, canvas, guiEvent=None)
    Bases: matplotlib.backend_bases.Event

    An event triggered by a figure being closed

    In addition to the Event attributes, the following event attributes are defined:

class matplotlib.backend_bases.Cursors

    HAND = 0

    MOVE = 3

    POINTER = 1

    SELECT_REGION = 2
```
class matplotlib.backend_bases.DrawEvent(name, canvas, renderer)
    Bases: matplotlib.backend_bases.Event

    An event triggered by a draw operation on the canvas

    In addition to the Event attributes, the following event attributes are defined:
    renderer the RendererBase instance for the draw event

class matplotlib.backend_bases.Event(name, canvas, guiEvent=None)
    A matplotlib event. Attach additional attributes as defined in FigureCanvasBase.mpl_connect().
    The following attributes are defined and shown with their default values
    name the event name
    canvas the FigureCanvas 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.

    Public attributes
    figure A matplotlib.figure.Figure instance
    blit(bbox=None)
        blit the canvas in bbox (default entire canvas)
    button_press_event(x, y, button, dblclick=False, guiEvent=None)
        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 be call all functions connected to the ‘button_press_event’ with a MouseEvent instance.
    button_release_event(x, y, button, guiEvent=None)
        Backend derived classes should call this function on any mouse button release.
        x the canvas coordinates where 0=left
        y the canvas coordinates where 0=bottom
        guiEvent the native UI event that generated the mpl event
        This method will be call all functions connected to the ‘button_release_event’ with a MouseEvent instance.
    close_event(guiEvent=None)
        This method will be called by all functions connected to the ‘close_event’ with a CloseEvent
    draw(*args, **kwargs)
        Render the Figure
    draw_cursor(event)
        Draw a cursor in the event.axes if inaxes is not None. Use native GUI drawing for efficiency if possible
    draw_event(renderer)
        This method will be call all functions connected to the ‘draw_event’ with a DrawEvent
    draw_idle(*args, **kwargs)
        draw() only if idle; defaults to draw but backends can override

842 Chapter 55. backends
**enter_notify_event** *(guiEvent=\texttt{None}, xy=\texttt{None})*

Backend derived classes should call this function when entering canvas

\texttt{guiEvent} the native UI event that generated the mpl event

\texttt{xy} the coordinate location of the pointer when the canvas is entered

**events** = ['resize_event', 'draw_event', 'key_press_event', 'key_release_event', 'button_press_event', 'button_release_event', ...

**filetypes** = {'\texttt{pgf}': 'LaTeX PGF Figure', '\texttt{svgz}': 'Scalable Vector Graphics', '\texttt{tif}': 'Tagged Image File Format'

**flush_events**()

Flush the GUI events for the figure. Implemented only for backends with GUIs.

**get_default_filename**()

Return a string, which includes extension, suitable for use as a default filename.

**get_default_filetype**()

Get the default savefig file format as specified in rcParam savefig.format. Returned string
excludes period. Overridden in backends that only support a single file type.

**get_supported_filetypes**()

Return dict of savefig file formats supported by this backend

**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 ['\texttt{jpg}', 'jpeg'].

**get_width_height**()

Return the figure width and height in points or pixels (depending on the backend), truncated to
integers

**get_window_title**()

Get the title text of the window containing the figure. Return None if there is no window (eg, a
PS backend).

**grab_mouse***(\texttt{ax})*

Set the child axes which are currently 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.

**idle_event** *(\texttt{guiEvent=\texttt{None}})*

Called when GUI is idle.

**is_saving**()

Returns True when the renderer is in the process of saving to a file, rather than rendering for
an on-screen buffer.

**key_press_event** *(\texttt{key, guiEvent=\texttt{None}})*

This method will be call all functions connected to the ‘key_press_event’ with a *KeyEvent*

**key_release_event** *(\texttt{key, guiEvent=\texttt{None}})*

This method will be call all functions connected to the ‘key_release_event’ with a *KeyEvent*
leave_notify_event(guiEvent=None)

Backend derived classes should call this function when leaving canvas

guiEvent the native UI event that generated the mpl event

motion_notify_event(x, y, guiEvent=None)

Backend derived classes should call this function on any motion-notify-event.

x the canvas coordinates where 0 = left

y the canvas coordinates where 0 = bottom

guiEvent the native UI event that generated the mpl event

This method will be call all functions connected to the 'motion_notify_event' with a

MouseEvent instance.

mpl_connect(s, func)

Connect event with string s to func. The signature of func is:

def func(event)

where event is a matplotlib.backend_bases.Event. The following events are recognized

• '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'

For the location events (button and key press/release), if the mouse is over the axes, the variable

event.inaxes will be set to the Axes the event occurs is over, and additionally, the variables

event.xdata and event.ydata will be defined. This is the mouse location in data coords.

See KeyEvent and MouseEvent for more info.

Return value is a connection id that can be used with mpl_disconnect().

Example usage:

def on_press(event):
    print('you pressed', event.button, event.xdata, event.ydata)

cid = canvas.mpl_connect('button_press_event', on_press)

mpl_disconnect(cid)

Disconnect callback id cid

Example usage:
cid = canvas.mpl_connect('button_press_event', on_press)
#...later
canvas.mpl_disconnect(cid)

new_timer(*args, **kwargs)
Creates a new backend-specific subclass of backend_bases.Timer. This is useful for getting periodic events through the backend’s native event loop. Implemented only for backends with GUIs.

optional arguments:
interval Timer interval in milliseconds
callbacks Sequence of (func, args, kwargs) where func(args, **kwargs) will be executed by the timer every *interval.

onHilite(ev)
Mouse event processor which highlights the artists under the cursor. Connect this to the ‘motion_notify_event’ using:

canvas.mpl_connect('motion_notify_event', canvas.onHilite)

onRemove(ev)
Mouse event processor which removes the top artist under the cursor. Connect this to the ‘mouse_press_event’ using:

canvas.mpl_connect('mouse_press_event', canvas.onRemove)

pick(mouseevent)

pick_event(mouseevent, artist, **kwargs)
This method will be called by artists who are picked and will fire off PickEvent callbacks registered listeners

print_bmp(*args, **kwargs)

print_eps(*args, **kwargs)

print_figure(filename, dpi=None, facecolor='w', edgecolor='w', orientation='portrait', format=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.

Arguments are:
filename can also be a file object on image backends
orientation only currently applies to PostScript printing.
dpi the dots per inch to save the figure in; if None, use savefig.dpi
facecolor the facecolor of the figure
dgedgecolor the edgecolor of the figure
orientation landscape’ | ‘portrait’ (not supported on all backends)
format when set, forcibly set the file format to save to
bbox_inches Bbox in inches. Only the given portion of the figure is saved. If ‘tight’, try to
figure out the tight bbox of the figure. If None, use savefig.bbox
pad_inches Amount of padding around the figure when bbox_inches is ‘tight’. If None, use
savefig.pad_inches
bbox_extra_artists A list of extra artists that will be considered when the tight bbox is calcu-
lated.

print_jpeg(filename_or_obj, *args, **kwargs)
Supported kwargs:
quality: The image quality, on a scale from 1 (worst) to 95 (best). The default is 95, if not
given in the matplotlibrc file in the savefig.jpeg_quality parameter. Values above 95
should be avoided; 100 completely disables the JPEG quantization stage.
optimize: If present, indicates that the encoder should make an extra pass over the image in
order to select optimal encoder settings.
progressive: If present, indicates that this image should be stored as a progressive JPEG file.

print_jpg(filename_or_obj, *args, **kwargs)
Supported kwargs:
quality: The image quality, on a scale from 1 (worst) to 95 (best). The default is 95, if not
given in the matplotlibrc file in the savefig.jpeg_quality parameter. Values above 95
should be avoided; 100 completely disables the JPEG quantization stage.
optimize: If present, indicates that the encoder should make an extra pass over the image in
order to select optimal encoder settings.
progressive: If present, indicates that this image should be stored as a progressive JPEG file.

print_pdf(*args, **kwargs)

print_pgf(*args, **kwargs)

print_png(*args, **kwargs)

print_ps(*args, **kwargs)

print_raw(*args, **kwargs)

print_rgba(*args, **kwargs)

print_svg(*args, **kwargs)

print_svgz(*args, **kwargs)

print_tif(filename_or_obj, *args, **kwargs)
print_tiff(filename_or_obj, *args, **kwargs)

releaseMouse(ax)
Release the mouse grab held by the axes, ax. Usually called by the widgets. It is ok to call this even if you ax doesn’t have the mouse grab currently.

resize(w, h)
set the canvas size in pixels

resize_event()
This method will be call all functions connected to the ‘resize_event’ with a ResizeEvent

scroll_event(x, y, step, guiEvent=None)
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 be call all functions connected to the ‘scroll_event’ with a MouseEvent instance.

set_window_title(title)
Set the title text of the window containing the figure. Note that this has no effect if there is no window (eg, a PS backend).

start_event_loop(timeout)
Start an event loop. This is used to start a blocking event loop so that interactive functions, such as ginput and waitforbuttonpress, can wait for events. This should not be confused with the main GUI event loop, which is always running and has nothing to do with this.

This is implemented only for backends with GUIs.

start_event_loop_default(timeout=0)
Start an event loop. This is used to start a blocking event loop so that interactive functions, such as ginput and waitforbuttonpress, can wait for events. This should not be confused with the main GUI event loop, which is always running and has nothing to do with this.

This function provides default event loop functionality based on time.sleep that is meant to be used until event loop functions for each of the GUI backends can be written. As such, it throws a deprecated warning.

Call signature:

start_event_loop_default(self,timeout=0)

This call blocks until a callback function triggers stop_event_loop() or timeout is reached. If timeout is <=0, never timeout.

stop_event_loop()
Stop an event loop. This is used to stop a blocking event loop so that interactive functions, such as ginput and waitforbuttonpress, can wait for events.

This is implemented only for backends with GUIs.
stop_event_loop_default()
Stop an event loop. This is used to stop a blocking event loop so that interactive functions, such as ginput and waitforbuttonpress, can wait for events.

Call signature:

stop_event_loop_default(self)
supports_blit = True

switch_backends(FigureCanvasClass)
Instantiate an instance of FigureCanvasClass

This is used for backend switching, eg, to instantiate a FigureCanvasPS from a FigureCanvas-GTK. Note, deep copying is not done, so any changes to one of the instances (eg, setting figure size or line props), will be reflected in the other

class matplotlib.backend_bases.FigureManagerBase(canvas, num)
Helper class for pyplot mode, wraps everything up into a neat bundle

Public attributes:
canvas A FigureCanvasBase instance
num The figure number
destroy()

full_screen_toggle()

get_window_title()
Get the title text of the window containing the figure. Return None for non-GUI backends (eg, a PS backend).

key_press(event)
Implement the default mpl key bindings defined at Navigation Keyboard Shortcuts

key_press_handler_id = None
The returned id from connecting the default key handler via FigureCanvasBase.mpl_connect().

To disable default key press handling:

manager, canvas = figure.canvas.manager, figure.canvas
canvas.mpl_disconnect(manager.key_press_handler_id)

resize(w, h)
“For gui backends, resize the window (in pixels).

set_window_title(title)
Set the title text of the window containing the figure. Note that this has no effect for non-GUI backends (eg, a PS backend).
show()
For GUI backends, show the figure window and redraw. For non-GUI backends, raise an exception to be caught by show(), for an optional warning.

show_popup(msg)
Display message in a popup – GUI only

class matplotlib.backend_bases.GraphicsContextBase
An abstract base class that provides color, line styles, etc...

copy_properties(gc)
Copy properties from gc to self

dashd = {‘solid’: (None, None), ‘dashed’: (0, (6.0, 6.0)), ‘dotted’: (0, (1.0, 3.0)), ‘dashdot’: (0, (3.0, 5.0, 1.0, 5.0))}

get_alpha()
Return the alpha value used for blending - not supported on all backends

get_antialiased()
Return true if the object should try to do antialiased rendering

get_capstyle()
Return the capstyle as a string in (‘butt’, ‘round’, ‘projecting’) 

get_clip_path()
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()
Return the clip rectangle as a Bbox instance

get_dashes()
Return the dash information as an offset dashlist tuple.
The dash list is a even size list that gives the ink on, ink off in pixels.
See p107 of to PostScript BLUEBOOK for more info.
Default value is None

get_forced_alpha()
Return whether the value given by get_alpha() should be used to override any other alpha-channel values.

get_gid()
Return the object identifier if one is set, None otherwise.

get_hatch()
Gets the current hatch style

get_hatch_path(density=6.0)
Returns a Path for the current hatch.

get_joinstyle()
Return the line join style as one of (‘miter’, ‘round’, ‘bevel’)
get_linestyle(style)
Return the linestyle: one of ('solid', 'dashed', 'dashdot', 'dotted').

get_linewidth()
Return the line width in points as a scalar.

get_rgb()
returns a tuple of three or four floats from 0-1.

get_sketch_params()
Returns the sketch parameters for the artist.
Returns sketch_params: 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()
returns 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

get_url()
returns a url if one is set, None otherwise

restore()
Restore the graphics context from the stack - needed only for backends that save graphics contexts on a stack.

set_alpha(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(b)
True if object should be drawn with antialiased rendering

set_capstyle(cs)
Set the capstyle as a string in ('butt', 'round', 'projecting')

set_clip_path(path)
Set the clip path and transformation. Path should be a TransformedPath instance.

set_clip_rectangle(rectangle)
Set the clip rectangle with sequence (left, bottom, width, height)

set_dashes(dash_offset, dash_list)
Set the dash style for the gc.

dash_offset is the offset (usually 0).
dash_list specifies the on-off sequence as points. (None, None) specifies a solid line

set_foreground(fg, isRGBA=False)
Set the foreground color. fg can be a MATLAB format string, a html hex color string, an rgb or rgba unit tuple, or a float between 0 and 1. In the latter case, grayscale is used.

If you know fg is rgba, set isRGBA=True for efficiency.

set_gid(id)
Sets the id.

set_graylevel(frac)
Set the foreground color to be a gray level with frac

set_hatch(hatch)
Sets the hatch style for filling

set_joinstyle(js)
Set the join style to be one of ('miter', 'round', 'bevel')

set_linestyle(style)
Set the linestyle to be one of ('solid', 'dashed', 'dashdot', 'dotted'). One may specify customized dash styles by providing a tuple of (offset, dash pairs). For example, the predefined linestyles have following values:

'dashed' : (0, (6.0, 6.0)), 'dashdot' : (0, (3.0, 5.0, 1.0, 5.0)), 'dotted' : (0, (1.0, 3.0)),

set_linewidth(w)
Set the linewidth in points

set_sketch_params(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(snap)
Sets 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(url)
Sets the url for links in compatible backends

class matplotlib.backend_bases.IdleEvent(name, canvas, guiEvent=None)
Bases: matplotlib.backend_bases.Event
An event triggered by the GUI backend when it is idle – useful for passive animation

```python
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:

- **key** 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”).

Note: 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.

Example usage:

```python
def on_key(event):
    print('you pressed', event.key, event.xdata, event.ydata)

cid = fig.canvas.mpl_connect('key_press_event', on_key)
```

```python
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:

- **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
- **x** in figure coords, 0,0 = bottom, left

```python
inaxes = None

lastevent = None

x = None

xdata = None

y = None
```
```python
ydata = None

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:

- **button** button pressed None, 1, 2, 3, ‘up’, ‘down’ (up and down are used for scroll events)
- **key** the key depressed when the mouse event triggered (see KeyEvent)
- **step** number of scroll steps (positive for ‘up’, negative for ‘down’)

Example usage:

```python
def on_press(event):
    print('you pressed', event.button, event.xdata, event.ydata)

    cid = fig.canvas.mpl_connect('button_press_event', on_press)

x, y in figure coords, 0,0 = bottom, left button pressed None, 1, 2, 3, ‘up’, ‘down’

button = None

dblclick = None

inaxes = None

step = None

x = None

xdata = None

y = None

ydata = None

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
_init_toolbar() create your toolbar widget
draw_rubberband() (optional) draw the zoom to rect “rubberband” rectangle
press() (optional) whenever a mouse button is pressed, you’ll be notified with the event
release() (optional) whenever a mouse button is released, you’ll be notified with the event
dynamic_update() (optional) dynamically update the window while navigating
set_message() (optional) display message
set_history_buttons() (optional) you can change the history back / forward buttons to indicate disabled / enabled state.

That’s it, we’ll do the rest!

back(*args)
move back up the view lim stack
drag_pan(event)
the drag callback in pan/zoom mode
drag_zoom(event)
the drag callback in zoom mode
draw()
Redraw the canvases, update the locators
draw_rubberband(event, x0, y0, x1, y1)
Draw a rectangle rubberband to indicate zoom limits
dynamic_update()

forward(*args)
Move forward in the view lim stack
home(*args)
Restore the original view
mouse_move(event)

pan(*args)
Activate the pan/zoom tool. pan with left button, zoom with right
press(event)
Called whenever a mouse button is pressed.
press_pan(event)
the press mouse button in pan/zoom mode callback
press_zoom(event)
the press mouse button in zoom to rect mode callback
push_current()
push the current view limits and position onto the stack
release(event)
   this will be called whenever mouse button is released

release_pan(event)
   the release mouse button callback in pan/zoom mode

release_zoom(event)
   the release mouse button callback in zoom to rect mode

save_figure(*args)
   Save the current figure

set_cursor(cursor)
   Set the current cursor to one of the Cursors enums values

set_history_buttons()
   Enable or disable back/forward button

set_message(s)
   Display a message on toolbar or in status bar

toolitems = ((‘Home’, ‘Reset original view’, ‘home’, ‘home’),
               (‘Back’, ‘Back to previous view’, ‘back’, ‘back’),
               (‘Forward’, ‘Forward to next view’, ‘forward’, ‘forward’),
               (‘Configure subplots’,
                ‘Configure subplots’,
                ‘configure_subplots’,
                ‘configure_subplots’),
               (‘Save’, ‘Save the figure’, ‘filesave’, ‘save_figure’))

update()
   Reset the axes stack

zoom(*args)
   Activate zoom to rect mode

exception matplotlib.backend_bases.NonGuiException
   Bases: exceptions.Exception

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
   mouseevent the MouseEvent that generated the pick
   artist the Artist picked
   other extra class dependent attrs – eg a Line2D pick may define different extra attributes than a
       PatchCollection pick event

Example usage:

line, = ax.plot(rand(100), 'o', picker=5)  # 5 points tolerance

def on_pick(event):
   thisline = event.artist
   xdata, ydata = thisline.get_data()
   ind = event.ind
   print(‘on pick line:’, zip(xdata[ind], ydata[ind]))

   cid = fig.canvas.mpl_connect(‘pick_event’, on_pick)
class matplotlib.backend_bases.RendererBase

An abstract base class to handle drawing/rendering operations.

The following methods must be implemented in the backend:

- `draw_path()`
- `draw_image()`
- `draw_text()`
- `get_text_width_height_descent()`

The following methods should be implemented in the backend for optimization reasons:

- `draw_markers()`
- `draw_path_collection()`
- `draw_quad_mesh()`

`close_group(s)`

Close a grouping element with label `s`. Is only currently used by backend_svg.

draw_gouraud_triangle(gc, points, colors, transform)

Draw a Gouraud-shaded triangle.

`points` is a 3x2 array of (x, y) points for the triangle.

`colors` is a 3x4 array of RGBA colors for each point of the triangle.

`transform` is an affine transform to apply to the points.

draw_gouraud_triangles(gc, triangles_array, colors_array, transform)

Draws a series of Gouraud triangles.

`points` is a Nx3x2 array of (x, y) points for the trianglex.

`colors` is a Nx3x4 array of RGBA colors for each point of the triangles.

`transform` is an affine transform to apply to the points.

draw_image(gc, x, y, im)

Draw the image instance into the current axes;

`gc` a GraphicsContext containing clipping information
`x` is the distance in pixels from the left hand side of the canvas.
`y` the distance from the origin. That is, if origin is upper, `y` is the distance from top. If origin is lower, `y` is the distance from bottom

`im` the matplotlib._image.Image instance

draw_markers(gc, marker_path, marker_trans, path, trans, rgbFace=None)

Draws 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.

`gc` the GraphicsContextBase instance

`marker_trans` is an affine transform applied to the marker.

`trans` is an affine transform applied to the path.

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.

draw_path(gc, path, transform, rgbFace=None)

Draws a Path instance using the given affine transform.
**draw_path_collection** *(gc, master_transform, paths, all_transforms, offsets, offsetTrans, facecolors, edgecolors, linewidths, linestyles, antialiaseds, urls, offset_position)*

Draws 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.

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** *(gc, master_transform, meshWidth, meshHeight, coordinates, offsets, offsetTrans, facecolors, antialiased, edgecolors)*

This provides a fallback implementation of `draw_quad_mesh()` that generates paths and then calls `draw_path_collection()`.

**draw_text** *(gc, x, y, s, prop, angle, ismath='TeX!', mtext=None)*

Draw the text instance

`gc` the `GraphicsContextBase` instance
`x` the x location of the text in display coords
`y` the y location of the text baseline in display coords
`s` the text string
`prop` a `matplotlib.font_manager.FontProperties` instance
`angle` the rotation angle in degrees
`mtext` a `matplotlib.text.Text` instance

**backend implementers note**

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:

```python
def bbox_artist(self, renderer):
    return bbox_artist
```

to if 1, and then the actual bounding box will be plotted along with your text.

**flipy**

Return true if y small numbers are top for renderer Is used for drawing text (`matplotlib.text`) and images (`matplotlib.image`) only

**get_canvas_width_height**

return the canvas width and height in display coords

**get_image_magnification**

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()  
return the matplotlib.texmanager.TexManager instance

get_text_width_height_descent(s, prop, ismath)  
get the width and height, and the offset from the bottom to the baseline (descent), in display coords of the string s with FontProperties prop

new_gc()  
Return an instance of a GraphicsContextBase

open_group(s, gid=None)  
Open a grouping element with label s. If gid is given, use gid as the id of the group. Is only currently used by backend_svg.

option_image_nocomposite()  
override this method for renderers that do not necessarily want to rescale and composite raster images. (like SVG)

option_scale_image()  
override this method for renderers that support arbitrary scaling of image (most of the vector backend).

points_to_pixels(points)  
Convert points to display units  
**points** a float or a numpy array of float  
return points converted to pixels

You need to override this function (unless your backend doesn’t have a dpi, eg, postscript or svg). Some imaging systems assume some value for pixels per inch:

points to pixels = points * pixels_per_inch/72.0 * dpi/72.0

start_filter()  
Used in AggRenderer. Switch to a temporary renderer for image filtering effects.

start_rasterizing()  
Used in MixedModeRenderer. Switch to the raster renderer.

stop_filter(filter_func)  
Used in AggRenderer. 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.

stop_rasterizing()  
Used in MixedModeRenderer. Switch back to the vector renderer and draw the contents of the raster renderer as an image on the vector renderer.

strip_math(s)

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:
**width** width of the canvas in pixels

**height** height of the canvas in pixels

class matplotlib.backend_bases.ShowBase

Bases: object

Simple base class to generate a show() callable in backends.

Subclass must override mainloop() method.

mainloop()

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.

Mandatory functions that must be implemented:

- `_timer_start`: Contains backend-specific code for starting the timer
- `_timer_stop`: Contains backend-specific code for stopping the timer

Optional overrides:

- `_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`: This is the internal function that any timer object should call, which will handle the task of running all callbacks that have been set.

Attributes:

- **interval**: The time between timer events in milliseconds. Default is 1000 ms.
- **single_shot**: Boolean flag indicating whether this timer should operate as single shot (run once and then stop). Defaults to False.
- **callbacks**: Stores list of (func, args) tuples that will be called upon timer events. This list can be manipulated directly, or the functions add_callback and remove_callback can be used.

add_callback(func, *args, **kwargs)

Register func to be called by timer when the event fires. Any additional arguments provided will be passed to func.

interval

remove_callback(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.

single_shot

start(interval=None)

Start the timer object. interval is optional and will be used to reset the timer interval first if
stop()
Stop the timer.

**matplotlib.backend_bases.key_press_handler**(*event, canvas, toolbar=None*)

Implement the default mpl key bindings for the canvas and toolbar described at *Navigation Keyboard Shortcuts*

*event* a **KeyEvent** instance
*canvas* a **FigureCanvasBase** instance
*toolbar* a **NavigationToolbar2** instance

**matplotlib.backend_bases.register_backend**(*format, backend_class*)

### 55.2 matplotlib.backends.backend_gtkagg

**TODO** We’ll add this later, importing the gtk backends requires an active X-session, which is not compatible with cron jobs.

### 55.3 matplotlib.backends.backend_qt4agg

Render to qt from agg

**class matplotlib.backends.backend_qt4agg.FigureCanvasQTAgg**(*figure*)

Bases: matplotlib.backends.backend_qt4.FigureCanvasQT, matplotlib.backends.backend_agg.FigureCanvasAgg

The canvas the figure renders into. Calls the draw and print fig methods, creates the renderers, etc...

Public attribute

- **figure** - A Figure instance

**blit**(bbox=None)

Blit the region in bbox

**draw()**

Draw the figure with Agg, and queue a request for a Qt draw.

**drawRectangle**(rect)

**paintEvent**(e)

Copy the image from the Agg canvas to the qt.drawable. In Qt, all drawing should be done inside of here when a widget is shown onscreen.

**print_figure**(args, **kwargs)

**class matplotlib.backends.backend_qt4agg.FigureManagerQTAgg**(*canvas, num*)

Bases: matplotlib.backends.backend_qt4.FigureManagerQT
class matplotlib.backends.backend_qt4agg.NavigationToolbar2QTAgg(canvas, parent, coordinates=True)

coordinates: should we show the coordinates on the right?

matplotlib.backends.backend_qt4agg.new_figure_manager(num, *args, **kwargs)

Create a new figure manager instance

matplotlib.backends.backend_qt4agg.new_figure_manager_given_figure(num, figure)

Create a new figure manager instance for the given figure.

55.4 matplotlib.backends.backend_wxagg

class matplotlib.backends.backend_wxagg.FigureCanvasWxAgg(parent, id, figure)

Bases: matplotlib.backends.backend_agg.FigureCanvasAgg,
matplotlib.backends.backend_wx.FigureCanvasWx

The FigureCanvas contains the figure and does event handling.

In the wxPython backend, it is derived from wxPanel, and (usually) lives inside a frame instantiated by a FigureManagerWx. The parent window probably implements a wxSizer to control the displayed control size - but we give a hint as to our preferred minimum size.

Initialise a FigureWx instance.

   • Initialise the FigureCanvasBase and wxPanel parents.
   • Set event handlers for: EVT_SIZE (Resize event) EVT_PAINT (Paint event)

blit(bbox=None)

Transfer the region of the agg buffer defined by bbox to the display. If bbox is None, the entire buffer is transferred.

draw(drawDC=None)

Render the figure using agg.

filetypes = {'pgf': 'LaTeX PGF Figure', 'svgz': 'Scalable Vector Graphics', 'tif': 'Tagged Image File Format', 'ps': 'Postscript', 'raw': 'Raw RGBA bitmap', 'pdf': 'Portable Document Format', 'rgba': 'Raw RGBA bitmap', 'jpg': 'Joint Photographic Experts Group'}

print_figure(filename, *args, **kwargs)

class matplotlib.backends.backend_wxagg.FrameWxAgg(num, fig)

Bases: matplotlib.backends.backend_wx.FrameWx

get_canvas(fig)

class matplotlib.backends.backend_wxagg.NavigationToolbar2WxAgg(canvas)

Bases: matplotlib.backends.backend_wx.NavigationToolbar2Wx

get_canvas(frame, fig)
matplotlib.backends.backend_wxagg.new_figure_manager(num, *args, **kwargs)
Create a new figure manager instance

matplotlib.backends.backend_wxagg.new_figure_manager_given_figure(num, figure)
Create a new figure manager instance for the given figure.

## 55.5 matplotlib.backends.backend_pdf

A PDF matplotlib backend
Author: Jouni K Seppänen <jks@iki.fi>

class matplotlib.backends.backend_pdf.FigureCanvasPdf(fig)
Bases: matplotlib.backend_bases.FigureCanvasBase

The canvas the figure renders into. Calls the draw and print fig methods, creates the renderers, etc...

Public attribute
figure - A Figure instance

class matplotlib.backends.backend_pdf.Name(name)
Bases: object

PDF name object.

class matplotlib.backends.backend_pdf.Operator(op)
Bases: object

PDF operator object.

class matplotlib.backends.backend_pdf.PdfFile(filename)
Bases: object

PDF file object.

alphaState(alpha)
Return name of an ExtGState that sets alpha to the given value

embedTTF(filename, characters)
Embed the TTF font from the named file into the document.

fontName(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 (or dvi name) of the font.

imageObject(image)
Return name of an imageXObject representing the given image.

markerObject(path, trans, fillp, strokelp, lw, joinstyle, capstyle)
Return name of a markerXObject representing the given path.

reserveObject(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.

writeInfoDict()
Write out the info dictionary, checking it for good form
writeTrailer()
Write out the PDF trailer.

writeXref()
Write out the xref table.

class matplotlib.backends.backend_pdf.PdfPages(filename)
Bases: object
A multi-page PDF file.
Use like this:

# Initialize:
pp = PdfPages('foo.pdf')

# As many times as you like, create a figure fig, then either:
fig.savefig(pp, format='pdf') # note the format argument!
# or:
pp.savefig(fig)

# Once you are done, remember to close the object:
pp.close()

(In reality PdfPages is a thin wrapper around PdfFile, in order to avoid confusion when using savefig and forgetting the format argument.)

Create a new PdfPages object that will be written to the file named filename. The file is opened at once and any older file with the same name is overwritten.

close()
Finalize this object, making the underlying file a complete PDF file.

infodict()
Return a modifiable information dictionary object (see PDF reference section 10.2.1 ‘Document Information Dictionary’).

savefig(figure=None, **kwargs)
Save the Figure instance figure to this file as a new page. If figure is a number, the figure instance is looked up by number, and if figure is None, the active figure is saved. Any other keyword arguments are passed to Figure.savefig.

class matplotlib.backends.backend_pdf.Reference(id)
Bases: object
PDF reference object. Use PdfFile.reserveObject() to create References.

class matplotlib.backends.backend_pdf.Stream(id, len, file, extra=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().
id: object id of stream; len: an unused Reference object for the length of the stream, or None (to use a memory buffer); file: a PdfFile; extra: a dictionary of extra key-value pairs to include in the stream header

\texttt{end()}

Finalize stream.

\texttt{write(data)}

Write some data on the stream.

\texttt{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.

\texttt{matplotlib.backends.backend_pdf.new_figure_manager(num, *args, **kwargs)}

Create a new figure manager instance

\texttt{matplotlib.backends.backend_pdf.new_figure_manager_given_figure(num, figure)}

Create a new figure manager instance for the given figure.

\texttt{matplotlib.backends.backend_pdf.pdfRepr(obj)}

Map Python objects to PDF syntax.

### 55.6 matplotlib.dviread

An experimental 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:

\begin{verbatim}
dvi = Dvi(filename, 72)
# iterate over pages (but only one page is supported for now):
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:
        ...
\end{verbatim}

\texttt{class matplotlib.dviread.Dvi(filename, dpi)}

Bases: object

A dvi (“device-independent”) file, as produced by TeX. The current implementation only reads the first page and does not even attempt to verify the postamble.

Initialize the object. This takes the filename as input and opens the file; actually reading the file happens when iterating through the pages of the file.

\texttt{close()}

Close the underlying file if it is open.
class matplotlib.dviread.DviFont(scale, tfm, texname, vf)
Bases: object

Object that 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).

texname
Name of the font as used internally by TeX and friends. This is usually very different from any external font names, and dviread.PsfontsMap can be used to find the external name of the font.

size
Size of the font in Adobe points, converted from the slightly smaller TeX points.

widths
Widths of glyphs in glyph-space units, typically 1/1000ths of the point size.

size
texname
widths

class matplotlib.dviread.Encoding(filename)
Bases: object

Parses 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)

encoding

class matplotlib.dviread.PsfontsMap(filename)
Bases: object

A psfonts.map formatted file, mapping TeX fonts to PS fonts. Usage:

>>> map = PsfontsMap(find_tex_file('pdftex.map'))
>>> entry = map['ptmbo8r']
>>> entry.texname
'ptmbo8r'
>>> entry.psname
'Times-Bold'
>>> entry.encoding

55.6. matplotlib.dviread
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, dvipdfm.map. 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.

class matplotlib.dviread.Tfm(filename)

A TeX Font Metric file. This implementation covers only the bare minimum needed by the Dvi class.

    checksum
        Used for verifying against the dvi file.

    design_size
        Design size of the font (in what units?)

    width
        Width of each character, needs to be scaled by the factor specified in the dvi file. This is a dict because indexing may not start from 0.

    height
        Height of each character.

    depth
        Depth of each character.

    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.

    Usage:

    vf = Vf(filename)
glyph = vf[code]
glyph.text, glyph.boxes, glyph.width

matplotlib.dviread.find_tex_file(filename, format=None)

    Call kpsewhich to find a file in the texmf tree. If format is not None, it is used as the value for the
    --format option.

    Apparently most existing TeX distributions on Unix-like systems use kpathsea. I hear MikTeX (a
    popular distribution on Windows) doesn’t use kpathsea, so what do we do? (TODO)

    See also:

    Kpathsea documentation The library that kpsewhich is part of.

55.7 matplotlib.type1font

This module contains 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.

class matplotlib.type1font.Type1Font(input)
    Bases: object

    A class representing a Type-1 font, for use by backends.

    parts
        A 3-tuple of the cleartext part, the encrypted part, and the finale of zeros.

    prop
        A dictionary of font properties.
Initialize a Type-1 font. *input* can be either the file name of a pfb file or a 3-tuple of already-decoded Type-1 font parts.

**parts**

**prop**

**transform(** *effects*)**

Transform the font by slanting or extending. *effects* should be a dict where *effects[‘slant’]* is the tangent of the angle that the font is to be slanted to the right (so negative values slant to the left) and *effects[‘extend’]* is the multiplier by which the font is to be extended (so values less than 1.0 condense). Returns a new *Type1Font* object.
56.1 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.Bunch(**kwds)**

Often we want to just collect a bunch of stuff together, naming each item of the bunch; a dictionary’s OK for that, but a small do-nothing class is even handier, and prettier to use. Whenever you want to group a few variables:

```python
>>> point = Bunch(datum=2, squared=4, coord=12)
>>> point.datum
```

By: Alex Martelli
From: http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/52308

**class** matplotlib.cbook.CallbackRegistry(*args)**

Handle registering and disconnecting for a set of signals and callbacks:

```python
>>> 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
```
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. The Python stdlib weakref module can not create weak references to bound methods directly, so we need to create a proxy object to handle weak references to bound methods (or regular free functions). This technique was shared by Peter Parente on his “Mindtrove” blog.

def connect(s, func):
    register func to be called when a signal s is generated func will be called

def disconnect(cid):
    disconnect the callback registered with callback id cid

def process(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.GetRealpathAndStat

class matplotlib.cbook.Grouper(init=[]):
    Bases: object

    This class provides a lightweight way to group arbitrary objects together into disjoint sets when a full-blown graph data structure would be overkill.

    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.

    For example:

    >>> from matplotlib.cbook import Grouper
    >>> class Foo(object):
    ...     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()

Clean dead weak references from the dictionary

get_siblings(a)

Returns all of the items joined with a, including itself.

join(a, *args)

Join given arguments into the same set. Accepts one or more arguments.

joined(a, b)

Returns True if a and b are members of the same set.

class matplotlib.cbook.Idle(func)

Bases: matplotlib.cbook.Scheduler

Schedule callbacks when scheduler is idle

run()

waittime = 0.05

exception matplotlib.cbook(MatplotlibDeprecationWarning

Bases: exceptions.UserWarning

A class for issuing deprecation warnings for Matplotlib users.

In light of the fact that Python built-in 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.

http://docs.python.org/dev/whatsnew/2.7.html#the-future-for-python-2-x

class matplotlib.cbook.MemoryMonitor(nmax=20000)

clear()

plot(i0=0, isub=1, fig=None)

report(segments=4)

xy(i0=0, isub=1)
class matplotlib.cbook.Null(*args, **kwargs)
    Null objects always and reliably “do nothing.”

class matplotlib.cbook.RingBuffer(size_max)
    class that implements a not-yet-full buffer

    append(x)
        append an element at the end of the buffer

get()
    Return a list of elements from the oldest to the newest.

class matplotlib.cbook.Scheduler
    Bases: threading.Thread
    Base class for timeout and idle scheduling

    id = 0

    idlelock = <thread.lock object at 0x11e9810>  

    stop()

class matplotlib.cbook.Sorter
    Sort by attribute or item

    Example usage:

    sort = Sorter()

    list = [(1, 2), (4, 8), (0, 3)]
    dict = [{'a': 3, 'b': 4}, {'a': 5, 'b': 2}, {'a': 0, 'b': 0},
            {'a': 9, 'b': 9}]

    sort(list)     # default sort
    sort(list, 1)  # sort by index 1
    sort(dict, 'a')# sort a list of dicts by key 'a'

    byAttribute(data, attributename, inplace=1)

    byItem(data, itemindex=None, inplace=1)

    sort(data, itemindex=None, inplace=1)

class matplotlib.cbook.Stack(default=None)
    Bases: object

    Implement a stack where elements can be pushed on and you can move back and forth. But no pop. Should mimic home / back / forward in a browser
back()
    move the position back and return the current element

bubble(o)
    raise o to the top of the stack and return o. o must be in the stack

clear()
    empty the stack

empty()

forward()
    move the position forward and return the current element

home()
    push the first element onto the top of the stack

push(o)
    push object onto stack at current position - all elements occurring later than the current position are discarded

remove(o)
    remove element o from the stack

class matplotlib.cbook.Timeout(wait, func)
    Bases: matplotlib.cbook.Scheduler
    Schedule recurring events with a wait time in seconds

    run()

class matplotlib.cbook.Xlator
    Bases: dict
    All-in-one multiple-string-substitution class

    Example usage:

    text = "Larry Wall is the creator of Perl"
    adict = {
        "Larry Wall" : "Guido van Rossum",
        "creator" : "Benevolent Dictator for Life",
        "Perl" : "Python",
    }

    print multiple_replace(adict, text)

    xlat = Xlator(adict)
    print xlat.xlat(text)

    xlat(text)
        Translate text, returns the modified text.
align_iterators(func, *iterables)
This generator takes a bunch of iterables that are ordered by func It sends out ordered tuples:

(func(row), [rows from all iterators matching func(row)])

It is used by mlab.recs_join() to join record arrays

allequal(seq)
Return True if all elements of seq compare equal. If seq is 0 or 1 length, return True

allpairs(x)
return all possible pairs in sequence x

Condensed by Alex Martelli from this thread on c.l.python

alltrue(seq)
Return True if all elements of seq evaluate to True. If seq is empty, return False.

class converter(missing='Null', missingval=None)
Base class for handling string -> python type with support for missing values

is_missing(s)

dedent(s)
Remove excess indentation from docstring s.

Discards any leading blank lines, then removes up to n whitespace characters from each line, where n is the number of leading whitespace characters in the first line. It differs from textwrap.dedent in its deletion of leading blank lines and its use of the first non-blank line to determine the indentation.

It is also faster in most cases.

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 np.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().

```python
matplotlib.cbook.deprecated(since, message='', name='', alternative='', pending=False, obj_type='function')
```

Used to mark a function as deprecated.

**Parameters**

- `since`: str
  The release at which this API became deprecated. This is required.
- `message`: str, optional
  Override the default deprecation message. The format specifier `%(func)s` may be used for the name of the function, and `%(alternative)s` may be used in the deprecation message to insert the name of an alternative to the deprecated function. `%(obj_type)` may be used to insert a friendly name for the type of object being deprecated.
- `name`: str, optional
  The name of the deprecated function; if not provided the name is automatically determined from the passed in function, though this is useful in the case of renamed functions, where the new function is just assigned to the name of the deprecated function. For example:

  ```python
def new_function():
    ...
oldFunction = new_function
```

- `alternative`: str, optional
  An alternative function that the user may use in place of the deprecated function. The deprecation warning will tell the user about this alternative if provided.
- `pending`: bool, optional
  If True, uses a PendingDeprecationWarning instead of a DeprecationWarning.

```python
matplotlib.cbook.dict_delall(d, keys)
```

delete all of the `keys` from the dict `d`

```python
matplotlib.cbook.exception_to_str(s=None)
```

```python
matplotlib.cbook.finddir(o, match, case=False)
```

return all attributes of `o` which match string in `match`. If case is True require an exact case match.

```python
matplotlib.cbook.flatten(seq, scalarp=<function is_scalar_or_string at 0x192a410>)
```

Returns a generator of flattened nested containers

For example:

```python
>>> from matplotlib.cbook import flatten
>>> l = (('John', ['Hunter']), (1, 23), [[[42, (5, 23)], ]])
>>> print list(flatten(l))
['John', 'Hunter', 1, 23, 42, 5, 23]
```
matplotlib.cbook.get_recursive_filelist(args)
Recurse all the files and dirs in args ignoring symbolic links and return the files as a list of strings

matplotlib.cbook.get_sample_data(fname, asfileobj=True)
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.

Set the rc parameter examples.directory to the directory where we should look, if sample_data files are stored in a location different than default (which is ‘mpl-data/sample_data’ at the same level of ‘matplotlib’ Python module files).

If the filename ends in .gz, the file is implicitly ungzipped.

matplotlib.cbook.get_split_ind(seq, N)
seq is a list of words. Return the index into seq such that:
\[
\text{len(' '.join(seq[:ind]))} \leq N
\]

matplotlib.cbook.is_math_text(s)

matplotlib.cbook.is_numlike(obj)
return true if obj looks like a number

matplotlib.cbook.is_scalar(obj)
return true if obj is not string like and is not iterable

matplotlib.cbook.is_scalar_or_string(val)
Return whether the given object is a scalar or string like.

matplotlib.cbook.is_sequence_of_strings(obj)
Returns true if obj is iterable and contains strings

matplotlib.cbook.is_string_like(obj)
Return True if obj looks like a string

matplotlib.cbook.is_writable_file_like(obj)
return true if obj looks like a file object with a write method

matplotlib.cbook.issubclass_safe(x, klass)
return issubclass(x, klass) and return False on a TypeError

matplotlib.cbook.iterable(obj)
return true if obj is iterable

matplotlib.cbook.listFiles(root, patterns='*', recurse=1, return_folders=0)
Recursively list files

from Parnar and Martelli in the Python Cookbook
class matplotlib.cbook.maxdict(maxsize)
    Bases: dict

    A dictionary with a maximum size; this doesn’t override all the relevant methods to contrain size, just
setitem, so use with caution

matplotlib.cbook.mkdirs(newdir, mode=511)
    make directory newdir recursively, and set mode. Equivalent to

    > mkdir -p NEWDIR
    > chmod MODE NEWDIR

matplotlib.cbook.mplDeprecation
    alias of MatplotlibDeprecationWarning

matplotlib.cbook.oneture(seq)
    Return True if one element of seq is True. It seq is empty, return False.

matplotlib.cbook.pieces(seq, num=2)
    Break up the seq into num tuples

matplotlib.cbook.popall(seq)
    empty a list

matplotlib.cbook.print_cycles(objects, outstream=<open file ‘stdout’>, mode ‘w’ at
0x7f4157153150>, show_progress=False)

    objects A list of objects to find cycles in. It is often useful to pass in gc.garbage to find the cycles that
are preventing some objects from being garbage collected.
    outstream The stream for output.
    show_progress If True, print the number of objects reached as they are found.

matplotlib.cbook.recursive_remove(path)

matplotlib.cbook.report_memory(i=0)
    return the memory consumed by process

matplotlib.cbook.restrict_dict(d, keys)
    Return a dictionary that contains those keys that appear in both d and keys, with values from d.

matplotlib.cbook.reverse_dict(d)
    reverse the dictionary – may lose data if values are not unique!

matplotlib.cbook.safe_masked_invalid(x)

matplotlib.cbook.safezip(*args)
    make sure args are equal len before zipping

class matplotlib.cbook.silent_list(type, seq=None)
    Bases: list

    override repr when returning a list of matplotlib artists to prevent long, meaningless output. This is
meant to be used for a homogeneous list of a given type
matplotlib.cbook.simple_linear_interpolation\((a, \text{steps})\)

matplotlib.cbook.soundex\((\text{name}, \text{len}=4)\)
    soundex module conforming to Odell-Russell algorithm

matplotlib.cbook.strip_math\((s)\)
    remove latex formatting from mathtext

matplotlib.cbook.to_filehandle\((\text{fname}, \text{flag}=\text{'rU'}, \text{return_opened}=\text{False})\)
    \text{fname} can be a filename or a file handle. Support for gzipped files is automatic, if the filename ends in .gz. \text{flag} is a read/write flag for \text{file()}

class matplotlib.cbook.todate\((\text{fmt}=\text{''%Y-%m-%d''}, \text{missing}=\text{'Null'}, \text{missingval}=\text{None})\)
    Bases: matplotlib.cbook.converter
    convert to a date or None
    use a \text{time.strptime()} format string for conversion

class matplotlib.cbook.todatetime\((\text{fmt}=\text{''%Y-%m-%d''}, \text{missing}=\text{'Null'}, \text{missingval}=\text{None})\)
    Bases: matplotlib.cbook.converter
    convert to a datetime or None
    use a \text{time.strptime()} format string for conversion

class matplotlib.cbook.tofloat\((\text{missing}=\text{'Null'}, \text{missingval}=\text{None})\)
    Bases: matplotlib.cbook.converter
    convert to a float or None

class matplotlib.cbook.toint\((\text{missing}=\text{'Null'}, \text{missingval}=\text{None})\)
    Bases: matplotlib.cbook.converter
    convert to an int or None

class matplotlib.cbook.tostr\((\text{missing}=\text{'Null'}, \text{missingval}=\text{''}\))
    Bases: matplotlib.cbook.converter
    convert to string or None

matplotlib.cbook.unicode_safe\((s)\)

matplotlib.cbook.unique\((x)\)
    Return a list of unique elements of \text{x}

matplotlib.cbook.unmasked_index_ranges\((\text{mask}, \text{compressed}=\text{True})\)
    Find index ranges where \text{mask} is \text{False}.
    \text{mask} will be flattened if it is not already 1-D.

    Returns Nx2 numpy.ndarray with each row the start and stop indices for slices of the compressed numpy.ndarray corresponding to each of \text{N} uninterrupted runs of unmasked values. If optional argument \text{compressed} is \text{False}, it returns the start and stop indices into the original numpy.ndarray, not the compressed numpy.ndarray. Returns \text{None} if there are no unmasked values.
Example:

```python
y = ma.array(np.arange(5), mask = [0,0,1,0,0])
ii = unmasked_index_ranges(ma.getmaskarray(y))
# returns array [[0,2], [2,4]]

y.compressed()[ii[1,0]:ii[1,1]]
# returns array [3,4,]

ii = unmasked_index_ranges(ma.getmaskarray(y), compressed=False)
# returns array [[0, 2], [3, 5]]

y.filled()[ii[1,0]:ii[1,1]]
# returns array [3,4,]
```

Prior to the transforms refactoring, this was used to support masked arrays in Line2D.

```python
matplotlib.cbook.warn_deprecated(since, message=' ', name=' ', alternative=' ', pending=False, obj_type='attribute')
```

Used to display deprecation warning in a standard way.

**Parameters**

- **since**: str
  - The release at which this API became deprecated.

- **message**: str, optional
  - Override the default deprecation message. The format specifier `%func`s may be used for the name of the function, and `%(alternative)`s may be used in the deprecation message to insert the name of an alternative to the deprecated function. `%%(obj_type)` may be used to insert a friendly name for the type of object being deprecated.

- **name**: str, optional
  - The name of the deprecated function; if not provided the name is automatically determined from the passed in function, though this is useful in the case of renamed functions, where the new function is just assigned to the name of the deprecated function. For example:

```python
def new_function():
    ...
oldFunction = new_function
```

- **alternative**: str, optional
  - An alternative function that the user may use in place of the deprecated function. The deprecation warning will tell the user about this alternative if provided.

- **pending**: bool, optional
  - If True, uses a PendingDeprecationWarning instead of a DeprecationWarning.

- **obj_type**: str, optional
  - The object type being deprecated.

```python
matplotlib.cbook.wrap(prefix, text, cols)
```

wrap text with prefix at length cols
57.1 matplotlib.cm

This module provides a large set of colormaps, functions for registering new colormaps and for getting a
colormap by name, and a mixin class for adding color mapping functionality.

class matplotlib.cm.ScalarMappable(norm=None, cmap=None)

This is a mixin class to support scalar data to RGBA mapping. The ScalarMappable makes use of
data normalization before returning RGBA colors from the given colormap.

    Parameters norm : matplotlib.colors.Normalize instance
                      The normalizing object which scales data, typically into the interval
                      [0, 1].
    cmap : str or Colormap instance
           The colormap used to map normalized data values to RGBA colors.

add_checker(checker)

Add an entry to a dictionary of boolean flags that are set to True when the mappable is changed.

autoscale()

Autoscale the scalar limits on the norm instance using the current array

autoscale_None()

Autoscale the scalar limits on the norm instance using the current array, changing only limits
that are None

changed()

Call this whenever the mappable is changed to notify all the callbackSM listeners to the
‘changed’ signal

check_update(checker)

If mappable has changed since the last check, return True; else return False

cmap = None

The Colormap instance of this ScalarMappable.

colorbar = None

The last colorbar associated with this ScalarMappable. May be None.

get_array()

Return the array
get_clim()
return the min, max of the color limits for image scaling

get_cmap()
return the colormap

norm = None
The Normalization instance of this ScalarMappable.

set_array(A)
Set the image array from numpy array A

set_clim(*vmin, *vmax)
set the norm limits for image scaling; if vmin is a length 2 sequence, interpret it as (vmin, vmax) which is used to support setp

ACCEPTS: a length 2 sequence of floats

set_cmap(cmap)
set the colormap for luminance data

ACCEPTS: a colormap or registered colormap name

set_colorbar(*args, **kwargs)
Depreciated since version 1.3: The set_colorbar function was deprecated in version 1.3. Use the colorbar attribute instead.

set_norm(norm)
set the normalization instance

to_rgba(x, alpha=None, bytes=False)
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. 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.

Note: this method assumes the input is well-behaved; it does not check for anomalies such as x being a masked rgba array, or being an integer type other than uint8, or being a floating point rgba array with values outside the 0-1 range.

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.
If \emph{name} is a \texttt{matplotlib.colors.Colormap} instance, it will be returned.

If \emph{lut} is not None it must be an integer giving the number of entries desired in the lookup table, and \emph{name} must be a standard mpl colormap name with a corresponding data dictionary in \emph{datad}.

\texttt{matplotlib.cm.register_cmap(name=None, cmap=None, data=None, lut=None)}

Add a colormap to the set recognized by \texttt{get_cmap()}. It can be used in two ways:

\begin{verbatim}
register_cmap(name='swirly', cmap=swirly_cmap)
register_cmap(name='choppy', data=choppydata, lut=128)
\end{verbatim}

In the first case, \emph{cmap} must be a \texttt{matplotlib.colors.Colormap} instance. The \emph{name} is optional; if absent, the name will be the \emph{name} attribute of the \emph{cmap}.

In the second case, the three arguments are passed to the \texttt{LinearSegmentedColormap} initializer, and the resulting colormap is registered.

\texttt{matplotlib.cm.revcmap(data)}

Can only handle specification \emph{data} in dictionary format.
58.1 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).

```python
class matplotlib.collections.AsteriskPolygonCollection(numsides, rotation=0, sizes=(1,), **kwargs)
```

Bases: `matplotlib.collections.RegularPolyCollection`

Draw a collection of regular asterisks with `numsides` points. `numsides` the number of sides of the polygon. `rotation` the rotation of the polygon in radians. `sizes` gives the area of the circle circumscribing the regular polygon in points^2.

Valid Collection keyword arguments:
Matplotlib, Release 1.3.0

- `edgecolors`: None
- `facecolors`: None
- `linewidths`: None
- `antialiaseds`: None
- `offsets`: None
- `transOffset`: transforms.IdentityTransform()
- `norm`: None (optional for `matplotlib.cm.ScalarMappable`)
- `cmap`: None (optional for `matplotlib.cm.ScalarMappable`)

`offsets` and `transOffset` are used to translate the patch after rendering (default no offsets).

If any of `edgecolors`, `facecolors`, `linewidths`, `antialiaseds` are None, they default to their `matplotlib.rcParams` patch setting, in sequence form.

Example: see `examples/dynamic_collection.py` for complete example:

```python
offsets = np.random.rand(20, 2)
facecolors = [cm.jet(x) for x in np.random.rand(20)]
black = (0, 0, 0, 1)

collection = RegularPolyCollection(
    numsides=5, # a pentagon
    rotation=0, sizes=(50,),
    facecolors=facecolors,
    edgecolors=(black,),
    linewidths=(1,),
    offsets=offsets,
    transOffset=ax.transData,
)
```

```python
class matplotlib.collections.BrokenBarHCollection(xranges, yrange, **kwargs)
    Bases: matplotlib.collections.PolyCollection

    A collection of horizontal bars spanning `yrange` with a sequence of `xranges`.

    :param xranges: sequence of `(xmin, xwidth)`
    :param yrange: `ymin, ywidth`

    Valid Collection keyword arguments:
    - `edgecolors`: None
    - `facecolors`: None
    - `linewidths`: None
    - `antialiaseds`: None
    - `offsets`: None
    - `transOffset`: transforms.IdentityTransform()
    - `norm`: None (optional for `matplotlib.cm.ScalarMappable`)
    - `cmap`: None (optional for `matplotlib.cm.ScalarMappable`)

    `offsets` and `transOffset` are used to translate the patch after rendering (default no offsets).

    If any of `edgecolors`, `facecolors`, `linewidths`, `antialiaseds` are None, they default to their `matplotlib.rcParams` patch setting, in sequence form.

    static `span_where`(x, ymin, ymax, where, **kwargs)
    Create a BrokenBarHCollection to plot horizontal bars from over the regions in `x` where `where` is True. The bars range on the y-axis from `ymin` to `ymax`
```
A `BrokenBarHCollection` is returned. `kwargs` are passed on to the collection.

```python
class matplotlib.collections.CircleCollection(sizes, **kwargs)
```

**Bases:** `matplotlib.collections.Collection`

A collection of circles, drawn using splines.

- `sizes` Gives the area of the circle in points^2

Valid Collection keyword arguments:
- `edgecolors`: None
- `facecolors`: None
- `linewidths`: None
- `antialiaseds`: None
- `offsets`: None
- `transOffset`: `transforms.IdentityTransform()`
- `norm`: None (optional for `matplotlib.cm.ScalarMappable`
- `cmap`: None (optional for `matplotlib.cm.ScalarMappable`

`offsets` and `transOffset` are used to translate the patch after rendering (default no offsets)

If any of `edgecolors`, `facecolors`, `linewidths`, `antialiaseds` are None, they default to their `matplotlib.rcParams` patch setting, in sequence form.

```python
draw(artist, renderer, *args, **kwargs)
```

```python
def get_sizes()
    return sizes of circles
```

```python
class matplotlib.collections.Collection(edgecolors=None, facecolors=None, linewidths=None, linestyles=None, antialiaseds=None, offsets=None, transOffset=None, norm=None, cmap=None, pickradius=5.0, hatch=None, urls=None, offset_position='screen', zorder=1, **kwargs)
```

**Bases:** `matplotlib.artist.Artist, matplotlib.cm.ScalarMappable`

Base class for Collections. Must be subclassed to be usable.

All properties in a collection must be sequences or scalars; if scalars, they will be converted to sequences. The property of the ith element of the collection is:

```python
prop[i % len(props)]
```

Keyword arguments and default values:
- `edgecolors`: None
- `facecolors`: None
- `linewidths`: None
- `antialiaseds`: None
- `offsets`: None
- `transOffset`: `transforms.IdentityTransform()`
- `offset_position`: 'screen' (default) or 'data'
- `norm`: None (optional for `matplotlib.cm.ScalarMappable`)
Matplotlib, Release 1.3.0

- **cmap**: None (optional for *matplotlib.cm.ScalarMappable*)
- **hatch**: None
- **zorder**: 1

Offsets and `transOffset` are used to translate the patch after rendering (default no offsets). 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.

If any of `edgecolors`, `facecolors`, `linewidths`, `antialiaseds` are None, they default to their *matplotlib.rcParams* patch setting, in sequence form.

The use of *ScalarMappable* is optional. If the *ScalarMappable* matrix \_A is not None (ie a call to `set_array` has been made), at draw time a call to scalar mappable will be made to set the face colors.

Create a Collection

```
%(Collection)s
```

`contains(mouseevent)`

Test whether the mouse event occurred in the collection.

Returns True | False, dict(ind=itemlist), where every item in itemlist contains the event.

`draw(artist, renderer, *args, **kwargs)`

`get_dashes()`

`get_datalim(transData)`

`get_edgecolor()`

`get_edgecolors()`

`get_facecolor()`

`get_facecolors()`

`get_hatch()`

Return the current hatching pattern

`get_linestyle()`

`get_linestyles()`

`get_linewidth()`
get_linewidths()

get_offset_position()
Returns how offsets are applied for the collection. If offset_position is ‘screen’, the offset is applied after the master transform has been applied, i.e., 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.

get_offset_transform()

get_offsets()
Return the offsets for the collection.

get_paths()

get_pickradius()

get_transforms()

get_urls()

get_window_extent(renderer)

set_alpha(alpha)
Set the alpha transparencies of the collection. alpha must be a float or None.

ACCEPTS: float or None

set_antialiased(aa)
Set the antialiasing state for rendering.

ACCEPTS: Boolean or sequence of booleans

set_antialiaseds(aa)
alias for set_antialiased

set_color(c)
Set both the edgecolor and the facecolor.

ACCEPTS: matplotlib color arg or sequence of rgba tuples

See also:
set_facecolor(), set_edgecolor() For setting the edge or face color individually.

set_dashes(ls)
alias for set_linestyle

set_edgecolor(c)
Set the edgecolor(s) of the collection. c can be a matplotlib color arg (all patches have same
color), or a sequence of rgba tuples; if it is a sequence the patches will cycle through the sequence.

If \( c \) is ‘face’, the edge color will always be the same as the face color. If it is ‘none’, the patch boundary will not be drawn.

ACCEPTS: matplotlib color arg or sequence of rgba tuples

**set_edgecolors** \((c)\)

alias for set_edgecolor

**set_facecolor** \((c)\)

Set the facecolor(s) of the collection. \( c \) can be a matplotlib color arg (all patches have same color), or a sequence of rgba tuples; if it is a sequence the patches will cycle through the sequence.

If \( c \) is ‘none’, the patch will not be filled.

ACCEPTS: matplotlib color arg or sequence of rgba tuples

**set_facecolors** \((c)\)

alias for set_facecolor

**set_hatch** \((hatch)\)

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.

ACCEPTS: \[ `/' | `\` | `'| '-' | '+' | 'x' | 'o' | 'O' | '.' | '*' \] \]

**set_linestyle** \((ls)\)

Set the linestyle(s) for the collection.

ACCEPTS: \[ `solid` | `dashed`, `dashdot`, `dotted` | (offset, on-off-dash-seq) \]

**set_linestyles** \((ls)\)

alias for set_linestyle
**set_linewidth(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.

ACCEPTS: float or sequence of floats

**set_linewidths(lw)**
alias for set_linewidth

**set_lw(lw)**
alias for set_linewidth

**set_offset_position(offset_position)**
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.

**set_offsets(offsets)**
Set the offsets for the collection. *offsets* can be a scalar or a sequence.

ACCEPTS: float or sequence of floats

**set_paths()**

**set_pickradius(pr)**

**set_urls(urls)**

**update_from(other)**
copy properties from other to self

**update_scalarmappable()**
If the scalar mappable array is not none, update colors from scalar data

**class** `matplotlib.collections.EllipseCollection(widths, heights, angles, units='points', **kwargs)`

Bases: `matplotlib.collections.Collection`

A collection of ellipses, drawn using splines.

**widths**: sequence
lengths of first axes (e.g., major axis lengths)

**heights**: sequence
lengths of second axes

**angles**: sequence
angles of first axes, degrees CCW from the X-axis

**units**: ['points' | 'inches' | 'dots' | 'width' | 'height' | 'x' | 'y' | 'xy']
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.

Additional kwargs inherited from the base `Collection`:
Valid Collection keyword arguments:
Matplotlib, Release 1.3.0

- `edgecolors`: None
- `facecolors`: None
- `linewidths`: None
- `antialiaseds`: None
- `offsets`: None
- `transOffset`: `transforms.IdentityTransform()`
- `norm`: None (optional for `matplotlib.cm.ScalarMappable`)
- `cmap`: None (optional for `matplotlib.cm.ScalarMappable`)

`offsets` and `transOffset` are used to translate the patch after rendering (default no offsets)

If any of `edgecolors`, `facecolors`, `linewidths`, `antialiaseds` are None, they default to their `matplotlib.rcParams` patch setting, in sequence form.

`draw`(*artist, renderer, *args, **kwargs*)

class `matplotlib.collections.EventCollection`(*positions, orientation=None, lineoffset=0, linelength=1, linewidth=None, color=None, linestyle='solid', antialiased=None, **kwargs*)

Bases: `matplotlib.collections.LineCollection`

A collection of discrete events.

An event is a 1-dimensional value, usually the position of something along an axis, such as time or length. Events do not have an amplitude. They are displayed as v

- `positions` a sequence of numerical values or a 1D numpy array. Can be None
- `orientation` [`'horizontal'` | `'vertical'` | `None` ] defaults to ‘horizontal’ if not specified or None
- `lineoffset` a single numerical value, corresponding to the offset of the center of the markers from the origin
- `linelength` a single numerical value, corresponding to the total height of the marker (i.e. the marker stretches from lineoffset+linelength/2 to lineoffset-linelength/2). Defaults to 1
- `linewidth` a single numerical value
- `color` must be a sequence of RGBA tuples (eg arbitrary color strings, etc, not allowed).
- `linestyle` [ `solid` | `dashed` | `dashdot` | `dotted` ]
- `antialiased` 1 or 2

If `linewidth`, `color`, or `antialiased` is None, they default to their rcParams setting, in sequence form.

- `norm` None (optional for `matplotlib.cm.ScalarMappable`)
- `cmap` None (optional for `matplotlib.cm.ScalarMappable`)

`pickradius` is the tolerance for mouse clicks picking a line. The default is 5 pt.

The use of `ScalarMappable` is optional. If the `ScalarMappable` array `_A` is not None (ie a call to `set_array()` has been made), at draw time a call to scalar mappable will be made to set the colors.

Example:
add_positions(position)
add one or more events at the specified positions

append_positions(position)
add one or more events at the specified positions

extend_positions(position)
add one or more events at the specified positions

get_color()
get the color of the lines used to mark each event

get_linelength()
get the length of the lines used to mark each event

get_lineoffset()
get the offset of the lines used to mark each event

get_linestyle()
get the style of the lines used to mark each event [ ‘solid’ | ‘dashed’ | ‘dashdot’ | ‘dotted’ ]

get_linewidth()
get the width of the lines used to mark each event

get_orientation()
get the orientation of the event line, may be: [ ‘horizontal’ | ‘vertical’ ]
get_positions()
    return an array containing the floating-point values of the positions

is_horizontal()
    True if the eventcollection is horizontal, False if vertical

set_linelength(linelength)
    set the length of the lines used to mark each event

set_lineoffset(lineoffset)
    set the offset of the lines used to mark each event

set_orientation(orientation=None)
    set the orientation of the event line [ ‘horizontal’ | ‘vertical’ | None ] defaults to ‘horizontal’ if not specified or None

set_positions(positions)
    set the positions of the events to the specified value

switch_orientation()
    switch the orientation of the event line, either from vertical to horizontal or vice versus

class matplotlib.collections.LineCollection(segments, lineweights=None, colors=None, antialiaseds=None, linestyles=’solid’, offsets=None, transOffset=None, norm=None, cmap=None, pickradius=5, zorder=2, **kwargs)

Bases: matplotlib.collections.Collection

All parameters must be sequences or scalars; if scalars, they will be converted to sequences. The property of the ith line segment is:

prop[i % len(props)]

i.e., the properties cycle if the len of props is less than the number of segments.

segments 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 be a different length.

colors must be a sequence of RGBA tuples (eg arbitrary color strings, etc, not allowed).

antialiaseds must be a sequence of ones or zeros

linestyles [ ‘solid’ | ‘dashed’ | ‘dashdot’ | ‘dotted’ ] a string or dash tuple. The dash tuple is:

    (offset, onoffseq),

    where onoffseq is an even length tuple of on and off ink in points.

If lineweights, colors, or antialiaseds is None, they default to their rcParams setting, in sequence form.

If offsets and transOffset are not None, then offsets are transformed by transOffset and applied after the segments have been transformed to display coordinates.
If `offsets` is not None but `transOffset` is None, then the `offsets` are added to the segments before any transformation. In this case, a single offset can be specified as:

```python
offsets=(xo,yo)
```

and this value will be added cumulatively to each successive segment, so as to produce a set of successively offset curves.

- **norm**: None (optional for `matplotlib.cm.ScalarMappable`)
- **cmap**: None (optional for `matplotlib.cm.ScalarMappable`)

`pickradius` is the tolerance for mouse clicks picking a line. The default is 5 pt.

- **zorder**: The zorder of the LineCollection. Default is 2

The use of `ScalarMappable` is optional. If the `ScalarMappable` array `_A` is not None (ie a call to `set_array()` has been made), at draw time a call to scalar mappable will be made to set the colors.

- **color**
  
  Set the color(s) of the line collection. `c` can be a `matplotlib` color arg (all patches have same color), or a sequence or rgba tuples; if it is a sequence the patches will cycle through the sequence.

  ACCEPTS: `matplotlib` color arg or sequence of rgba tuples

  ```python
  get_color()
  
  get_colors()
  
  get_segments()
  ```

  ```python
  set_color(c)
  ```

  Set the color(s) of the line collection. `c` can be a `matplotlib` color arg (all patches have same color), or a sequence or rgba tuples; if it is a sequence the patches will cycle through the sequence.

  ACCEPTS: `matplotlib` color arg or sequence of rgba tuples

  ```python
  set_paths(segments)
  
  set_segments(segments)
  
  set_verts(segments)
  ```

### 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 matplotlib.rcParams patch setting, in sequence form.

The use of ScalarMappable is optional. If the ScalarMappable matrix _A is not None (i.e. a call to set_array has been made), at draw time a call to scalar mappable will be made to set the face colors.

set_paths(patches)

class matplotlib.collections.PathCollection(paths, sizes=None, **kwargs)
Bases: matplotlib.collections.Collection

This is the most basic Collection subclass.

paths is a sequence of matplotlib.path.Path instances.

Valid Collection keyword arguments:
- edgecolors: None
- facecolors: None
- linewidths: None
- antialiaseds: None
- offsets: None
- transOffset: transforms.IdentityTransform()
- norm: None (optional for matplotlib.cm.ScalarMappable)
- cmap: None (optional for matplotlib.cm.ScalarMappable)

offsets and transOffset are used to translate the patch after rendering (default no offsets)

If any of edgecolors, facecolors, linewidths, antialiaseds are None, they default to their matplotlib.rcParams patch setting, in sequence form.

draw(artist, renderer, *args, **kwargs)

get_paths()

get_sizes()

set_paths(paths)

class matplotlib.collections.PolyCollection(verts, sizes=None, closed=True, **kwargs)
Bases: matplotlib.collections.Collection

verts is a sequence of (verts0, verts1, ...) where verts_i is a sequence of xy tuples of vertices, or an equivalent numpy array of shape (nv, 2).

sizes is None (default) or a sequence of floats that scale the corresponding verts_i. The scaling is applied before the Artist master transform; if the latter is an identity transform, then the overall scaling.
is such that if $verts_i$ specify a unit square, then $sizes_i$ is the area of that square in points$^2$. If len($sizes$) < $nv$, the additional values will be taken cyclically from the array.

`closed`, when `True`, will explicitly close the polygon.

Valid Collection keyword arguments:

- `edgecolors`: None
- `facecolors`: None
- `linewidths`: None
- `antialiaseds`: None
- `offsets`: None
- `transOffset`: transforms.IdentityTransform()
- `norm`: None (optional for `matplotlib.cm.ScalarMappable`)
- `cmap`: None (optional for `matplotlib.cm.ScalarMappable`)

`offsets` and `transOffset` are used to translate the patch after rendering (default no offsets).

If any of `edgecolors`, `facecolors`, `linewidths`, `antialiaseds` are None, they default to their `matplotlib.rcParams` patch setting, in sequence form.

**draw**(artist, renderer, *args, **kwargs)

**set_paths**(verts, closed=True)

This allows one to delay initialization of the vertices.

**set_verts**(verts, closed=True)

This allows one to delay initialization of the vertices.

**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’

**static convert_mesh_to_paths**(meshWidth, meshHeight, coordinates)

Converts a given mesh into a sequence of `matplotlib.path.Path` objects for easier rendering by backends that do not directly support quadmeshes.
This function is primarily of use to backend implementers.

**convert_mesh_to_triangles** *(meshWidth, meshHeight, coordinates)*

Converts a given mesh into a sequence of triangles, each point with its own color. This is useful for experiments using `draw_quaourad_triangle`.

**draw** *(artist, renderer, *args, **kwargs)*

**get_datalim** *(transData)*

**get_paths** *

**set_paths** *

class **matplotlib.collections.RegularPolyCollection** *(numsides, rotation=0, sizes=(1,), **kwargs)*

Bases: **matplotlib.collections.Collection**

Draw a collection of regular polygons with *numsides*.

*numsides* the number of sides of the polygon

*rotation* the rotation of the polygon in radians

*sizes* gives the area of the circle circumscribing the regular polygon in points^2

Valid Collection keyword arguments:

- *edgecolors*: None
- *facecolors*: None
- *linwidths*: None
- *antialiaseds*: None
- *offsets*: None
- *transOffset*: transforms.IdentityTransform()
- *norm*: None (optional for **matplotlib.cm.ScalarMappable**)
- *cmap*: None (optional for **matplotlib.cm.ScalarMappable**)

*offsets* and *transOffset* are used to translate the patch after rendering (default no offsets)

If any of *edgecolors*, *facecolors*, *linwidths*, *antialiaseds* are None, they default to their **matplotlib.rcParams** patch setting, in sequence form.

Example: see examples/dynamic_collection.py for complete example:

```python
offsets = np.random.rand(20, 2)
facecolors = [cm.jet(x) for x in np.random.rand(20)]
black = (0, 0, 0, 1)

collection = RegularPolyCollection(
    numsides=5,  # a pentagon
    rotation=0, sizes=(50,),
    facecolors=facecolors,
    edgecolors=(black,),
    linewidths=(1,),
    offsets=offsets,
```
transOffset = ax.transData,
)

draw(artifact, renderer, *args, **kwargs)

get_numsides()

get_rotation()

get_sizes()

class matplotlib.collections.StarPolygonCollection(numsides, rotation=0, sizes=(1,), **kwargs)

Bases: matplotlib.collections.RegularPolyCollection

Draw a collection of regular stars with numsides points.

numsides the number of sides of the polygon

rotation the rotation of the polygon in radians

sizes gives the area of the circle circumscribing the regular polygon in points^2

Valid Collection keyword arguments:

- edgecolors: None
- facecolors: None
- linewidths: None
- antialiaseds: None
- offsets: None
- transOffset: transforms.IdentityTransform()
- norm: None (optional for matplotlib.cm.ScalarMappable)
- cmap: None (optional for matplotlib.cm.ScalarMappable)

offsets and transOffset are used to translate the patch after rendering (default no offsets)

If any of edgecolors, facecolors, linewidths, antialiaseds are None, they default to their matplotlib.rcParams patch setting, in sequence form.

Example: see examples/dynamic_collection.py for complete example:

offsets = np.random.rand(20,2)
facecolors = [cm.jet(x) for x in np.random.rand(20)]
black = (0,0,0,1)

collection = RegularPolyCollection(
    numsides=5, # a pentagon
    rotation=0, sizes=(50,),
    facecolors = facecolors,
    edgecolors = (black,),
    linewidths = (1,),
    offsets = offsets,
    transOffset = ax.transData,
)
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.

static convert_mesh_to_paths(tri)

Converts a given mesh into a sequence of matplotlib.path.Path objects for easier rendering by backends that do not directly support meshes.

This function is primarily of use to backend implementers.

draw(artist, renderer, *args, **kwargs)

get_paths()

set_paths()
COLORBAR

59.1 matplotlib.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()`.

```python
class matplotlib.colorbar.Colorbar(ax, mappable, **kw)
```

Bases: `matplotlib.colorbar.ColorbarBase`

This class connects a `ColorbarBase` to a `ScalarMappable` such as a `AxesImage` generated via `imshow()`.

It is not intended to be instantiated directly; instead, use `colorbar()` or `colorbar()` to make your colorbar.

**add_lines(CS, erase=True)**

Add the lines from a non-filled `ContourSet` to the colorbar.

Set `erase` to False if these lines should be added to any pre-existing lines.

**on_mappable_changed(mappable)**

Updates 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.

**update_bruteforce(mappable)**

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.
update_normal(mappable)

update solid, lines, etc. Unlike update_bruceforce, it does not clear the axes. This is meant to be called when the image or contour plot to which this colorbar belongs is changed.

class matplotlib.colorbar.ColorbarBase(ax, cmap=None, norm=None, alpha=None, values=None, boundaries=None, orientation='vertical', ticklocation='auto', extend='neither', spacing='uniform', ticks=None, format=None, drawedges=False, filled=True, extendfrac=None, extendrect=False, label='')

Bases: matplotlib.cm.ScalarMappable

Draw a colorbar in an existing axes.

This is a base class for the Colorbar class, which is the basis for the colorbar() function and the colorbar() method, which are the usual ways of creating a colorbar.

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 list of LineCollection if lines were drawn, otherwise an empty list
    dividers a LineCollection if drawedges is True, otherwise None

Useful public methods are set_label() and add_lines().

add_lines(levels, colors, linewidths, erase=True)

Draw lines on the colorbar.

    colors and linewidths must be scalars or sequences the same length as levels.

    Set erase to False to add lines without first removing any previously added lines.

ax = None

The axes that this colorbar lives in.

config_axis()

draw_all()

Calculate any free parameters based on the current cmap and norm, and do all the drawing.

set_alpha(alpha)

set_label(label, **kw)

Label the long axis of the colorbar
**set_ticklabels(ticklabels, update_ticks=True)**

Set tick labels. Tick labels are updated immediately unless `update_ticks` is `False`. To manually update the ticks, call `update_ticks` method explicitly.

**set_ticks(ticks, update_ticks=True)**

Set tick locations. Tick locations are updated immediately unless `update_ticks` is `False`. To manually update the ticks, call `update_ticks` method explicitly.

**update_ticks()**

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 which is created using `Patch` rather than the default `pcolor()`.

It uses a list of `Patch` instances instead of a `PatchCollection` because the latter does not allow the hatch pattern to vary among the members of the collection.

---

**matplotlib.colorbar.colorbar_factory(cax, mappable, **kwargs)**

Creates a colorbar on the given axes for the given mappable.

Typically, for automatic colorbar placement given only a mappable use `colorbar()`.

**matplotlib.colorbar.make_axes(parents, location=None, orientation=None, fraction=0.15, shrink=1.0, aspect=20, **kw)**

Resize and reposition parent axes, and return a child axes suitable for a colorbar:

```python
cax, kw = make_axes(parent, **kw)
```

Keyword arguments may include the following (with defaults):

- **location** ["'None'" | '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'" | 'vertical' | 'horizontal'] The orientation of the colorbar. Typically, this keyword shouldn’t be used, as it can be derived from the `location` keyword.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>orientation</td>
<td>vertical or horizontal</td>
</tr>
<tr>
<td>fraction</td>
<td>0.15; fraction of original axes to use for colorbar</td>
</tr>
<tr>
<td>pad</td>
<td>0.05 if vertical, 0.15 if horizontal; fraction of original axes between colorbar and new image axes</td>
</tr>
<tr>
<td>shrink</td>
<td>1.0; fraction by which to shrink the colorbar</td>
</tr>
<tr>
<td>aspect</td>
<td>20; ratio of long to short dimensions</td>
</tr>
<tr>
<td>anchor</td>
<td>(0.0, 0.5) if vertical; (0.5, 1.0) if horizontal; the anchor point of the colorbar axes</td>
</tr>
<tr>
<td>panchor</td>
<td>(1.0, 0.5) if vertical; (0.5, 0.0) if horizontal; the anchor point of the colorbar parent axes. If False, the parent axes’ anchor will be unchanged</td>
</tr>
</tbody>
</table>
Returns (cax, kw), the child axes and the reduced kw dictionary to be passed when creating the colorbar instance.

```
matplotlib.colorbar.make_axes_gridspec(parent, **kw)
```

Resize and reposition a parent axes, and return a child axes suitable for a colorbar. 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 subplot parent.
- `make_axes creates an instance of Axes. make_axes_gridspec` creates an instance of Subplot.
- `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:

```
cax, kw = make_axes_gridspec(parent, **kw)
```

Keyword arguments may include the following (with defaults):

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>orientation</code></td>
<td>‘vertical’ or ‘horizontal’</td>
</tr>
<tr>
<td><code>fraction</code></td>
<td>0.15; fraction of original axes to use for colorbar</td>
</tr>
<tr>
<td><code>pad</code></td>
<td>0.05 if vertical, 0.15 if horizontal; fraction of original axes between colorbar and new image axes</td>
</tr>
<tr>
<td><code>shrink</code></td>
<td>1.0; fraction by which to shrink the colorbar</td>
</tr>
<tr>
<td><code>aspect</code></td>
<td>20; ratio of long to short dimensions</td>
</tr>
<tr>
<td><code>anchor</code></td>
<td>(0.0, 0.5) if vertical; (0.5, 1.0) if horizontal; the anchor point of the colorbar axes</td>
</tr>
<tr>
<td><code>pananchor</code></td>
<td>(1.0, 0.5) if vertical; (0.5, 0.0) if horizontal; the anchor point of the colorbar parent axes. If False, the parent axes’ anchor will be unchanged</td>
</tr>
</tbody>
</table>

All but the first of these are stripped from the input kw set.

Returns (cax, kw), the child axes and the reduced kw dictionary to be passed when creating the colorbar instance.
For a visual representation of the matplotlib colormaps, see the “Color” section in the gallery.

### 60.1 matplotlib.colors

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. Colormapping typically involves two steps: a data array is first mapped onto the range 0-1 using an instance of `Normalize` or of a subclass; then this number in the 0-1 range is mapped to a color using an instance of a subclass of `Colormap`. Two are provided here: `LinearSegmentedColormap`, which is used to generate all the built-in colormap instances, but is also useful for making custom colormaps, and `ListedColormap`, which is used for generating a custom colormap from a list of color specifications.

The module also provides a single instance, `colorConverter`, of the `ColorConverter` class providing methods for converting single color specifications or sequences of them to *RGB* or *RGBA*.

Commands which take color arguments can use several formats to specify the colors. For the basic built-in colors, you can use a single letter

- b: blue
- g: green
- r: red
- c: cyan
- m: magenta
- y: yellow
- k: black
- w: white

Gray shades can be given as a string encoding a float in the 0-1 range, e.g.:
color = '0.75'

For a greater range of colors, you have two options. You can specify the color using an html hex string, as in:

color = '#eeefff'

or you can pass an \( R, G, B \) tuple, where each of \( R, G, B \) are in the range \([0,1]\).

Finally, legal html names for colors, like ‘red’, ‘burlywood’ and ‘chartreuse’ are supported.

class matplotlib.colors.BoundaryNorm(boundaries, ncolors, clip=False)
    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.

\( \text{boundaries} \) a monotonically increasing sequence
\( \text{ncolors} \) number of colors in the colormap to be used

If:

\[ b[i] \leq v < b[i+1] \]

then \( v \) is mapped to color \( j \); as \( i \) varies from 0 to len(boundaries)-2, \( j \) goes from 0 to ncolors-1.

Out-of-range values are mapped to -1 if low and ncolors if high; these are converted to valid indices by Colormap.__call__().

\textbf{inverse}(\textit{value})

class matplotlib.colors.ColorConverter
    Bases: object

Provides methods for converting color specifications to \( RGB \) or \( RGBA \)

Caching is used for more efficient conversion upon repeated calls with the same argument.

Ordinarily only the single instance instantiated in this module, colorConverter, is needed.

cache = {'0.8': (0.8, 0.8, 0.8), '0.5': (0.5, 0.5, 0.5), 'yellow': (1.0, 1.0, 0.0), '0.6': (0.6, 0.6, 0.6), 'magenta': (1.0, 0.0, ... 0.0), 'green': (0.0, 0.5019607843137255, 0.0), 'w': (1.0, 1.0, 1.0), 'y': (0.75, 0.75, 0), '0.90': (0.9, 0.9, 0.9)}

colors = {'c': (0.0, 0.75, 0.75), 'b': (0.0, 0.0, 1.0), 'w': (1.0, 1.0, 1.0), 'g': (0.0, 0.5, 0.0), 'y': (0.75, 0.75, 0), 'k': (0.0, 0.0, 0.0), 'r': (1.0, 0.0, 0.0)}

to_rgb(\textit{arg})

Returns an \( RGB \) tuple of three floats from 0-1.

\textit{arg} can be an \( RGB \) or \( RGBA \) sequence or a string in any of several forms:
1. a letter from the set ‘rgbcmykw’
2. A hex color string, like '#00FFFF'
3. A standard name, like 'aqua'
4. A float, like '0.4', indicating gray on a 0-1 scale

if arg is RGBA, the A will simply be discarded.

to_rgba(arg, alpha=None)
Returns an RGBA tuple of four floats from 0-1.

For acceptable values of arg, see to_rgb(). In addition, if arg is “none” (case-insensitive), then (0,0,0,0) will be returned. If arg is an RGBA sequence and alpha is not None, alpha will replace the original A.

to_rgba_array(c, alpha=None)
Returns a numpy array of RGBA tuples.

Accepts a single mpl color spec or a sequence of specs.

Special case to handle “no color”: if c is “none” (case-insensitive), then an empty array will be returned. Same for an empty list.

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. It is worth noting that matplotlib.cm.ScalarMappable subclasses 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.

colorbar_extend = None
    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()

set_bad(color='k', alpha=None)
Set color to be used for masked values.

set_over(color='k', alpha=None)
Set color to be used for high out-of-range values. Requires norm.clip = False

set_under(color='k', alpha=None)
Set color to be used for low out-of-range values. Requires norm.clip = False

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. The `shade()` is used to produce rgb values for a shaded relief image given a data array.

Specify the azimuth (measured clockwise from south) and altitude (measured up from the plane of the surface) of the light source in degrees.

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).

```
shade(data, cmap)
```

Take the input data array, convert to HSV values in the given colormap, then adjust those color values to given 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.

```
shade_rgb(rgb, elevation, fraction=1.0)
```

Take the input RGB array (ny*nx*3) adjust their color values to given the impression of a shaded relief map with a specified light source using the elevation (ny*nx). A new RGB array ((ny*nx*3)) is returned.

```
class matplotlib.colors.LinearSegmentedColormap(name,  
segmentdata,  
N=256,  
gamma=1.0)
```

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 \(x, y_0, y_1\) 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 \(y_1[i]\) and \(y_0[i+1]\):

\[
\begin{align*}
\text{row } i & : \quad x \quad y_0 \quad y_1 \\
\text{row } i+1 & : \quad x \quad y_0 \quad y_1
\end{align*}
\]

Hence \(y_0\) in the first row and \(y_1\) 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.

```python
static from_list(name, colors, N=256, gamma=1.0)
```

Make a linear segmented colormap with \(name\) from a sequence of \(colors\) which evenly transitions from \(colors[0]\) at \(val=0\) to \(colors[-1]\) at \(val=1\). \(N\) is the number of rgb quantization levels. Alternatively, a list of (value, color) tuples can be given to divide the range unevenly.

```python
set_gamma(gamma)
```

Set a new gamma value and regenerate color map.

### class matplotlib.colors.ListedColormap(colors, name='from_list', N=None)

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.

Make a colormap from a list of colors.

- **colors** a list of matplotlib color specifications, or an equivalent \(N\times3\) or \(N\times4\) floating point array (\(N\) rgb or rgba values)
- **name** a string to identify the colormap
- **\(N\)** the 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:

\[
N < \text{len(colors)}
\]

the list will be truncated at \(N\). If:

\[
N > \text{len(colors)}
\]

the list will be extended by repetition.

### class matplotlib.colors.LogNorm(vmin=None, vmax=None, clip=False)

Normalize a given value to the 0-1 range on a log scale.
If $v_{\text{min}}$ or $v_{\text{max}}$ is not given, they are taken from the input’s minimum and maximum value respectively. If $\text{clip}$ is $\text{True}$ and the given value falls outside the range, the returned value will be 0 or 1, whichever is closer. Returns 0 if:

$v_{\text{min}} = v_{\text{max}}$

Works with scalars or arrays, including masked arrays. If $\text{clip}$ is $\text{True}$, masked values are set to 1; otherwise they remain masked. Clipping silently defeats the purpose of setting the over, under, and masked colors in the colormap, so it is likely to lead to surprises; therefore the default is $\text{clip} = \text{False}$.

**autoscale**($A$)

Set $v_{\text{min}}, v_{\text{max}}$ to min, max of $A$.

**autoscale_None**($A$)

autoscale only None-valued $v_{\text{min}}$ or $v_{\text{max}}$

**inverse**(value)

class matplotlib.colors.NoNorm($v_{\text{min}}=\text{None}, v_{\text{max}}=\text{None}, \text{clip}=False$)

Bases: matplotlib.colors.Normalize

Dummy replacement for Normalize, for the case where we want to use indices directly in a ScalarMappable.

If $v_{\text{min}}$ or $v_{\text{max}}$ is not given, they are taken from the input’s minimum and maximum value respectively. If $\text{clip}$ is $\text{True}$ and the given value falls outside the range, the returned value will be 0 or 1, whichever is closer. Returns 0 if:

$v_{\text{min}} = v_{\text{max}}$

Works with scalars or arrays, including masked arrays. If $\text{clip}$ is $\text{True}$, masked values are set to 1; otherwise they remain masked. Clipping silently defeats the purpose of setting the over, under, and masked colors in the colormap, so it is likely to lead to surprises; therefore the default is $\text{clip} = \text{False}$.

**inverse**(value)

class matplotlib.colors.Normalize($v_{\text{min}}=\text{None}, v_{\text{max}}=\text{None}, \text{clip}=False$)

Bases: object

A class which, when called, can normalize data into the $[0.0, 1.0]$ interval.

If $v_{\text{min}}$ or $v_{\text{max}}$ is not given, they are taken from the input’s minimum and maximum value respectively. If $\text{clip}$ is $\text{True}$ and the given value falls outside the range, the returned value will be 0 or 1, whichever is closer. Returns 0 if:

$v_{\text{min}} = v_{\text{max}}$

Works with scalars or arrays, including masked arrays. If $\text{clip}$ is $\text{True}$, masked values are set to 1; otherwise they remain masked. Clipping silently defeats the purpose of setting the over, under, and
masked colors in the colormap, so it is likely to lead to surprises; therefore the default is \( \text{clip} = \text{False} \).

\[ \text{autoscale}(A) \]
Set \( vmin, vmax \) to min, max of \( A \).

\[ \text{autoscale\_None}(A) \]
autoscale only None-valued \( vmin \) or \( vmax \)

\[ \text{inverse}(value) \]

\[ \text{static process\_value}(value) \]
Homogenize the input \( value \) for easy and efficient normalization.

\[ value \] can be a scalar or sequence.

Returns \( result, is\_scalar \), where \( result \) is a masked array matching \( value \). Float dtypes are preserved; integer types with two bytes or smaller are converted to np.float32, and larger types are converted to np.float. Preserving float32 when possible, and using in-place operations, can greatly improve speed for large arrays.

Experimental; we may want to add an option to force the use of float32.

\[ \text{scaled()} \]
return true if \( vmin \) and \( vmax \) set

\[ \text{class matplotlib.colors.SymLogNorm}(\text{linthresh}, \text{linscale}=1.0, \text{vmin}=\text{None}, \text{vmax}=\text{None}, \text{clip}=\text{False}) \]

Bases: \text{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 \( \text{linthresh} \) allows the user to specify the size of this range (-\( \text{linthresh} \), \( \text{linthresh} \)).

\( \text{linthresh} \): The range within which the plot is linear (to avoid having the plot go to infinity around zero).

\( \text{linscale} \): This allows the linear range (-\( \text{linthresh} \) to \( \text{linthresh} \)) 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 \( \text{linscale} = 1.0 \) (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. Defaults to 1.

\[ \text{autoscale}(A) \]
Set \( vmin, vmax \) to min, max of \( A \).

\[ \text{autoscale\_None}(A) \]
autoscale only None-valued \( vmin \) or \( vmax \)

\[ \text{inverse}(value) \]

\[ \text{matplotlib.colors.from\_levels\_and\_colors}(\text{levels}, \text{colors}, \text{extend}='\text{neither}') \]

A helper routine to generate a cmap and a norm instance which behave similar to contourf’s levels
Matplotlib, Release 1.3.0

and colors arguments.

**Parameters**

- **levels**: sequence of numbers
  
The quantization levels used to construct the `BoundaryNorm`. Values $v$ are quantized to level $i$ if $\text{lev}[i] \leq v < \text{lev}[i+1]$.

- **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, norm)`: tuple containing a `Colormap` and a `Normalize` instance

```python
matplotlib.colors.hex2color(s)
```

Take a hex string `s` and return the corresponding rgb 3-tuple Example: `#efefef` -> `(0.93725, 0.93725, 0.93725)`

```python
matplotlib.colors.hsv_to_rgb(hsv)
```

Convert hsv values in a numpy array to rgb values both input and output arrays have shape `(M,N,3)`

```python
matplotlib.colors.is_color_like(c)
```

Return `True` if `c` can be converted to `RGB`

```python
matplotlib.colors.makeMappingArray(N, data, gamma=1.0)
```

Create an $N$-element 1-d lookup table

`data` represented by a list of $x,y0,y1$ mapping correspondences. Each element in this list represents how a value between 0 and 1 (inclusive) represented by $x$ is mapped to a corresponding value between 0 and 1 (inclusive). The two values of $y$ are to allow for discontinuous mapping functions (say as might be found in a sawtooth) where $y0$ represents the value of $y$ for values of $x$ <= to that given, and $y1$ is the value to be used for $x$ > than that given). The list must start with $x=0$, end with $x=1$, and all values of $x$ must be in increasing order. Values between the given mapping points are determined by simple linear interpolation.

Alternatively, data can be a function mapping values between 0 - 1 to 0 - 1.

The function returns an array “result” where `result[x*(N-1)]` gives the closest value for values of $x$ between 0 and 1.

```python
matplotlib.colors.no_norm(*args, **kwargs)
```

Deprecated since version 1.3: The no_norm class alias was deprecated in version 1.3. Use NoNorm instead.

Dummy replacement for Normalize, for the case where we want to use indices directly in a `ScalarMappable`.

```python
matplotlib.colors.normalize(*args, **kwargs)
```

Deprecated since version 1.3: The normalize class alias was deprecated in version 1.3. Use Normalize instead.

A class which, when called, can normalize data into the $[0.0, 1.0]$ interval.

```python
matplotlib.colors.rgb2hex(rgb)
```

Given an rgb or rgba sequence of 0-1 floats, return the hex string
**matplotlib.colors.rgb_to_hsv(arr)**

convert rgb values in a numpy array to hsv values input and output arrays should have shape (M,N,3)
61.1 matplotlib.dates

Matplotlib provides sophisticated date plotting capabilities, standing on the shoulders of Python datetime, the add-on modules pytz and dateutil. datetime objects are converted to floating point numbers which represent time in days since 0001-01-01 UTC, plus 1. For example, 0001-01-01, 06:00 is 1.25, not 0.25. The helper functions date2num(), num2date() and drange() are used to facilitate easy conversion to and from datetime and numeric ranges.

Note: Like Python’s datetime, mpl 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 mpl 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 [31]: date(2006,4,1).toordinal() - date(1,1,1).toordinal()
Out[31]: 732401
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.

All the matplotlib date converters, tickers and formatters are timezone aware, and the default timezone is given by the timezone parameter in your matplotlibrc file. If you leave out a tz timezone instance, the default from your rc file will be assumed. If you want to use a custom time zone, pass a pytz.timezone instance with the tz keyword argument to num2date(), plot_date(), and any custom date tickers or locators you create. See pytz for information on pytz and timezone handling.

The dateutil module provides additional code to handle date ticking, making it easy to place ticks on any kinds of dates. See examples below.

### 61.1.1 Date tickers

Most of the date tickers can locate single or multiple values. For example:

```python
# 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:

```python
# tick on mondays every second week
loc = WeekdayLocator(byweekday=MO, interval=2)
```

The rrule locator allows completely general date ticking:

```python
# tick every 5th easter
rule = rrulewrapper(YEARLY, byeaster=1, interval=5)
loc = RRuleLocator(rule)
```

Here are all the date tickers:

- **MinuteLocator**: locate minutes
- **HourLocator**: locate hours
- **DayLocator**: locate specified days of the month
- **WeekdayLocator**: locate days of the week, eg MO, TU
- **MonthLocator**: locate months, eg 7 for july
- **YearLocator**: locate years that are multiples of base
- **RRuleLocator**: locate using a matplotlib.dates.rrulewrapper. The rrulewrapper is a simple wrapper around a dateutil.rrule (dateutil) which allow almost arbitrary date tick specifications. See rrule example.
• **AutoDateLocator**: On autoscale, this class picks the best **MultipleDateLocator** to set the view limits and the tick locations.

### 61.1.2 Date formatters

Here all all the date formatters:

- **AutoDateFormatter**: attempts to figure out the best format to use. This is most useful when used with the **AutoDateLocator**.
- **DateFormatter**: use `strftime()` format strings
- **IndexDateFormatter**: date plots with implicit $x$ indexing.

```python
def date2num(d)
    d is either a `datetime` instance or a sequence of datetimes.
    Return value is a floating point number (or sequence of floats) which gives the number of days (fraction part represents hours, minutes, seconds) since 0001-01-01 00:00:00 UTC, plus one. 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.

def num2date(x, tz=None)
    x is a float value which gives the number of days (fraction part represents hours, minutes, seconds) since 0001-01-01 00:00:00 UTC plus one. 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.
    If $x$ is a sequence, a sequence of `datetime` objects will be returned.

def drange(dstart, dend, delta)
    Return a date range as float Gregorian ordinals. `dstart` and `dend` are `datetime` instances. `delta` is a `datetime.timedelta` instance.

def epoch2num(e)
    Convert an epoch or sequence of epochs to the new date format, that is days since 0001.

def num2epoch(d)
    Convert days since 0001 to epoch. $d$ can be a number or sequence.

def mx2num(mxdates)
    Convert mx `datetime` instance (or sequence of mx instances) to the new date format.
```

```python
class DateFormatter(fmt, tz=None)
    Bases: matplotlib.ticker.Formatter

    Tick location is seconds since the epoch. Use a `strftime()` format string.
    Python only supports `datetime strftime()` formatting for years greater than 1900. Thanks to Andrew Dalke, Dalke Scientific Software who contributed the `strftime()` code below to include dates earlier than this year.
    `fmt` is an `strftime()` format string; `tz` is the `tzinfo` instance.
illegal_s = <_sre.SRE_Pattern object at 0x280b820>

set_tzinfo(tz)

strftime(dt, fmt)

class matplotlib.dates.IndexDateFormatter(t, fmt, tz=None)
   Bases: matplotlib.ticker.Formatter

   Use with IndexLocator to cycle format strings by index.
   t is a sequence of dates (floating point days). fmt is a strftime() format string.

class matplotlib.dates.AutoDateFormatter(locator, tz=None, defaultfmt='%Y-%m-%d')
   Bases: matplotlib.ticker.Formatter

   This class 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 = {
      365.0 : '%Y',
      30. : '%b %Y',
      1.0 : '%b %d %Y',
      1./24. : '%H:%M:%D',
      1. / (24. * 60.): '%H:%M:%S.%f',
   }

   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:

   formatter = AutoDateFormatter()
   formatter.scaled[1./(24.*60.)] = '%H:%M' # only show min and sec

   Autofmft the date labels. The default format is the one to use if none of the times in scaled match

class matplotlib.dates.DateLocator(tz=None)
   Bases: matplotlib.ticker.Locator

   tz is a tzinfo instance.

   datalim_to_dt()

   hms0d = {'byminute': 0, 'byhour': 0, 'bysecond': 0}

   nonsingular(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(tz)

viewlim_to_dt()

class matplotlib.dates.RRuleLocator(o, tz=None)
    Bases: matplotlib.dates.DateLocator
    autoscale()
        Set the view limits to include the data range.

static get_unit_generic(freq)

class matplotlib.dates.AutoDateLocator(tz=None, minticks=5, maxticks=None, interval_multiples=False)
    Bases: matplotlib.dates.DateLocator
    On autoscale, this class picks the best DateLocator to set the view limits and the tick locations.

minticks is the minimum number of ticks desired, which is used to select the type of ticking (yearly, monthly, etc.).

maxticks is the maximum number of ticks desired, which controls any interval between ticks (ticking every other, every 3, etc.). For really 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.

tz is a tzinfo instance.

interval_multiples is a boolean that indicates whether ticks should be chosen to be multiple of the interval. This will lock ticks 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.

The AutoDateLocator has an interval dictionary that maps the frequency of the tick (a constant from dateutil.rrule) and a multiple allowed for that ticking. The default looks like this:

```
self.interval = {
    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],
    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],
}
```

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. You can customize this dictionary by doing:
locator = AutoDateLocator()
locator.interval[HOURLY] = [3] # only show every 3 hours

autoscale()  
Try to choose the view limits intelligently.

get_locator(dmin, dmax)  
Pick the best locator based on a distance.

nonsingular(vmin, vmax)

refresh()  
Refresh internal information based on current limits.

set_axis(axis)

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()  
Set the view limits to include the data range.

class matplotlib.dates.MonthLocator(bymonth=None, bymonthday=1, interval=1, tz=None)  
Bases: matplotlib.dates.RRuleLocator
Make ticks on occurances of each month month, eg 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.WeekdayLocator(byweekday=1, interval=1, tz=None)  
Bases: matplotlib.dates.RRuleLocator
Make ticks on occurances 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.
interval specifies the number of weeks to skip. For example, interval=2 plots every second week.

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.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.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.MicrosecondLocator(interval=1, tz=None)
   Bases: matplotlib.dates.DateLocator
   Make ticks on occurrences of each microsecond.

   interval is the interval between each iteration. For example, if interval=2, mark every second microsecond.

   set_axis(axis)

   set_data_interval(vmin, vmax)
set_view_interval($vmin, vmax$)

class matplotlib.dates.rrule($freq$, $dtstart=None$, $interval=1$, $wkst=None$, $count=None$, $until=None$, $bysetpos=None$, $bymonth=None$, $bymonthday=None$, $byyearday=None$, $byeaster=None$, $byweek=None$, $byweekday=None$, $byhour=None$, $byminute=None$, $bysecond=None$, $until=None$, $bysetpos=None$, $bymonth=None$, $bymonthday=None$, $byyearday=None$, $byeaster=None$, $byweek=None$, $byweekday=None$, $byhour=None$, $byminute=None$, $bysecond=None$, $cache=False$)

Bases: dateutil.rrule.rrulebase

class matplotlib.dates.relativedelta($dt1=None$, $dt2=None$, $years=0$, $months=0$, $days=0$, $leapdays=0$, $weeks=0$, $hours=0$, $minutes=0$, $seconds=0$, $microseconds=0$, $year=None$, $month=None$, $day=None$, $weekday=None$, $yearday=None$, $nlyearday=None$, $nlyearday=None$, $hour=None$, $minute=None$, $second=None$, $microsecond=None$)

The relativedelta type 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’s two different ways to build a relativedelta instance. The first one is passing it two date/datetime classes:

relativedelta(datetime1, datetime2)

And the other way is to use the following keyword arguments:

- **year, month, day, hour, minute, second, microsecond**: Absolute information.
- **years, months, weeks, days, hours, minutes, seconds, microseconds**: Relative information, may be negative.
- **weekday**: One of the weekday instances (MO, TU, etc). 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.
- **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.

Here is the behavior of operations with relativedelta:

1. Calculate the absolute year, using the ‘year’ argument, or the original datetime year, if the argument is not present.
2. Add the relative ‘years’ argument to the absolute year.
3. Do steps 1 and 2 for month/months.
4. Calculate the absolute day, using the ‘day’ argument, or the original datetime day, if the argument is not present. Then, subtract from the day until it fits in the year and month found after their operations.
5. Add the relative ‘days’ argument to the absolute day. Notice that the ‘weeks’ argument is multiplied by 7 and added to ‘days’.
6. Do steps 1 and 2 for hour/hours, minute/minutes, second/seconds, microsecond/microseconds.
7. If the ‘weekday’ argument is present, calculate the weekday, with the given (wday, nth) tuple. wday is the index of the weekday (0-6, 0=Mon), and nth is the number of weeks to add forward or backward, depending on its signal. Notice that if the calculated date is already Monday, for example, using (0, 1) or (0, -1) won’t change the day.
matplotlib.dates.\texttt{seconds}(s)
   \hspace{1em} Return seconds as days.

matplotlib.dates.\texttt{minutes}(m)
   \hspace{1em} Return minutes as days.

matplotlib.dates.\texttt{hours}(h)
   \hspace{1em} Return hours as days.

matplotlib.dates.\texttt{weeks}(w)
   \hspace{1em} Return weeks as days.
62.1 `matplotlib.figure`

The figure module provides the top-level `Artist`, the `Figure`, which contains all the plot elements. The following classes are defined

`SubplotParams` control the default spacing of the subplots

`Figure` top level container for all plot elements

```python
class matplotlib.figure.AxesStack
    Bases: matplotlib.cbook.Stack

    Specialization of the Stack to handle all tracking of Axes in a Figure. This stack stores key, (ind, axes) pairs, where:
    • key should be a hash of the args and kwargs used in generating the Axes.
    • ind is a serial number for tracking the order in which axes were added.
    The AxesStack is a callable, where `ax_stack()` returns the current axes. Alternatively the `current_key_axes()` will return the current key and associated axes.

    `add(key, a)`
    Add Axes `a`, with key `key`, to the stack, and return the stack.
    If `a` is already on the stack, don’t add it again, but return `None`.

    `as_list()`
    Return a list of the Axes instances that have been added to the figure

    `bubble(a)`
    Move the given axes, which must already exist in the stack, to the top.

    `current_key_axes()`
    Return a tuple of (key, axes) for the active axes.
    If no axes exists on the stack, then returns (None, None).

    `get(key)`
    Return the Axes instance that was added with `key`. If it is not present, return `None`.

    `remove(a)`
    Remove the axes from the stack.
```
class `matplotlib.figure.Figure`(
    figsize=None, dpi=None, facecolor=None, edgecolor=None,
    linewidth=0.0, frameon=None, subplotpars=None,
    tight_layout=None
)

Bases: `matplotlib.artist.Artist`

The Figure instance supports callbacks through a `callbacks` attribute which is a `matplotlib.cbook.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.

`patch` The figure patch is drawn by a `matplotlib.patches.Rectangle` instance

`suppressComposite` For multiple figure images, the figure will make composite images depending on the renderer option `image_nocomposite` function. If `suppressComposite` is True|False, this will override the renderer.

`figsize` w,h tuple in inches

`dpi` Dots per inch

`facecolor` The figure patch facecolor; defaults to rc `figure.facecolor`

`edgecolor` The figure patch edge color; defaults to rc `figure.edgecolor`

`linewidth` The figure patch edge linewidth; the default linewidth of the frame

`frameon` If False, suppress drawing the figure frame

`subplotpars` A `SubplotParams` instance, defaults to rc

`tight_layout` 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. Defaults to rc `figure.autolayout`.

`add_axes`(*args, **kwargs)

Add an axes at position `rect` [left, bottom, width, height] where all quantities are in fractions of figure width and height. kwangs are legal `Axes` kwangs plus `projection` which sets the projection type of the axes. (For backward compatibility, `polar=True` may also be provided, which is equivalent to `projection='polar'`). Valid values for `projection` are: ['aitoff', 'hammer', 'lambert', 'mollweide', 'polar', 'rectlinear']. Some of these projections support additional kwangs, which may be provided to `add_axes()`. Typical usage:

```python
rect = 1,b,w,h
fig.add_axes(rect)
fig.add_axes(rect, frameon=False, axisbg='g')
fig.add_axes(rect, polar=True)
fig.add_axes(rect, projection='polar')
fig.add_axes(ax)
```

If the figure already has an axes with the same parameters, then it will simply make that axes current and return it. If you do not want this behavior, e.g., you want to force the creation of a new Axes, you must use a unique set of args and kwangs. The axes `label` attribute has been exposed for this purpose. e.g., if you want two axes that are otherwise identical to be added to the figure, make sure you give them unique labels:

```python
fig.add_axes(rect, label='axes1')
fig.add_axes(rect, label='axes2')
```

In rare circumstances, `add_axes` may be called with a single argument, an Axes instance already created in the present figure but not in the figure’s list of axes. For example, if an axes has been removed with `delaxes()`, it can be restored with:
fig.add_axes(ax)

In all cases, the Axes instance will be returned.

In addition to projection, the following kwargs are supported:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>adjustable</td>
<td>['box'</td>
</tr>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>anchor</td>
<td>unknown</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>aspect</td>
<td>unknown</td>
</tr>
<tr>
<td>autoscale_on</td>
<td>unknown</td>
</tr>
<tr>
<td>autoscalex_on</td>
<td>unknown</td>
</tr>
<tr>
<td>autoscaley_on</td>
<td>unknown</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>axes_locator</td>
<td>unknown</td>
</tr>
<tr>
<td>axisbgcolor</td>
<td>any matplotlib color - see colors()</td>
</tr>
<tr>
<td>axis_off</td>
<td>unknown</td>
</tr>
<tr>
<td>axis_on</td>
<td>unknown</td>
</tr>
<tr>
<td>axisbelow</td>
<td>[ True</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path,Transform)</td>
</tr>
<tr>
<td>color_cycle</td>
<td>unknown</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>cursor_props</td>
<td>a (float, color) tuple</td>
</tr>
<tr>
<td>figure</td>
<td>unknown</td>
</tr>
<tr>
<td>frame_on</td>
<td>[ True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>navigate</td>
<td>[ True</td>
</tr>
<tr>
<td>navigate_mode</td>
<td>unknown</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>position</td>
<td>unknown</td>
</tr>
<tr>
<td>rasterization_zorder</td>
<td>unknown</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>title</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
</tbody>
</table>

Continued on next page
Table 62.1 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xbound</td>
<td>unknown</td>
</tr>
<tr>
<td>xlabel</td>
<td>unknown</td>
</tr>
<tr>
<td>xlim</td>
<td>length 2 sequence of floats</td>
</tr>
<tr>
<td>xmargin</td>
<td>unknown</td>
</tr>
<tr>
<td>xscale</td>
<td>['linear'</td>
</tr>
<tr>
<td>xticklabels</td>
<td>sequence of strings</td>
</tr>
<tr>
<td>xticks</td>
<td>sequence of floats</td>
</tr>
<tr>
<td>ybound</td>
<td>unknown</td>
</tr>
<tr>
<td>ylabel</td>
<td>unknown</td>
</tr>
<tr>
<td>ylim</td>
<td>length 2 sequence of floats</td>
</tr>
<tr>
<td>ymargin</td>
<td>unknown</td>
</tr>
<tr>
<td>yscale</td>
<td>['linear'</td>
</tr>
<tr>
<td>yticklabels</td>
<td>sequence of strings</td>
</tr>
<tr>
<td>yticks</td>
<td>sequence of floats</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

**add_axobserver(func)**

Whenever the axes state change, `func(self)` will be called.

**add_subplot(*args, **kwargs)**

Add a subplot. Examples:

```
fig.add_subplot(111)
```

# equivalent but more general
```
fig.add_subplot(1,1,1)
```

# add subplot with red background
```
fig.add_subplot(212, axisbg='r')
```

# add a polar subplot
```
fig.add_subplot(111, projection='polar')
```

# add Subplot instance sub
```
fig.add_subplot(sub)
```

`kwargs` are legal `Axes` kwargs plus `projection`, which chooses a projection type for the axes. (For backward compatibility, `polar=True` may also be provided, which is equivalent to `projection='polar'`). Valid values for `projection` are: ['aitoff', 'hammer', 'lambert', 'mollweide', 'polar', 'rectilinear']. Some of these projections support additional `kwargs`, which may be provided to `add_axes()`.

The `Axes` instance will be returned.

If the figure already has a subplot with key (`args, kwargs`) then it will simply make that subplot current and return it.
The following kwargs are supported:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>adjustable</td>
<td>['box', 'datalim', 'box-forced']</td>
</tr>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>anchor</td>
<td>unknown</td>
</tr>
<tr>
<td>animated</td>
<td>[True, False]</td>
</tr>
<tr>
<td>aspect</td>
<td>unknown</td>
</tr>
<tr>
<td>autoscale_on</td>
<td>unknown</td>
</tr>
<tr>
<td>autoscalex_on</td>
<td>unknown</td>
</tr>
<tr>
<td>autoscaley_on</td>
<td>unknown</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>axes_locator</td>
<td>unknown</td>
</tr>
<tr>
<td>axis_bgcolor</td>
<td>any matplotlib color - see colors()</td>
</tr>
<tr>
<td>axis_off</td>
<td>unknown</td>
</tr>
<tr>
<td>axis_on</td>
<td>unknown</td>
</tr>
<tr>
<td>axisbelow</td>
<td>[True, False]</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True, False]</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform), Patch, None]</td>
</tr>
<tr>
<td>color_cycle</td>
<td>unknown</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>cursor_props</td>
<td>a (float, color) tuple</td>
</tr>
<tr>
<td>figure</td>
<td>unknown</td>
</tr>
<tr>
<td>frame_on</td>
<td>[True, False]</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>lod</td>
<td>[True, False]</td>
</tr>
<tr>
<td>navigate</td>
<td>[True, False]</td>
</tr>
<tr>
<td>navigate_mode</td>
<td>unknown</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None, float, boolean, callable]</td>
</tr>
<tr>
<td>position</td>
<td>unknown</td>
</tr>
<tr>
<td>rasterization_zorder</td>
<td>unknown</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True, False, None]</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>title</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True, False]</td>
</tr>
<tr>
<td>xbound</td>
<td>unknown</td>
</tr>
<tr>
<td>xlabel</td>
<td>unknown</td>
</tr>
<tr>
<td>xlim</td>
<td>length 2 sequence of floats</td>
</tr>
<tr>
<td>xmargin</td>
<td>unknown</td>
</tr>
</tbody>
</table>

Continued on next page
Table 62.2 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xscale</td>
<td>['linear', 'log', 'symlog']</td>
</tr>
<tr>
<td>xticklabels</td>
<td>sequence of strings</td>
</tr>
<tr>
<td>xticks</td>
<td>sequence of floats</td>
</tr>
<tr>
<td>ybound</td>
<td>unknown</td>
</tr>
<tr>
<td>ylabel</td>
<td>unknown</td>
</tr>
<tr>
<td>ylim</td>
<td>length 2 sequence of floats</td>
</tr>
<tr>
<td>ymargin</td>
<td>unknown</td>
</tr>
<tr>
<td>yscale</td>
<td>['linear', 'log', 'symlog']</td>
</tr>
<tr>
<td>yticklabels</td>
<td>sequence of strings</td>
</tr>
<tr>
<td>yticks</td>
<td>sequence of floats</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

**autofmt_xdate**(bottom=0.2, rotation=30, ha='right')

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.

- **bottom** The bottom of the subplots for subplots_adjust()
- **rotation** The rotation of the xtick labels
- **ha** The horizontal alignment of the xticklabels

**axes**
Read-only: list of axes in Figure

**clear()**
Clear the figure – synonym for clf().

**clf(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**(mappable, cax=None, ax=None, use_gridspec=True, **kw)
Create a colorbar for a ScalarMappable instance, mappable.

Documentation for the pylab 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(s) 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** False | 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 grid_spec module.

Additional keyword arguments are of two kinds:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>orientation</strong></td>
<td>vertical or horizontal</td>
</tr>
<tr>
<td><strong>fraction</strong></td>
<td>0.15; fraction of original axes to use for colorbar</td>
</tr>
<tr>
<td><strong>pad</strong></td>
<td>0.05 if vertical, 0.15 if horizontal; fraction of original axes between colorbar and new image axes</td>
</tr>
<tr>
<td><strong>shrink</strong></td>
<td>1.0; fraction by which to shrink the colorbar</td>
</tr>
<tr>
<td><strong>aspect</strong></td>
<td>20; ratio of long to short dimensions</td>
</tr>
<tr>
<td><strong>anchor</strong></td>
<td>(0.0, 0.5) if vertical; (0.5, 1.0) if horizontal; the anchor point of the colorbar axes</td>
</tr>
<tr>
<td><strong>panchor</strong></td>
<td>(1.0, 0.5) if vertical; (0.5, 0.0) if horizontal; the anchor point of the colorbar parent axes. If False, the parent axes’ anchor will be unchanged</td>
</tr>
</tbody>
</table>
Matplotlib, Release 1.3.0

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>extend</td>
<td>[‘neither’</td>
</tr>
<tr>
<td>extendfrac</td>
<td>[None</td>
</tr>
<tr>
<td>extendrect</td>
<td>[False</td>
</tr>
<tr>
<td>spacing</td>
<td>[‘uniform’</td>
</tr>
<tr>
<td>ticks</td>
<td>[None</td>
</tr>
<tr>
<td>format</td>
<td>[None</td>
</tr>
<tr>
<td>drawedges</td>
<td>[False</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boundaries</td>
<td>None or a sequence</td>
</tr>
<tr>
<td>values</td>
<td>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.</td>
</tr>
</tbody>
</table>

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 \textit{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 \texttt{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:

\begin{verbatim}
cbar = colorbar()
cbar.solids.set_edgecolor("face")
draw()
\end{verbatim}

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).

\textbf{returns}: \texttt{Colorbar} instance; see also its base class, \texttt{ColorbarBase}. Call the \texttt{set_label()} method to label the colorbar.

\textbf{contains}(\texttt{mouseevent})

Test whether the mouse event occurred on the figure.

Returns True,{} 

\textbf{delaxes}(\texttt{a})

remove \texttt{a} from the figure and update the current axes

\textbf{dpi}

\textbf{draw}(\texttt{artist, renderer, *args, **kwargs})

Render the figure using \texttt{matplotlib.backend_bases.RendererBase} instance \texttt{renderer}.

\textbf{draw_artist}(\texttt{a})

\texttt{draw} \texttt{matplotlib.artist.Artist} instance \texttt{a} only – this is available only after the figure is drawn

\textbf{figimage}(\texttt{X, xo=0, yo=0, alpha=None, norm=None, cmap=None, vmin=None, vmax=None, origin=None, **kwargs})

Adds a non-resampled image to the figure.

call signatures:

\begin{verbatim}
figimage(X, **kwargs)
\end{verbatim}

adds a non-resampled array \texttt{X} to the figure.

\begin{verbatim}
figimage(X, xo, yo)
\end{verbatim}

with pixel offsets \texttt{xo, yo}.

\texttt{X} must be a float array:
- If \( X \) is \( M \times N \), assume luminance (grayscale)
- If \( X \) is \( M \times N \times 3 \), assume RGB
- If \( X \) is \( M \times N \times 4 \), assume RGBA

Optional keyword arguments:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( xo ) or ( yo )</td>
<td>An integer, the ( x ) and ( y ) image offset in pixels</td>
</tr>
<tr>
<td>cmap</td>
<td>a <code>matplotlib.colors.Colormap</code> instance, e.g. <code>cm.jet</code>. If <code>None</code>, default to the rc <code>image.cmap</code> value</td>
</tr>
<tr>
<td>norm</td>
<td>a <code>matplotlib.colors.Normalize</code> instance. The default is <code>normalization()</code>. This scales luminance ( \rightarrow 0 ) to ( 1 )</td>
</tr>
<tr>
<td>vmin</td>
<td>used to scale a luminance image to 0-1. If either is <code>None</code>, the min and max of the luminance values will be used. Note if you pass a norm instance, the settings for <code>vmin</code> and <code>vmax</code> will be ignored.</td>
</tr>
<tr>
<td>alpha</td>
<td>the alpha blending value, default is <code>None</code></td>
</tr>
<tr>
<td>origin</td>
<td>[‘upper’</td>
</tr>
</tbody>
</table>

`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 size \([0,1,0,1]\).

An `matplotlib.image.FigureImage` instance is returned.
Additional kwargs are Artist kwargs passed on to `FigureImage`

```python
(gca(**kwargs))
```

Return 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:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>adjustable</code></td>
<td>[‘box’</td>
</tr>
<tr>
<td><code>agg_filter</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>alpha</code></td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td><code>anchor</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>animated</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>aspect</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>autoscale_on</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>autoscalex_on</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>autoscaley_on</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>axes</code></td>
<td>an <code>Axes</code> instance</td>
</tr>
<tr>
<td><code>axes_locator</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>axis_bgcolor</code></td>
<td>any matplotlib color - see <code>colors()</code></td>
</tr>
<tr>
<td><code>axis_off</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>axis_on</code></td>
<td>unknown</td>
</tr>
</tbody>
</table>

Continued on next page
Table 62.3 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>axisbelow</td>
<td>[True</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color_cycle</td>
<td>unknown</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>cursor_props</td>
<td>a (float, color) tuple</td>
</tr>
<tr>
<td>figure</td>
<td>unknown</td>
</tr>
<tr>
<td>frame_on</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>navigate</td>
<td>[True</td>
</tr>
<tr>
<td>navigate_mode</td>
<td>unknown</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>position</td>
<td>unknown</td>
</tr>
<tr>
<td>rasterization_zorder</td>
<td>unknown</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>title</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xbound</td>
<td>unknown</td>
</tr>
<tr>
<td>xlabel</td>
<td>unknown</td>
</tr>
<tr>
<td>xlim</td>
<td>length 2 sequence of floats</td>
</tr>
<tr>
<td>xmargin</td>
<td>unknown</td>
</tr>
<tr>
<td>xscale</td>
<td>['linear'</td>
</tr>
<tr>
<td>xticklabels</td>
<td>sequence of strings</td>
</tr>
<tr>
<td>xticks</td>
<td>sequence of floats</td>
</tr>
<tr>
<td>ybound</td>
<td>unknown</td>
</tr>
<tr>
<td>ylabel</td>
<td>unknown</td>
</tr>
<tr>
<td>ylim</td>
<td>length 2 sequence of floats</td>
</tr>
<tr>
<td>ymargin</td>
<td>unknown</td>
</tr>
<tr>
<td>yscale</td>
<td>['linear'</td>
</tr>
<tr>
<td>yticklabels</td>
<td>sequence of strings</td>
</tr>
<tr>
<td>yticks</td>
<td>sequence of floats</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

get_axes()
get_children()
get a list of artists contained in the figure

get_default_bbox_extra_artists()

get_dpi()
Return the dpi as a float

get_edgecolor()
Get the edge color of the Figure rectangle

get_facecolor()
Get the face color of the Figure rectangle

get_figheight()
Return the figheight as a float

get_figwidth()
Return the figwidth as a float

get_frameon()
get the boolean indicating frameon

get_size_inches()

get_tight_layout()
Return the Boolean flag, True to use :meth:'tight_layout' when drawing.

get_tightbbox(renderer)
Return a (tight) bounding box of the figure in inches.

It only accounts axes title, axis labels, and axis ticklabels. Needs improvement.

get_window_extent(*args, **kwargs)
get the figure bounding box in display space; kwargs are void

ginput(n=1, timeout=30, show_clicks=True, mouse_add=1, mouse_pop=3, mouse_stop=2)
Call signature:

ginput(self, n=1, timeout=30, show_clicks=True, mouse_add=1, mouse_pop=3, mouse_stop=2)

Blocking call to interact with the figure.
This will wait for n clicks from the user and return a list of the coordinates of each click.
If timeout is zero or negative, does not timeout.
If n is zero or negative, accumulate clicks until a middle click (or potentially both mouse buttons at once) terminates the input.
Right clicking cancels last input.
The buttons used for the various actions (adding points, removing points, terminating the inputs) can be overridden via the arguments `mouse_add`, `mouse_pop` and `mouse_stop`, that give the associated mouse button: 1 for left, 2 for middle, 3 for right.

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.

```python
hold(b=None)
```

Set the hold state. If hold is None (default), toggle the hold state. Else set the hold state to boolean value b.

e.g.:

```python
hold()  # toggle hold
hold(True)  # hold is on
hold(False)  # hold is off
```

```python
legend(handles, labels, *args, **kwargs)
```

Place a legend in the figure. Labels are a sequence of strings, handles is a sequence of `Line2D` or `Patch` instances, and `loc` can be a string or an integer specifying the legend location.

**USAGE:**

```python
legend( (line1, line2, line3),
       ('label1', 'label2', 'label3'),
       'upper right')
```

The `loc` location codes are:

- `'best'`: 0, (currently not supported for figure legends)
- `'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,

`loc` can also be an (x,y) tuple in figure coords, which specifies the lower left of the legend box.

**Keyword arguments:**

- `prop`: [None | `FontProperties | dict]` A `matplotlib.font_manager.FontProperties` instance. If `prop` is a dictionary, a new instance will be created with `prop`. If `None`, use rc settings.
- `numpoints`: integer The number of points in the legend line, default is 4
- `scatterpoints`: integer The number of points in the legend line, default is 4
scatter_offsets: list of floats a list of yoffsets for scatter symbols in legend

markerscale: [ None | scalar ] The relative size of legend markers vs. original. If None, use rc settings.

fancybox: [ None | False | True ] if True, draw a frame with a round fancybox. If None, use rc

shadow: [ None | False | True ] If True, draw a shadow behind legend. If None, use rc

ncol [integer] number of columns. default is 1

mode [[“expand” | None ]] if mode is “expand”, the legend will be horizontally expanded to fill the axes area (or bbox_to_anchor)

title [string] the legend title

Padding and spacing between various elements use following keywords parameters. The dimensions of these values are given as a fraction of the fontsize. Values from rcParams will be used if None.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>borderpad</td>
<td>the fractional whitespace inside the legend border</td>
</tr>
<tr>
<td>labelsspacing</td>
<td>the vertical space between the legend entries</td>
</tr>
<tr>
<td>handlelength</td>
<td>the length of the legend handles</td>
</tr>
<tr>
<td>handletextpad</td>
<td>the pad between the legend handle and text</td>
</tr>
<tr>
<td>borderaxespad</td>
<td>the pad between the axes and legend border</td>
</tr>
<tr>
<td>columnspacing</td>
<td>the spacing between columns</td>
</tr>
</tbody>
</table>

Note: Not all kinds of artist are supported by the legend. See LINK (FIXME) for details.

Example:
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)
```

The output formats available depend on the backend being used.

Arguments:

- **fname**: A string containing a path to a filename, or a Python file-like object, or possibly some backend-dependent object such as PdfPages.

  If `format` is `None` and `fname` is a string, the output format is deduced from the extension of the filename. If the filename has no extension, the value of the rc parameter `savefig.format` is used.

  If `fname` is not a string, remember to specify `format` to ensure that the correct backend is used.

Keyword arguments:

- **dpi**: [ `None` | scalar > 0 ] The resolution in dots per inch. If `None` it will default to the value `savefig.dpi` in the matplotlibrc file.
**facecolor, edgecolor:** the colors of the figure rectangle

**orientation:** ['landscape', 'portrait'] not supported on all backends; currently only on Postscript output

**papertype:** One of 'letter', 'legal', 'executive', 'ledger', 'a0' through 'a10', 'b0' through 'b10'. Only supported for Postscript output.

**format:** One of the file extensions supported by the active backend. Most backends support png, pdf, ps, eps and svg.

**transparent:** 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.

**frameon:** If True, the figure patch will be colored, if False, the figure background will be transparent. If not provided, the rcParam 'savefig.frameon' will be used.

**bbox_inches:** Bbox 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:** Amount of padding around the figure when bbox_inches is 'tight'.

**bbox_extra_artists:** A list of extra artists that will be considered when the tight bbox is calculated.

**sca(a)**
Set the current axes to be a and return a

**set_canvas(canvas)**
Set the canvas the contains the figure

Accepts: a FigureCanvas instance

**set_dpi(val)**
Set the dots-per-inch of the figure

Accepts: float

**set_edgecolor(color)**
Set the edge color of the Figure rectangle

Accepts: any matplotlib color - see help(colors)

**set_facecolor(color)**
Set the face color of the Figure rectangle

Accepts: any matplotlib color - see help(colors)

**set_figheight(val)**
Set the height of the figure in inches

Accepts: float

**set_figwidth(val)**
Set the width of the figure in inches

Accepts: float
set_frameon($b$)
Set whether the figure frame (background) is displayed or invisible

ACCEPTS: boolean

set_size_inches($w$, $h$, forward=False)
Set the figure size in inches

Usage:

```python
fig.set_size_inches($w$, $h$)  # OR
fig.set_size_inches(($w$, $h$))
```

optional kwarg `forward=True` will cause the canvas size to be automatically updated; eg you can resize the figure window from the shell

ACCEPTS: a w,h tuple with w,h in inches

set_tight_layout(tight)
Set whether `tight_layout()` is used upon drawing. If None, the rc-Params['figure.autolayout'] value will be set.

When providing a dict containing the keys `pad`, `w_pad`, `h_pad` and `rect`, the default `tight_layout()` paddings will be overridden.

ACCEPTS: [True | False | dict | None]

show(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 will raise an AttributeError.

For non-GUI backends, this does nothing, in which case a warning will be issued if `warn` is True (default).

subplots_adjust(*args, **kwargs)
Call signature:

```python
subplots_adjust(left=None, bottom=None, right=None, top=None, 
                wspace=None, hspace=None)
```

Update the `SubplotParams` with `kwargs` (defaulting to `rc` when `None`) and update the subplot locations

suptitle($t$, **kwargs)
Add a centered title to the figure.

`kwargs` are `matplotlib.text.Text` properties. Using figure coordinates, the defaults are:

- `x [0.5]` The x location of the text in figure coords
- `y [0.98]` The y location of the text in figure coords
- `horizontalalignment` ['center'] The horizontal alignment of the text
- `verticalalignment` ['top'] The vertical alignment of the text
A `matplotlib.text.Text` instance is returned.

Example:

```python
fig.suptitle('this is the figure title', fontsize=12)
```

```python
text(x, y, s, *args, **kwargs)
```
Add text to figure.

Call signature:

```python
text(x, y, s, fontdict=None, **kwargs)
```
Add text to figure at location $x, y$ (relative 0-1 coords). See `text()` for the meaning of the other arguments.

**kwargs control the `Text` properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an <code>Axes</code> instance</td>
</tr>
<tr>
<td>backgroundcolor</td>
<td>any <code>matplotlib</code> color</td>
</tr>
<tr>
<td>bbox</td>
<td>rectangle prop dict</td>
</tr>
<tr>
<td>clip_box</td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>{(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>any <code>matplotlib</code> color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>family or fontfamily or fontname or name</td>
<td>[FONTNAME</td>
</tr>
<tr>
<td>figure</td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td>fontproperties or font_properties</td>
<td>a <code>matplotlib.font_manager.FontProperties</code> instance</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>horizontalalignment or ha</td>
<td>[‘center’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linespacing</td>
<td>float (multiple of font size)</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>multialignment</td>
<td>[‘left’</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>position</td>
<td>(x,y)</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>rotation</td>
<td>[ angle in degrees</td>
</tr>
<tr>
<td>rotation_mode</td>
<td>unknown</td>
</tr>
<tr>
<td>size or fontsize</td>
<td>[size in points</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
</tbody>
</table>
Table 62.4 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>stretch</strong> or fontstretch</td>
<td>[a numeric value in range 0-1000</td>
</tr>
<tr>
<td><strong>style</strong> or fontstyle</td>
<td>[ ‘normal’</td>
</tr>
<tr>
<td><strong>text</strong></td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td><strong>transform</strong></td>
<td>Transform instance</td>
</tr>
<tr>
<td><strong>url</strong></td>
<td>a url string</td>
</tr>
<tr>
<td><strong>variant</strong> or fontvariant</td>
<td>[ ‘normal’</td>
</tr>
<tr>
<td><strong>verticalalignment</strong> or va or ma</td>
<td>[ ‘center’</td>
</tr>
<tr>
<td><strong>visible</strong></td>
<td>[True</td>
</tr>
<tr>
<td><strong>x</strong></td>
<td>float</td>
</tr>
<tr>
<td><strong>y</strong></td>
<td>float</td>
</tr>
<tr>
<td><strong>zorder</strong></td>
<td>any number</td>
</tr>
</tbody>
</table>

**tight_layout**(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] padding (height/width) between edges of adjacent subplots.
  Defaults to pad_inches.
- **rect** [if rect is given, it is interpreted as a rectangle] (left, bottom, right, top) in the normalized figure coordinate that the whole subplots area (including labels) will fit into. Default is (0, 0, 1, 1).

waitforbuttonpress(timeout=-1)

Call signature:

waitforbuttonpress(self, timeout=-1)

Blocking call to interact with the figure.

This will return True if a key was pressed, False if a mouse button was pressed and None if timeout was reached without either being pressed.

If timeout is negative, does not timeout.

class matplotlib.figure.SubplotParams(left=None, bottom=None, right=None, top=None, wspace=None, hspace=None)

A class to hold the parameters for a subplot

All dimensions are fraction of the figure width or height. All values default to their rc params

The following attributes are available

- **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
The top of the subplots of the figure

The amount of width reserved for blank space between subplots

The amount of height reserved for white space between subplots

Update the current values. If any kwarg is None, default to the current value, if set, otherwise to rc

**matplotlib.figure.figaspect(arg)**

Create a figure with specified aspect ratio. If arg is a number, use that aspect ratio. If arg is an array, figaspect will determine the width and height for a figure that would fit array preserving aspect ratio. The figure width, height in inches are returned. Be sure to create an axes with equal with and height,

eg

Example usage:

```
# 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)
```

Thanks to Fernando Perez for this function
63.1 `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.

Experimental support is included for using fontconfig on Unix variant platforms (Linux, OS X, Solaris). To enable it, set the constant `USE_FONTCONFIG` in this file to `True`. Fontconfig has the advantage that it is the standard way to look up fonts on X11 platforms, so if a font is installed, it is much more likely to be found.

```python
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')

On import, the `FontManager` singleton instance creates a list of TrueType fonts based on the font properties: name, style, variant, weight, stretch, and size. The `findfont()` method does a nearest neighbor search to find the font that most closely matches the specification. If no good enough match is found, a default font is returned.

```python
findfont(prop, fontext='ttf', directory=None, fallback_to_default=True, rebuild_if_missing=True)
```

Search the font list for the font that most closely matches the `FontProperties` `prop`.

`findfont()` 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 Vera Sans) is returned.

directory, is specified, will only return fonts from the given directory (or subdirectory of that directory).
The result is cached, so subsequent lookups don’t have to perform the $O(n)$ nearest neighbor search.

If `fallback_to_default` is True, will fallback to the default font family (usually “Bitstream Vera Sans” or “Helvetica”) if the first lookup hard-fails.

See the W3C Cascading Style Sheet, Level 1 documentation for a description of the font finding algorithm.

```python
static get_default_size()
    Return the default font size.

get_default_weight()
    Return the default font weight.

score_family(families, family2)
    Returns a match score between the list of font families in `families` and the font family name `family2`.
    An exact match anywhere in the list returns 0.0.
    A match by generic font name will return 0.1.
    No match will return 1.0.

score_size(size1, size2)
    Returns 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(stretch1, stretch2)
    Returns 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(style1, style2)
    Returns 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(variant1, variant2)
    Returns a match score between `variant1` and `variant2`.
    An exact match returns 0.0, otherwise 1.0.

score_weight(weight1, weight2)
    Returns a match score between `weight1` and `weight2`.
```
The result is the absolute value of the difference between the CSS numeric values of weight1 and weight2, normalized between 0.0 and 1.0.

**set_default_weight**(weight)

Set the default font weight. The initial value is ‘normal’.

**update_fonts**(filenames)

Update the font dictionary with new font files. Currently not implemented.

**class** matplotlib.font_manager.FontProperties(**family**=None, **style**=None, **variant**=None, **weight**=None, **stretch**=None, **size**=None, **fname**=None, **_init**=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 in matplotlibrc.
- **style**: Either ‘normal’, ‘italic’ or ‘oblique’.
- **variant**: Either ‘normal’ or ‘small-caps’.

The default font property for TrueType fonts (as specified in the default matplotlibrc file) is:

- sans-serif, normal, normal, normal, normal, scalable.

Alternatively, a font may be specified using an absolute path to a .ttf file, by using the **fname** kwarg.

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. See the documentation on fontconfig patterns. This support does not require fontconfig to be installed. 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**( )

Return a deep copy of self
get_family()  
Return a list of font names that comprise the font family.

get_file()  
Return the filename of the associated font.

get_fontconfig_pattern()  
Get a fontconfig pattern suitable for looking up the font as specified with fontconfig’s fc-match utility.

See the documentation on fontconfig patterns.

This support does not require fontconfig to be installed or support for it to be enabled. We are merely borrowing its pattern syntax for use here.

get_name()  
Return the name of the font that best matches the font properties.

get_size()  
Return the font size.

get_size_in_points()  

get_slant()  
Return the font style. Values are: ‘normal’, ‘italic’ or ‘oblique’.

get_stretch()  

get_style()  
Return the font style. Values are: ‘normal’, ‘italic’ or ‘oblique’.

get_variant()  
Return the font variant. Values are: ‘normal’ or ‘small-caps’.

get_weight()  

set_family(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 text.usetex is True.

set_file(file)  
Set the filename of the fontfile to use. In this case, all other properties will be ignored.

set_fontconfig_pattern(pattern)  
Set the properties by parsing a fontconfig pattern.

See the documentation on fontconfig patterns.
This support does not require fontconfig to be installed or support for it to be enabled. We are merely borrowing its pattern syntax for use here.

**set_name** *(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 text.usetex is True.

**set_size** *(size)*

**set_slant** *(style)*
Set the font style. Values are: ‘normal’, ‘italic’ or ‘oblique’.

**set_stretch** *(stretch)*

**set_style** *(style)*
Set the font style. Values are: ‘normal’, ‘italic’ or ‘oblique’.

**set_variant** *(variant)*
Set the font variant. Values are: ‘normal’ or ‘small-caps’.

**set_weight** *(weight)*

```python
matplotlib.font_manager.OSXInstalledFonts(directories=None, fontext='ttf')
```
Get list of font files on OS X - ignores font suffix by default.

```python
matplotlib.font_manager.fmfFontProperty(fontpath, font)
```
A function for populating a FontKey instance by extracting information from the AFM font file. *font* is a class:AFM instance.

```python
matplotlib.font_manager.createFontList(fontfiles, fontext='ttf')
```
A function to create a font lookup list. The default is to create a list of TrueType fonts. An AFM font list can optionally be created.

```python
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.

```python
matplotlib.font_manager.findfont(prop, **kw)
```

```python
matplotlib.font_manager.get_fontconfig_fonts(fontext='ttf')
```
Grab a list of all the fonts that are being tracked by fontconfig by making a system call to fc-list. This is an easy way to grab all of the fonts the user wants to be made available to applications, without needing knowing where all of them reside.
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)
    Returns True if 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.list_fonts(directory, extensions)
    Return a list of all fonts matching any of the extensions, possibly upper-cased, found recursively under the directory.

matplotlib.font_manager.pickle_dump(data, filename)
    Equivalent to pickle.dump(data, open(filename, ‘w’)) but closes the file to prevent filehandle leakage.

matplotlib.font_manager.pickle_load(filename)
    Equivalent to pickle.load(open(filename, ‘r’)) but closes the file to prevent filehandle leakage.

matplotlib.font_manager.ttfFontProperty(font)
    A function for populating the FontKey by extracting information from the TrueType font file.

    font is a FT2Font instance.

matplotlib.font_manager.tffdict_to_fnames(d)
    flatten a tffdict to all the filenames it contains

matplotlib.font_manager.weight_as_number(weight)
    Return the weight property as a numeric value. String values are converted to their corresponding numeric value.

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. A list of TrueType font filenames are returned by default, or AFM fonts if fontext == ‘afm’.

63.2 matplotlib.fontconfig_pattern

A module for parsing and generating fontconfig patterns.

See the fontconfig pattern specification for more information.

class matplotlib.fontconfig_pattern.FontconfigPatternParser
    A simple pyparsing-based parser for fontconfig-style patterns.

    See the fontconfig pattern specification for more information.
parse(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()
sub(repl, string[, count = 0]) -> newstring Return the string obtained by replacing the leftmost non-overlapping occurrences of pattern in string by the replacement repl.

matplotlib.fontconfig_pattern.family_unescape()
sub(repl, string[, count = 0]) -> newstring 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.value_escape()
sub(repl, string[, count = 0]) -> newstring Return the string obtained by replacing the leftmost non-overlapping occurrences of pattern in string by the replacement repl.

matplotlib.fontconfig_pattern.value_unescape()
sub(repl, string[, count = 0]) -> newstring Return the string obtained by replacing the leftmost non-overlapping occurrences of pattern in string by the replacement repl.
64.1 matplotlib.gridspec

gridspec is a module which specifies the location of the subplot in the figure.

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.

```python
class matplotlib.gridspec.GridSpec(nrows, ncols, left=None, bottom=None, right=None, top=None, wspace=None, hspace=None, width_ratios=None, height_ratios=None)
```

Bases: matplotlib.gridspec.GridSpecBase

A class that specifies the geometry of the grid that a subplot will be placed. The location of grid is determined by similar way as the SubplotParams.

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.

get_subplot_params(fig=None)

return a dictionary of subplot layout parameters. The default parameters are from rcParams unless a figure attribute is set.

locally_modified_subplot_params()

tight_layout(fig, 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] padding (height/width) between edges of adjacent subplots. Defaults to pad_inches.
- **rect**: [if rect is given, it is interpreted as a rectangle] (left, bottom, right, top) in the normalized figure coordinate that the whole subplots area (including labels) will fit into. Default is (0, 0, 1, 1).
update(**kwargs)
    Update the current values. If any kwarg is None, default to the current value, if set, otherwise
to rc.

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.
    The number of rows and number of columns of the grid need to be set. Optionally, the ratio of heights
    and widths of rows and columns can be specified.

    get_geometry()
        get the geometry of the grid, eg 2,3

    get_grid_positions(fig)
        return lists of bottom and top position of rows, left and right positions of columns.

    get_height_ratios()

    get_subplot_params(fig=None)

    get_width_ratios()

    new_subplotspec(loc, rowspan=1, colspan=1)
        create and return a SubplotSpec instance.

    set_height_ratios(height_ratios)

    set_width RATios(width_ratios)

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 Sub-
    plotSpec.
    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.

    get_subplot_params(fig=None)
        return a dictionary of subplot layout parameters.

    get_topmost_subplotspec()
        get the topmost SubplotSpec instance associated with the subplot
class matplotlib.gridspec.SubplotSpec(gridspec, num1, num2=None)
    Bases: object

    specifies the location of the subplot in the given GridSpec.
    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.
    The index starts from 0.

    get_geometry()
        get the subplot geometry, eg 2,2,3. Unlike SuplorParams, index is 0-based

    get_gridspec()

    get_position(fig, return_all=False)
        update the subplot position from fig.subplotpars

    get_topmost_subplotspec()
        get the topmost SubplotSpec instance associated with the subplot
65.1 `matplotlib.legend`

The legend module defines the `Legend` class, which is responsible for drawing legends associated with axes and/or figures.

The `Legend` class can be considered as a container of legend handles and legend texts. Creation of corresponding legend handles from the plot elements in the axes or figures (e.g., lines, patches, etc.) are specified by the handler map, which defines the mapping between the plot elements and the legend handlers to be used (the default legend handlers are defined in the `legend_handler` module). Note that not all kinds of artist are supported by the legend yet (See `Legend guide` for more information).

```python
class matplotlib.legend.DraggableLegend(legend, use_blit=False, update='loc')
    Bases: matplotlib.offsetbox.DraggableOffsetBox
    update [If “loc”, update loc parameter of] legend upon finalizing. If “bbox”, update bbox_to_anchor parameter.
    artist_picker(legend, evt)

def finalize_offset()
```

```python
class matplotlib.legend.Legend(parent, handles, labels, loc=None, numpoints=None, markerscale=None, scatterpoints=None, scatteryoffsets=None, prop=None, fontsize=None, borderpad=None, labelsspacing=None, handlelength=None, handleheight=None, handletextpad=None, borderaxespad=None, columnspacing=None, ncol=1, mode=None, fancybox=None, shadow=None, title=None, framealpha=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. Labels are a sequence of strings and loc can be a string or an integer specifying the legend location.

The location codes are:

- `'best'`: 0, (only implemented for axis legends)
- `'upper right'`: 1,
The pad and spacing parameters are measured in font-size units. e.g., a fontsize of 10 points and a handlelength=5 implies a handlelength of 50 points. Values from rcParams will be used if None.

Users can specify any arbitrary location for the legend using the bbox_to_anchor keyword argument. bbox_to_anchor can be an instance of BboxBase(or its derivatives) 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 = {'right': 5, 'center left': 6, 'upper right': 1, 'lower right': 4, 'best': 0, 'center': 10, 'lower left': 3, 'center right': 7, 'upper left': 2, 'upper center': 9, 'lower center': 8}

contains(event)

draggable(state=None, use_blit=False, update='loc')
  Set the draggable state – if state is
  • None : toggle the current state
  • True : turn draggable on
  • False : turn draggable off
  If draggable is on, you can drag the legend on the canvas with the mouse. The DraggableLegend helper instance is returned if draggable is on.
  The update parameter control which parameter of the legend changes when dragged. If update is “loc”, the loc parameter of the legend is changed. If “bbox”, the bbox_to_anchor parameter is changed.

draw(artist, renderer, *args, **kwargs)
  Draw everything that belongs to the legend

draw_frame(b)
  b is a boolean. Set draw frame to b

get_bbox_to_anchor()
  return the bbox that the legend will be anchored

get_children()
  return a list of child artists

classmethod get_default_handler_map()
  A class method that returns the default handler map.

get_frame()
  return the Rectangle instance used to frame the legend

get_frame_on()
  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()
  return the handler map.

get_lines()
  return a list of lines.Line2D instances in the legend
get_patches()
    return a list of patch instances in the legend

get_texts()
    return a list of text.Text instance in the legend

get_title()
    return Text instance for the legend title

get_window_extent(*args, **kwargs)
    return a extent of the the legend

def set_bbox_to_anchor(bbox, transform=None)
    set the bbox that the legend will be anchored.
    bbox can be a BboxBase instance, a tuple of [left, bottom, width, height] in the given transform (normalized axes coordinate if None), or a tuple of [left, bottom] where the width and height will be assumed to be zero.

classmethod set_default_handler_map(handler_map)
    A class method to set the default handler map.

def set_frame_on(b)
    Set whether the legend box patch is drawn
    ACCEPTS: [ True | False ]

def set_title(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
66.1 matplotlib.markers

This module contains functions to handle markers. Used by both the marker functionality of `plot` and `scatter`.

All possible markers are defined here:

<table>
<thead>
<tr>
<th>marker</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;.&quot;</td>
<td>point</td>
</tr>
<tr>
<td>&quot;,&quot;</td>
<td>pixel</td>
</tr>
<tr>
<td>&quot;o&quot;</td>
<td>circle</td>
</tr>
<tr>
<td>&quot;v&quot;</td>
<td>triangle_down</td>
</tr>
<tr>
<td>&quot;^&quot;</td>
<td>triangle_up</td>
</tr>
<tr>
<td>&quot;&lt;&quot;</td>
<td>triangle_left</td>
</tr>
<tr>
<td>&quot;&gt;&quot;</td>
<td>triangle_right</td>
</tr>
<tr>
<td>&quot;1&quot;</td>
<td>tri_down</td>
</tr>
<tr>
<td>&quot;2&quot;</td>
<td>tri_up</td>
</tr>
<tr>
<td>&quot;3&quot;</td>
<td>tri_left</td>
</tr>
<tr>
<td>&quot;4&quot;</td>
<td>tri_right</td>
</tr>
<tr>
<td>&quot;8&quot;</td>
<td>octagon</td>
</tr>
<tr>
<td>&quot;s&quot;</td>
<td>square</td>
</tr>
<tr>
<td>&quot;p&quot;</td>
<td>pentagon</td>
</tr>
<tr>
<td>&quot;*&quot;</td>
<td>star</td>
</tr>
<tr>
<td>&quot;h&quot;</td>
<td>hexagon1</td>
</tr>
<tr>
<td>&quot;H&quot;</td>
<td>hexagon2</td>
</tr>
<tr>
<td>&quot;+&quot;</td>
<td>plus</td>
</tr>
<tr>
<td>&quot;x&quot;</td>
<td>x</td>
</tr>
<tr>
<td>&quot;D&quot;</td>
<td>diamond</td>
</tr>
<tr>
<td>&quot;d&quot;</td>
<td>thin_diamond</td>
</tr>
<tr>
<td>&quot;</td>
<td></td>
</tr>
<tr>
<td>&quot;_&quot;</td>
<td>hline</td>
</tr>
<tr>
<td>TICKLEFT</td>
<td>tickleft</td>
</tr>
<tr>
<td>TICKRIGHT</td>
<td>tickright</td>
</tr>
<tr>
<td>TICKUP</td>
<td>tickup</td>
</tr>
</tbody>
</table>
Table 66.1 – continued from previous page

<table>
<thead>
<tr>
<th>marker</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TICKDOWN</td>
<td>tickdown</td>
</tr>
<tr>
<td>CARETLEFT</td>
<td>caretleft</td>
</tr>
<tr>
<td>CARETRIGHT</td>
<td>caretright</td>
</tr>
<tr>
<td>CARETUP</td>
<td>caretup</td>
</tr>
<tr>
<td>CARETDOWN</td>
<td>caretdown</td>
</tr>
<tr>
<td>“None”</td>
<td>nothing</td>
</tr>
<tr>
<td>None</td>
<td>nothing</td>
</tr>
<tr>
<td>‘’</td>
<td>nothing</td>
</tr>
<tr>
<td>‘’</td>
<td>nothing</td>
</tr>
<tr>
<td>‘$...$’</td>
<td>render the string using mathtext.</td>
</tr>
<tr>
<td>verts</td>
<td>a list of (x, y) pairs used for Path vertices.</td>
</tr>
<tr>
<td>path</td>
<td>a Path instance.</td>
</tr>
<tr>
<td>(numsides, style, angle)</td>
<td>see below</td>
</tr>
</tbody>
</table>

The marker can also be a tuple (numsides, style, angle), which will create a custom, regular symbol.

- **numsides**: the number of sides
- **style**: the style of the regular symbol:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a regular polygon</td>
</tr>
<tr>
<td>1</td>
<td>a star-like symbol</td>
</tr>
<tr>
<td>2</td>
<td>an asterisk</td>
</tr>
<tr>
<td>3</td>
<td>a circle (numsides and angle is ignored)</td>
</tr>
</tbody>
</table>

- **angle**: the angle of rotation of the symbol, in degrees

For backward compatibility, the form (verts, 0) is also accepted, but it is equivalent to just verts for giving a raw set of vertices that define the shape.

**class** matplotlib.markers.MarkerStyle(marker=None, fillstyle='full')

**Bases**: object

**MarkerStyle**

**Parameters**

- **marker**: string or array_like, optional, default: None
  - See the descriptions of possible markers in the module docstring.
- **fillstyle**: string, optional, default: ‘full’

**Attributes**

- **markers**
- **fillstyles**
  - tuple() -> empty tuple
- **filled_markers**
  - tuple() -> empty tuple
filled_markers = ('o', 'v', '^', '<', '>', '8', 's', 'p', '*', 'h', 'H', 'D', 'd')

fillstyles = ('full', 'left', 'right', 'bottom', 'top', 'none')

global.get_alt_path()
global.get_alt_transform()
global.get_capstyle()
global.get_fillstyle()
global.get_joinstyle()
global.get_marker()
global.get_path()
global.get_snap_threshold()
global.get_transform()
global.is_filled()


set_fillstyle(fillstyle)
    Sets fillstyle
    Parameters fillstyle : string amongst known fillstyles

set_marker(marker)
Matplotlib, Release 1.3.0

CHAPTER SIXTYSEVEN

MATHTEXT

67.1 matplotlib.mathtext

mathtext is a module for parsing a subset of the TeX math syntax and drawing them 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.

If you find TeX expressions that don’t parse or render properly, please email mdroe@stsci.edu, but please check KNOWN ISSUES below first.

class matplotlib.mathtext.Accent(c, state)
    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()

    render(x, y)
        Render the character to the canvas.

    shrink()

class matplotlib.mathtext.AutoHeightChar(c, height, depth, state, always=False, factor=None)
    Bases: matplotlib.mathtext.Hlist

    AutoHeightChar will create 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

    AutoWidthChar will create 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.

    alias = "\\“}
get_sized_alternatives_for_symbol(fontname, sym)

target = ‘]’

class matplotlib.mathtext.Box(width, height, depth)
    Bases: matplotlib.mathtext.Node
    Represents any node with a physical location.
    grow()

    render(x1, y1, x2, y2)

    shrink()

class matplotlib.mathtext.Char(c, state)
    Bases: matplotlib.mathtext.Node
    Represents 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 way, 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(next)
        Return the amount of kerning between this and the given character. Called when characters are
        strung together into Hlist to create Kern nodes.

    grow()

    is_slanted()

    render(x, y)
        Render the character to the canvas

    shrink()

matplotlib.mathtext.Error(msg)
    Helper class to raise parser errors.

class matplotlib.mathtext.Fill
    Bases: matplotlib.mathtext.Glue

class matplotlib.mathtext.Filll
    Bases: matplotlib.mathtext.Glue

class matplotlib.mathtext.Fillll
    Bases: matplotlib.mathtext.Glue
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.

default_font_prop: A FontProperties object to use for the default non-math font, or the base font for Unicode (generic) font rendering.

mathtext_backend: A subclass of MathTextBackend used to delegate the actual rendering.

def destroy()
    Fix any cyclical references before the object is about to be destroyed.

def get_kern(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(font, font_class, sym, fontsize, dpi)

    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

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(box)

Get the data needed by the backend to render the math expression. The return value is backend-specific.
**get_sized_alternatives_for_symbol** *(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** *(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** *

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** *(font, fontsize, dpi)*

Get the xheight for the given `font` and `fontsize`.

**render_glyph** *(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** *(x1, y1, x2, y2)*

Draw a filled rectangle from `(x1, y1)` to `(x2, y2)`.

**set_canvas_size** *(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=False)**

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.)

**grow** *

**shrink** *

**class matplotlib.mathtext.GlueSpec(width=0.0, stretch=0.0, stretch_order=0, shrink=0.0, shrink_order=0)**

Bases: `object`

See `Glue`.

**copy** *
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(w=0.0, m='additional')
        The main duty of hpack() is to compute the dimensions of the resulting boxes, and to 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.
        • w: specifies a width
        • m: is either ‘exactly’ or ‘additional’.
        Thus, hpack(w, ‘exactly’) produces a box whose width is exactly w, while hpack(w, ‘additional’) yields a box whose width is the natural width plus w. The default values produce a box with the natural width.

    kern()
        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 linked
        list in the correct way.

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()

    height = 0
shrink()

class matplotlib.mathtext.List(elements)
   Bases: matplotlib.mathtext.Box

   A list of nodes (either horizontal or vertical).

grow()

shrink()

class matplotlib.mathtext.MathTextParser(output)
   Bases: object

   Create a MathTextParser for the given backend output.

get_depth(texstr, dpi=120, fontsize=14)
   Returns the offset of the baseline from the bottom of the image in pixels.
   texstr A valid mathtext string, eg r’IQ: $\sigma_i=15$’
   dpi The dots-per-inch to render the text
   fontsize The font size in points

parse(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(texstr, dpi=120, fontsize=14)

   texstr A valid mathtext string, eg r’IQ: $\sigma_i=15$’
   dpi The dots-per-inch to render the text
   fontsize The font size in points

   Returns a tuple (array, depth)
      • array is an NxM uint8 alpha ubyte mask array of rasterized tex.
      • depth is the offset of the baseline from the bottom of the image in pixels.

to_png(filename, texstr, color='black', dpi=120, fontsize=14)

   Writes a tex expression to a PNG file.

   Returns the offset of the baseline from the bottom of the image in pixels.
   filename A writable filename or fileobject
   texstr A valid mathtext string, eg r’IQ: $\sigma_i=15$’
   color A valid matplotlib color argument
   dpi The dots-per-inch to render the text
   fontsize The font size in points

   Returns the offset of the baseline from the bottom of the image in pixels.

to_rgba(texstr, color='black', dpi=120, fontsize=14)

   texstr A valid mathtext string, eg r’IQ: $\sigma_i=15$’
**color** Any matplotlib color argument

**dpi** The dots-per-inch to render the text

**fontsize** The font size in points

Returns a tuple \((array, depth)\)
- \(array\) is an N\times M \text{uint8} alpha ubyte mask array of rasterized tex.
- \(depth\) is the offset of the baseline from the bottom of the image in pixels.

---

**exception** `matplotlib.mathtext.MathTextWarning`

Bases: `exceptions.Warning`

**class** `matplotlib.mathtext.MathTextBackend`

Bases: `object`

The base class for the mathtext backend-specific code. The purpose of `MathTextBackend` subclasses is to interface between mathtext and a specific matplotlib graphics backend.

Subclasses need to override the following:
- `render_glyph()`
- `render_filled_rect()`
- `get_results()`

And optionally, if you need to use a Freetype hinting style:
- `get_hinting_type()`

**get_hinting_type()**

Get the Freetype hinting type to use with this particular backend.

**get_results**(*box*)

Return a backend-specific tuple to return to the backend after all processing is done.

**render_filled_rect**(*x1, y1, x2, y2*)

Draw a filled black rectangle from \((x1, y1)\) to \((x2, y2)\).

**render_glyph**(*ox, oy, info*)

Draw a glyph described by \(info\) to the reference point \((ox, oy)\).

**set_canvas_size**(*w, h, d*)

Dimension 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()**

**get_results**(*box, used_characters*)

**render_glyph**(*ox, oy, info*)

**render_rect_filled**(*x1, y1, x2, y2*)
set_canvas_size(w, h, d)

class matplotlib.mathtext.MathtextBackendBitmap
    Bases: matplotlib.mathtext.MathtextBackendAgg
    get_results(box, used_characters)

class matplotlib.mathtext.MathtextBackendCairo
    Bases: matplotlib.mathtext.MathtextBackend
    Store information to write a mathtext rendering to the Cairo backend.
    get_results(box, used_characters)

    render_glyph(ox, oy, info)

    render_rect_filled(x1, y1, x2, y2)

class matplotlib.mathtext.MathtextBackendPath
    Bases: matplotlib.mathtext.MathtextBackend
    Store information to write a mathtext rendering to the text path machinery.
    get_results(box, used_characters)

    render_glyph(ox, oy, info)

    render_rect_filled(x1, y1, x2, y2)

class matplotlib.mathtext.MathtextBackendPdf
    Bases: matplotlib.mathtext.MathtextBackend
    Store information to write a mathtext rendering to the PDF backend.
    get_results(box, used_characters)

    render_glyph(ox, oy, info)

    render_rect_filled(x1, y1, x2, y2)

class matplotlib.mathtext.MathtextBackendPs
    Bases: matplotlib.mathtext.MathtextBackend
    Store information to write a mathtext rendering to the PostScript backend.
    get_results(box, used_characters)
render_glyph(ox, oy, info)

render_rect_filled(x1, y1, x2, y2)

class matplotlib.mathtext.MathtextBackendSvg
    Bases: matplotlib.mathtext.MathtextBackend
    Store information to write a mathtext rendering to the SVG backend.
    get_results(box, used_characters)

    render_glyph(ox, oy, info)

    render_rect_filled(x1, y1, x2, y2)

class matplotlib.mathtext.NegFil
    Bases: matplotlib.mathtext.Glue

class matplotlib.mathtext.NegFill
    Bases: matplotlib.mathtext.Glue

class matplotlib.mathtext.NegFilll
    Bases: matplotlib.mathtext.Glue

class matplotlib.mathtext.Node
    Bases: object
    A node in the TeX box model
    get_kerning(next)

grow()
    Grows one level larger. There is no limit to how big something can get.

    render(x, y)

shrink()
    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
    This is the pyparsing-based parser for math expressions. It actually parses full strings containing math expressions, in that 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.

\texttt{copy()}

\texttt{font}

\texttt{Parser.accelt(s, loc, toks)}

\texttt{Parser.auto_delim(s, loc, toks)}

\texttt{Parser.binom(s, loc, toks)}

\texttt{Parser.c_over_c(s, loc, toks)}

\texttt{Parser.customspace(s, loc, toks)}

\texttt{Parser.end_group(s, loc, toks)}

\texttt{Parser.font(s, loc, toks)}

\texttt{Parser.frac(s, loc, toks)}

\texttt{Parser.function(s, loc, toks)}

\texttt{Parser.genfrac(s, loc, toks)}

\texttt{Parser.get_state()}

Get the current \texttt{State} of the parser.

\texttt{Parser.group(s, loc, toks)}

\texttt{Parser.is_dropsb(nucleus)}

\texttt{Parser.is_overunder(nucleus)}

\texttt{Parser.is_slanted(nucleus)}
Parser.main(s, loc, toks)

Parser.math(s, loc, toks)

Parser.math_string(s, loc, toks)

Parser.non_math(s, loc, toks)

Parser.operatorname(s, loc, toks)

Parser.overline(s, loc, toks)

Parser.parse(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.

Parser.pop_state()
    Pop a State off of the stack.

Parser.push_state()
    Push a new State onto the stack which is just a copy of the current state.

Parser.required_group(s, loc, toks)

Parser.simple_group(s, loc, toks)

Parser.space(s, loc, toks)

Parser.sqrt(s, loc, toks)

Parser.stackrel(s, loc, toks)

Parser.start_group(s, loc, toks)

Parser.subsuper(s, loc, toks)

Parser.symbol(s, loc, toks)

Parser.unknown_symbol(s, loc, toks)
class matplotlib.mathtext.Rule(width, height, depth, state)
Bases: matplotlib.mathtext.Box

A Rule node stands for 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(x, y, w, h)

class matplotlib.mathtext.Ship
Bases: object

Once the boxes 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(box)

vlist_out(box)

class matplotlib.mathtext.SsGlue
Bases: matplotlib.mathtext.Glue

class matplotlib.mathtext.StandardPsFonts(*args, **kwargs)
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.

basepath = '/home/mdboom/python/lib/python2.7/site-packages/matplotlib-1.3.0-py2.7-linux-x86_64.egg/matplotlib/mpl-data/fonts/afm'

fontmap = {'bf': 'pncb8a', 'tt': 'pccr8a', 'it': 'pncr8a', 'rm': 'pncr8a', 'sf': 'pncr8a', 'cal': 'pzcmi8a', 'rm': 'pncr8a', 'sf': 'pncr8a'}

get_kern(font1, fontclass1, sym1, fontsize1, font2, fontclass2, sym2, fontsize2, dpi)

get_underline_thickness(font, fontsize, dpi)

get_xheight(font, fontsize, dpi)

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.

```python
cm_fallback = False
```

```python
get_sized_alternatives_for_symbol(fontname, sym)
```

```python
use_cmex = False
```

```python
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).
```

```python
class matplotlib.mathtext.SubSuperCluster
    Bases: matplotlib.mathtext.Hlist
    SubSuperCluster is a sort of hack to get around that fact that this code do 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.
```

```python
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).
```

```python
class CachedFont(font)

    TruetypeFonts.destroy()
```

```python
    TruetypeFonts.get_kern(font1, fontclass1, sym1, fontsize1, font2, fontclass2, sym2, fontsize2, dpi)
```

```python
    TruetypeFonts.get_underline_thickness(font, fontsize, dpi)
```

```python
    TruetypeFonts.get_xheight(font, fontsize, dpi)
```

```python
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.
```
get_sized_alternatives_for_symbol(\texttt{fontname, sym})

\texttt{use_cmex = True}

class \texttt{matplotlib.\texttt{mathtext.\texttt{VCentered}(elements)}}
\hspace{1em} Bases: \texttt{matplotlib.\texttt{mathtext.\texttt{Hlist}}}

A convenience class to create a \texttt{Vlist} whose contents are centered within its enclosing box.

class \texttt{matplotlib.\texttt{mathtext.\texttt{Vbox}(height, depth)}}
\hspace{1em} Bases: \texttt{matplotlib.\texttt{mathtext.\texttt{Box}}}

A box with only height (zero width).

class \texttt{matplotlib.\texttt{mathtext.\texttt{Vlist}(elements, h=0.0, m='additional')}}
\hspace{1em} Bases: \texttt{matplotlib.\texttt{mathtext.\texttt{List}}}

A vertical list of boxes.

\texttt{vpack(h=0.0, m='additional', l=inf)}

The main duty of \texttt{vpack()} is to compute the dimensions of the resulting boxes, and to adjust the glue if one of those dimensions is pre-specified.

• \texttt{h}: specifies a height
• \texttt{m}: is either ‘exactly’ or ‘additional’.
• \texttt{l}: a maximum height

Thus, \texttt{vpack(h, 'exactly')} produces a box whose height is exactly \texttt{h}, while \texttt{vpack(h, 'additional')} yields a box whose height is the natural height plus \texttt{h}. The default values produce a box with the natural width.

class \texttt{matplotlib.\texttt{mathtext.\texttt{Vrule(state)}}}
\hspace{1em} Bases: \texttt{matplotlib.\texttt{mathtext.\texttt{Rule}}}

Convenience class to create a vertical rule.

\texttt{matplotlib.\texttt{mathtext.\texttt{get_unicode_index(symbol)} \to integer}}

Return the integer index (from the Unicode table) of \texttt{symbol}. \texttt{symbol} can be a single unicode character, a \TeX{} command (i.e. \texttt{r'pi'}), or a Type1 symbol name (i.e. \texttt{‘phi’}).

\texttt{matplotlib.\texttt{mathtext.\texttt{math_to_image}(s, filename_or_obj, prop=\texttt{None}, dpi=\texttt{None}, format=\texttt{None})}}

Given a math expression, renders it in a closely-clipped bounding box to an image file. \texttt{s} A math expression. The math portion should be enclosed in dollar signs.

\texttt{filename_or_obj} A filepath or writable file-like object to write the image data to.

\texttt{prop} If provided, a FontProperties() object describing the size and style of the text.

\texttt{dpi} Override the output dpi, otherwise use the default associated with the output format.

\texttt{format} The output format, e.g., ‘svg’, ‘pdf’, ‘ps’ or ‘png’. If not provided, will be deduced from the filename.

\texttt{matplotlib.\texttt{mathtext.\texttt{unichr_safe(index)}}}

Return the Unicode character corresponding to the index, or the replacement character if this is a narrow build of Python and the requested character is outside the BMP.
68.1 matplotlib.mlab

Numerical python functions written for compatibility with MATLAB commands with the same names.

68.1.1 MATLAB compatible 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

**find()**  

Return the indices where some condition is true; numpy.nonzero is similar but more general.

**griddata()**  

interpolate irregularly distributed data to a regular grid.

**prctile()**  find the percentiles of a sequence

**prepca()**  Principal Component Analysis

**psd()**  Power spectral density using Welch’s average periodogram

**rk4()**  A 4th order Runge-Kutta integrator for 1D or ND systems

**specgram()**  Spectrogram (power spectral density over segments of time)

68.1.2 Miscellaneous functions

Functions that don’t exist in MATLAB, but are useful anyway:

**cohere_pairs()**  Coherence over all pairs. This is not a MATLAB function, but we compute coherence a lot in my lab, and we compute it for a lot of pairs. This function is optimized to do this efficiently by caching the direct FFTs.

**rk4()**  A 4th order Runge-Kutta ODE integrator in case you ever find yourself stranded without scipy (and the far superior scipy.integrate tools)
contiguous_regions() return the indices of the regions spanned by some logical mask

cross_from_below() return the indices where a 1D array crosses a threshold from below
cross_from_above() return the indices where a 1D array crosses a threshold from above

68.1.3 record array helper functions

A collection of helper methods for numpy record arrays

See misc-examples-index

rec2txt() pretty print a record array
rec2csv() store record array in CSV file
csv2rec() import record array from CSV file with type inspection
rec_append_fields() adds field(s)/array(s) to record array
rec_drop_fields() drop fields from record array
rec_join() join two record arrays on sequence of fields
recs_join() a simple join of multiple recarrays using a single column as a key
rec_groupby() summarize data by groups (similar to SQL GROUP BY)
rec_summarize() helper code to filter rec array fields into new fields

For the rec viewer functions (e.g., rec2csv), there are a bunch of Format objects you can pass into the functions that will do things like color negative values red, set percent formatting and scaling, etc.

Example usage:

```python
r = csv2rec('somefile.csv', checkrows=0)
formatd = dict(
    weight = FormatFloat(2),
    change = FormatPercent(2),
    cost = FormatThousands(2),
)
rec2excel(r, 'test.xls', formatd=formatd)
rec2csv(r, 'test.csv', formatd=formatd)
scroll = rec2gtk(r, formatd=formatd)

win = gtk.Window()
win.set_size_request(600,800)
win.add(scroll)
win.show_all()
gtk.main()
```
68.1.4 Deprecated functions

The following are deprecated; please import directly from numpy (with care–function signatures may differ):

**load()** load ASCII file - use numpy.loadtxt

**save()** save ASCII file - use numpy.savetxt

```python
class matplotlib.mlab.FIFOBuffer(*args, **kwargs)
```
A FIFO queue to hold incoming x, y data in a rotating buffer using numpy arrays under the hood. It is assumed that you will call asarrays much less frequently than you add data to the queue – otherwise another data structure will be faster.

This can be used to support plots where data is added from a real time feed and the plot object wants to grab data from the buffer and plot it to screen less frequently than the incoming.

If you set the `dataLim` attr to BBox (eg `matplotlib.Axes.dataLim`), the `dataLim` will be updated as new data come in.

TODO: add a grow method that will extend nmax

**Note:** mlab seems like the wrong place for this class.

Deprecated since version 1.3: The FIFOBuffer class was deprecated in version 1.3.

Buffer up to `nmax` points.

```python
add(x, y)
```
Add scalar x and y to the queue.

```python
asarrays()
```
Return x and y as arrays; their length will be the len of data added or `nmax`.

```python
last()
```
Get the last x, y or `None`. `None` if no data set.

```python
register(func, N)
```
Call `func` every time `N` events are passed; `func` signature is `func(fifo)`.

```python
update_datalim_to_current()
```
Update the `datalim` in the current data in the fifo.

```python
class matplotlib.mlab.FormatBool
```
Bases: `matplotlib.mlab.FormatObj`

```python
fromstr(s)
```

```python
toval(x)
```

```python
class matplotlib.mlab.FormatDate(fmt)
```
Bases: `matplotlib.mlab.FormatObj`

```python
fromstr(s)
```
toval(x)

class matplotlib.mlab.FormatDatetime(fmt='%Y-%m-%d %H:%M:%S')
    Bases: matplotlib.mlab.FormatDate
    fromstr(x)

class matplotlib.mlab.FormatFloat(precision=4, scale=1.0)
    Bases: matplotlib.mlab.FormatFormatStr
    fromstr(s)
    toval(x)

class matplotlib.mlab.FormatFormatStr(fmt)
    Bases: matplotlib.mlab.FormatObj
    tostr(x)

class matplotlib.mlab.FormatInt
    Bases: matplotlib.mlab.FormatObj
    fromstr(s)
    tostr(x)
    toval(x)

class matplotlib.mlab.FormatMillions(precision=4)
    Bases: matplotlib.mlab.FormatFloat
    class matplotlib.mlab.FormatObj
        fromstr(s)
        tostr(x)
        toval(x)

class matplotlib.mlab.FormatPercent(precision=4)
    Bases: matplotlib.mlab.FormatFloat

class matplotlib.mlab.FormatString
    Bases: matplotlib.mlab.FormatObj
tostr(x)

class matplotlib.mlab.FormatThousands (precision=4)
    Bases: matplotlib.mlab.FormatFloat

class matplotlib.mlab.PCA(a)
    compute the SVD of a and store data for PCA. Use project to project the data onto a reduced set of dimensions
    Inputs:
        a: a numobservations x numdims array
    Attrs:
        a a centered unit sigma version of input a
        numrows, numcols: the dimensions of a
        mu : a numdims array of means of a
        sigma : a numdims array of standard deviation of a
        fracs : the proportion of variance of each of the principal components
        Yt : the weight vector for projecting a numdims point or array into PCA space
        Y : a projected into PCA space

    The factor loadings are in the Wt factor, ie the factor loadings for the 1st principal component are given by Wt[0]

center(x)
    center the data using the mean and sigma from training set a

project(x, minfrac=0.0)
    project x onto the principle axes, dropping any axes where fraction of variance<minfrac

matplotlib.mlab.amap(function, sequence[, sequence, ...]) → array.
    Works like map(), but it returns an array. This is just a convenient shorthand for numpy.array(map(...)).

matplotlib.mlab.base_repr(number, base=2, padding=0)
    Return the representation of a number in any given base.

matplotlib.mlab.binary_repr(number, max_length=1025)
    Return the binary representation of the input number as a string.

    This is more efficient than using base_repr() with base 2.

    Increase the value of max_length for very large numbers. Note that on 32-bit machines, 2**1023 is the largest integer power of 2 which can be converted to a Python float.

matplotlib.mlab.bivariate_normal(X, Y, sigmax=1.0, sigmay=1.0, mux=0.0, muy=0.0, sigmaxy=0.0)
    Bivariate Gaussian distribution for equal shape X, Y.

    See bivariate normal at mathworld.
matplotlib.mlab.center_matrix(M, dim=0)

Return the matrix M with each row having zero mean and unit std.

If dim = 1 operate on columns instead of rows. (dim is opposite to the numpy axis kwarg.)

matplotlib.mlab.cohere(x, y, NFFT=256, Fs=2, detrend=<function detrend_none at 0x24627d0>, window=<function window_hanning at 0x2462578>, noverlap=0, pad_to=None, sides='default', scale_by_freq=None)

The coherence between x and y. Coherence is the normalized cross spectral density:

\[ C_{xy} = \frac{|P_{xy}|^2}{P_{xx}P_{yy}} \]  

\[(68.1)\]

x, y Array or sequence containing the data

Keyword arguments:

**NFFT**: integer The number of data points used in each block for the FFT. Must be even; a power 2 is most efficient. The default value is 256. This should NOT be used to get zero padding, or the scaling of the result will be incorrect. Use pad_to for this instead.

**Fs**: scalar The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, freqs, in cycles per time unit. The default value is 2.

**detrend**: callable 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 pylab module defines detrend_none(), detrend_mean(), and detrend_linear(), but you can use a custom function as well.

**window**: callable or ndarray 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. The default is window_hanning(). 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.

**pad_to**: integer 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 psd (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.

**sides**: [ ‘default’ | ‘onesided’ | ‘twosided’ ] Specifies which sides of the PSD to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. ‘onesided’ forces the return of a one-sided PSD, while ‘twosided’ forces two-sided.

**scale_by_freq**: boolean Specifies 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**: integer The number of points of overlap between blocks. The default value is 0 (no overlap).

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() and csd() For information about the methods used to compute $P_{xy}$, $P_{xx}$ and $P_{yy}$.

\begin{verbatim}
matplotlib.mlab.cohere_pairs(X, ij, NFFT=256, Fs=2, detrend=<function detrend_none at 0x24627d0>, window=<function window_hanning at 0x2462578>, noverlap=0, preferSpeedOverMemory=True, progressCallback=<function donothing_callback at 0x2462cf8>, returnPxx=False)
\end{verbatim}

Call signature:

\begin{verbatim}
Cxy, Phase, freqs = cohere_pairs(X, ij, ...)
\end{verbatim}

Compute the coherence and phase for all pairs $ij$, in X.

$X$ is a numSamples * numCols array.

$ij$ is a list of tuples. Each tuple is a pair of indexes into the columns of X for which you want to compute coherence. For example, if X has 64 columns, and you want to compute all nonredundant pairs, define $ij$ as:

\begin{verbatim}
ij = []
for i in range(64):
    for j in range(i+1,64):
        ij.append( (i,j) )
\end{verbatim}

preferSpeedOverMemory is an optional bool. Defaults to true. If False, limits the caching by only making one, rather than two, complex cache arrays. This is useful if memory becomes critical. Even when preferSpeedOverMemory is False, cohere_pairs() will still give significant performance gains over calling cohere() for each pair, and will use substantially less memory than if preferSpeedOverMemory is True. In my tests with a 43000,64 array over all nonredundant pairs, preferSpeedOverMemory = True delivered a 33% performance boost on a 1.7GHZ Athlon with 512MB RAM compared with preferSpeedOverMemory = False. But both solutions were more than 10x faster than naively crunching all possible pairs through cohere().

Returns:

\begin{verbatim}
(Cxy, Phase, freqs)
\end{verbatim}

where:

- $Cxy$: dictionary of $(i, j)$ tuples -> coherence vector for that pair. I.e., $Cxy[(i,j)] = \text{cohere}(X[:,i], X[:,j])$. Number of dictionary keys is len(ij).
- Phase: dictionary of phases of the cross spectral density at each frequency for each pair. Keys are $(i, j)$.
- freqs: vector of frequencies, equal in length to either the coherence or phase vectors for any $(i, j)$ key.

e.g., to make a coherence Bode plot:

\begin{verbatim}
subplot(211)
pplot( freqs, Cxy[(12,19)])
\end{verbatim}
For a large number of pairs, `cohere_pairs()` can be much more efficient than just calling `cohere()` for each pair, because it caches most of the intensive computations. If \( N \) is the number of pairs, this function is \( O(N) \) for most of the heavy lifting, whereas calling cohere for each pair is \( O(N^2) \). However, because of the caching, it is also more memory intensive, making 2 additional complex arrays with approximately the same number of elements as \( X \).

See `test/cohere_pairs_test.py` in the src tree for an example script that shows that this `cohere_pairs()` and `cohere()` give the same results for a given pair.

See also:

- `psd()` For information about the methods used to compute \( P_{xy}, P_{xx} \) and \( P_{yy} \).

```
import matplotlib.pyplot as plt

import matplotlib.mlab

matplotlib.mlab.contiguous_regions(mask)
    return a list of (ind0, ind1) such that mask[ind0:ind1].all() is True and we cover all such regions
    TODO: this is a pure python implementation which probably has a much faster numpy impl

matplotlib.mlab.cross_from_above(x, threshold)
    return the indices into x where x crosses some threshold from below, eg the i's where:
        x[i-1]>threshold and x[i]<=threshold

See also:
    cross_from_below() and contiguous_regions()

matplotlib.mlab.cross_from_below(x, threshold)
    return the indices into x where x crosses some threshold from below, eg the i's where:
        x[i-1]<threshold and x[i]>=threshold

Example code:

```python
import matplotlib.pyplot as plt

t = np.arange(0.0, 2.0, 0.1)
s = np.sin(2*np.pi*t)

fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(t, s, '-o')
ax.axhline(0.5)
ax.axhline(-0.5)
ind = cross_from_below(s, 0.5)
ax.vlines(t[ind], -1, 1)
ind = cross_from_above(s, -0.5)
ax.vlines(t[ind], -1, 1)
```
plt.show()

See also:

cross_from_above() and contiguous_regions()

matplotlib.mlab.csd(x, y, NFFT=256, Fs=2, detrend=<function detrend_none at 0x24627d0>,
                       window=<function window_hanning at 0x2462578>, noverlap=0,
                       pad_to=None, sides='default', scale_by_freq=None)

The cross power spectral density by Welch’s average periodogram method. The vectors $x$ and $y$ are divided into $NFFT$ length blocks. Each block is detrended by the function $\text{detrend}$ and windowed by the function $\text{window}$. $\text{noverlap}$ gives the length of the overlap between blocks. 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 $\text{len}(x) < NFFT$ or $\text{len}(y) < NFFT$, they will be zero padded to $NFFT$.

$x, y$ Array or sequence containing the data

Keyword arguments:

$NFFT$: integer The number of data points used in each block for the FFT. Must be even; a power 2 is most efficient. The default value is 256. This should NOT be used to get zero padding, or the scaling of the result will be incorrect. Use $\text{pad_to}$ for this instead.

$Fs$: scalar The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, $\text{freqs}$, in cycles per time unit. The default value is 2.

detrend: callable The function applied to each segment before $\text{fft}$-ing, designed to remove the mean or linear trend. Unlike in MATLAB, where the $\text{detrend}$ parameter is a vector, in matplotlib it is a function. The $\text{pylab}$ module defines $\text{detrend_none()}$, $\text{detrend_mean()}$, and $\text{detrend_linear()}$, but you can use a custom function as well.

window: callable or ndarray A function or a vector of length $NFFT$. To create window vectors see $\text{window_hanning()}$, $\text{window_none()}$, $\text{numpy.blackman()}$, $\text{numpy.hamming()}$, $\text{numpy.bartlett()}$, $\text{scipy.signal()}$, $\text{scipy.signal.get_window()}$, etc. The default is $\text{window_hanning()}$. 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.

pad_to: integer 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 PSD (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 $\text{fft()}$. The default is None, which sets $\text{pad_to}$ equal to $NFFT$.

sides: [‘default’ | ‘onesided’ | ‘twosided’] Specifies which sides of the PSD to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. ‘onesided’ forces the return of a one-sided PSD, while ‘twosided’ forces two-sided.

scale_by_freq: boolean Specifies 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.
nooverlap: integer  The number of points of overlap between blocks. The default value is 0 (no overlap).

Returns the tuple \((Pxy, \text{freqs})\).


matplotlib.mlab.csv2rec(fname, comments='#', skiprows=0, checkrows=0, delimiter=',', converterd=None, names=None, missing='', missingd=None, use_mrecords=False, dayfirst=False, yearfirst=False)

Load data from comma/space/tab delimited file in \textit{fname} into a numpy record array and return the record array.

If names is None, a header row is required to automatically assign the recarray names. The headers will be lower cased, spaces will be converted to underscores, and illegal attribute name characters removed. If names is not None, it is a sequence of names to use for the column names. In this case, it is assumed there is no header row.

- \textit{fname}: can be a filename or a file handle. Support for gzipped files is automatic, if the filename ends in `.gz`
- \textit{comments}: the character used to indicate the start of a comment in the file, or None to switch off the removal of comments
- \textit{skiprows}: is the number of rows from the top to skip
- \textit{checkrows}: is the number of rows to check to validate the column data type. When set to zero all rows are validated.
- \textit{converterd}: if not None, is a dictionary mapping column number or munged column name to a converter function.
- \textit{names}: if not None, is a list of header names. In this case, no header will be read from the file
- \textit{missingd} is a dictionary mapping munged column names to field values which signify that the field does not contain actual data and should be masked, e.g., ‘0000-00-00’ or ‘unused’
- \textit{missing}: a string whose value signals a missing field regardless of the column it appears in
- \textit{use_mrecords}: if True, return an mrecords.fromrecords record array if any of the data are missing
- \textit{dayfirst}: default is False so that MM-DD-YY has precedence over DD-MM-YY. See http://labix.org/python-dateutil#head-b95ce2094d189a89f80f5ae52a05b4ab7b41af47 for further information.
- \textit{yearfirst}: default is False so that MM-DD-YY has precedence over YY-MM-DD. See http://labix.org/python-dateutil#head-b95ce2094d189a89f80f5ae52a05b4ab7b41af47 for further information.

If no rows are found, None is returned – see examples/loadrec.py

matplotlib.mlab.csvformat_factory(format)

matplotlib.mlab.demean(x, axis=0)

Return x minus its mean along the specified axis

matplotlib.mlab.detrend(x, key=None)

matplotlib.mlab.detrend_linear(y)

Return y minus best fit line; ‘linear’ detrending
matplotlib.mlab.detrend_mean(x)
    Return x minus the mean(x)

matplotlib.mlab.detrend_none(x)
    Return x: no detrending

matplotlib.mlab.dist(x, y)
    Return the distance between two points.

matplotlib.mlab.dist_point_to_segment(p, s0, s1)
    Get the distance of a point to a segment.
    
    p, s0, s1 are xy sequences
    
    This algorithm from http://softsurfer.com/Archive/algorithm_0102/algorithm_0102.htm#Distance%20to%20Ray%20or%20Segment

matplotlib.mlab.distances_along_curve(X)
    Computes the distance between a set of successive points in N dimensions.
    
    Where X is an M x N array or matrix. The distances between successive rows is computed. Distance is the standard Euclidean distance.

matplotlib.mlab.donething_callback(*args)

matplotlib.mlab.entropy(y, bins)
    Return the entropy of the data in y.
    
    $\sum p_i \log_2(p_i)$
    
    where $p_i$ is the probability of observing y in the $i^{th}$ bin of bins. bins can be a number of bins or a range of bins; see numpy.histogram().
    
    Compare S with analytic calculation for a Gaussian:
    
    \[
x = \mu + \sigma \cdot \text{randn}(200000)
    \]
    
    \[
    S_{\text{analytic}} = 0.5 \cdot (1.0 + \log(2\pi\sigma^2))
    \]

matplotlib.mlab.exp_safe(x)
    Compute exponentials which safely underflow to zero.
    
    Slow, but convenient to use. Note that numpy provides proper floating point exception handling with access to the underlying hardware.

matplotlib.mlab.fftsurr(x, detrend=<function detrend_none at 0x24627d0>, window=<function window_none at 0x24625f0>)
    Compute an FFT phase randomized surrogate of x.

matplotlib.mlab.find(condition)
    Return the indices where ravel(condition) is true

matplotlib.mlab.frange([start], stop[, step, keywords]) → array of floats
    Return a numpy ndarray containing a progression of floats. Similar to numpy.arange(), but defaults to a closed interval.
    
    frange(x0, x1) returns [x0, x0+1, x0+2, ..., x1]; start defaults to 0, and the endpoint is included. This behavior is different from that of range() and numpy.arange(). This is deliberate, since frange() will probably be more useful for generating lists of points for function evaluation,
and endpoints are often desired in this use. The usual behavior of \texttt{range()} can be obtained by setting
the keyword \texttt{closed} = 0, in this case, \texttt{frange()} basically becomes :func:`numpy.arange`.

When \texttt{step} is given, it specifies the increment (or decrement). All arguments can be floating point
numbers.

\texttt{frange(x0,x1,d)} returns \([x0,x0+d,x0+2d,\ldots,x_{\text{fin}}]\) where \(x_{\text{fin}} \leq x1\).

\texttt{frange()} can also be called with the keyword \texttt{npts}. This sets the number of points the list should
contain (and overrides the value \texttt{step} might have been given). \texttt{numpy.arange()} doesn’t offer this
option.

Examples:

```python
>>> frange(3)
array([ 0., 1., 2., 3.])
>>> frange(3,closed=0)
array([ 0., 1., 2.])
>>> frange(1,6,2)
array([1, 3, 5]) or 1,3,5,7, depending on floating point vagueries
>>> frange(1,6.5,npts=5)
array([ 1. , 2.375, 3.75 , 5.125, 6.5 ])
```

\texttt{matplotlib.mlab.get_formatd}(*r*, \texttt{formatd}=None)

build a formatd guaranteed to have a key for every dtype name

\texttt{matplotlib.mlab.get_sparse_matrix(M, N, frac=0.1)}

Return a \(M \times N\) sparse matrix with \(frac\) elements randomly filled.

\texttt{matplotlib.mlab.get_xyz_where(Z, Cond)}

\(Z\) and \(Cond\) are \(M \times N\) matrices. \(Z\) are data and \(Cond\) is a boolean matrix where some condition is
satisfied. Return value is \((x, y, z)\) where \(x\) and \(y\) are the indices into \(Z\) and \(z\) are the values of \(Z\) at
those indices. \(x, y,\) and \(z\) are 1D arrays.

\texttt{matplotlib.mlab.griddata(x, y, z, xi, yi, interp=’nn’)}

\(zi = griddata(x,y,z,xi,yi)\) fits a surface of the form \(z = f(*)x, y\) to the data in the (usually)
nonuniformly spaced vectors \((x, y, z)\). \texttt{griddata()} interpolates this surface at the points specified
by \((xi, yi)\) to produce \(zi\). \(xi\) and \(yi\) must describe a regular grid, can be either 1D or 2D, but must be
monotonically increasing.

A masked array is returned if any grid points are outside convex hull defined by input data (no extrap-
olation is done).

If interp keyword is set to ‘\texttt{nn}’ (default), uses natural neighbor interpolation based on Delaunay trian-
gulation. By default, this algorithm is provided by the \texttt{matplotlib.delaunay} package, written by
Robert Kern. The triangulation algorithm in this package is known to fail on some nearly pathological
cases. For this reason, a separate toolkit (\texttt{mpl_toolkits.natgrid}) has been created that provides a
more robust algorithm for triangulation and interpolation. This toolkit is based on the NCAR nat-
grid library, which contains code that is not redistributable under a BSD-compatible license. When
installed, this function will use the \texttt{mpl_toolkits.natgrid} algorithm, otherwise it will use the
built-in \texttt{matplotlib.delaunay} package.

If the interp keyword is set to ‘\texttt{linear}’, then linear interpolation is used instead of natural neigh-
bor. In this case, the output grid is assumed to be regular with a constant grid spacing in both the
x and y directions. For regular grids with nonconstant grid spacing, you must use natural neighbor interpolation. Linear interpolation is only valid if matplotlib.delaunay package is used - mpl_toolkits.natgrid only provides natural neighbor interpolation.

The natgrid matplotlib toolkit can be downloaded from http://sourceforge.net/project/showfiles.php?group_id=80706&package_id=142792

```
matplotlib.mlab.identity(n, rank=2, dtype='l', typecode=None)
```

Returns the identity matrix of shape \((n, n, \ldots, n)\) (rank \(r\)).

For ranks higher than 2, this object is simply a multi-index Kronecker delta:

\[
\begin{bmatrix}
1 & \text{if } i_0=i_1=\ldots=i_R, \\
0 & \text{otherwise}
\end{bmatrix}
\]

Optionally a dtype (or typecode) may be given (it defaults to ‘l’).

Since rank defaults to 2, this function behaves in the default case (when only \(n\) is given) like numpy.identity\((n)\) – but surprisingly, it is much faster.

```
matplotlib.mlab.inside_poly(points, verts)
```

points is a sequence of \(x, y\) points. verts is a sequence of \(x, y\) vertices of a polygon.

Return value is a sequence of indices into points for the points that are inside the polygon.

```
matplotlib.mlab.is_closed_polygon(X)
```

Tests whether first and last object in a sequence are the same. These are presumably coordinates on a polygonal curve, in which case this function tests if that curve is closed.

```
matplotlib.mlab.ispower2(n)
```

Returns the log base 2 of \(n\) if \(n\) is a power of 2, zero otherwise.

Note the potential ambiguity if \(n == 1\): \(2**0 == 1\), interpret accordingly.

```
matplotlib.mlab.isvector(X)
```

Like the MATLAB function with the same name, returns True if the supplied numpy array or matrix \(X\) looks like a vector, meaning it has a one non-singleton axis (i.e., it can have multiple axes, but all must have length 1, except for one of them).

If you just want to see if the array has 1 axis, use \(X.ndim == 1\).

```
matplotlib.mlab.l1norm(a)
```

Return the \(l_1\) norm of \(a\), flattened out.

Implemented as a separate function (not a call to norm\() for speed).

```
matplotlib.mlab.l2norm(a)
```

Return the \(l_2\) norm of \(a\), flattened out.

Implemented as a separate function (not a call to norm\() for speed).

```
matplotlib.mlab.less_simple_linear_interpolation(x, y, xi, extrap=False)
```

This function provides simple (but somewhat less so than cbook.simple_linear_interpolation()) linear interpolation. simple_linear_interpolation\() will give a list of point between a start and an end, while this does true linear interpolation at an arbitrary set of points.
This is very inefficient linear interpolation meant to be used only for a small number of points in relatively non-intensive use cases. For real linear interpolation, use scipy.

```python
def levypdf(x, gamma, alpha):
    """Return the levy pdf evaluated at x for params gamma, alpha"""
```

```python
def log2(x, ln2=0.6931471805599453):
    """Return the log(x) in base 2. This is a _slow_ function but which is guaranteed to return the correct integer value if the input is an integer exact power of 2."""
```

```python
def logspace(xmin, xmax, N):
    """"""
```

```python
def longest_contiguous_ones(x):
    """Return the indices of the longest stretch of contiguous ones in x, assuming x is a vector of zeros and ones. If there are two equally long stretches, pick the first."""
```

```python
def longest_ones(x):
    """alias for longest_contiguous_ones"""
```

```python
def movavg(x, n):
    """Compute the len(n) moving average of x."""
```

```python
def norm_flat(a, p=2):
    """norm(a,p=2) -> l-p norm of a.flat"""
```

```python
    """Return the l-p norm of a, considered as a flat array. This is NOT a true matrix norm, since arrays of arbitrary rank are always flattened.
    
p can be a number or the string 'Infinity' to get the L-infinity norm."""
```

```python
def normpdf(x, *args):
    """Return the normal pdf evaluated at x; args provides mu, sigma"""
```

```python
def offset_line(y, yerr):
    """Offsets an array y by +/- an error and returns a tuple (y - err, y + err).
    
The error term can be:
    • A scalar. In this case, the returned tuple is obvious.
    • A vector of the same length as y. The quantities y +/- err are computed component-wise.
    • A tuple of length 2. In this case, yerr[0] is the error below y and yerr[1] is error above y. For example:
```

```python
from pylab import *
x = linspace(0, 2*pi, num=100, endpoint=True)
y = sin(x)
y_minus, y_plus = mlab.offset_line(y, 0.1)
plot(x, y)
fill_between(x, ym, y2=yp)
show()
```

```python
def path_length(X):
    """Computes the distance travelled along a polygonal curve in N dimensions."""
```
Where $X$ is an $M \times N$ array or matrix. Returns an array of length $M$ consisting of the distance along the curve at each point (i.e., the rows of $X$).

**matplotlib.mlab.poly_below**($xmin$, $xs$, $ys$)
Given a sequence of $xs$ and $ys$, return the vertices of a polygon that has a horizontal base at $xmin$ and an upper bound at the $ys$. $xmin$ is a scalar.

Intended for use with `matplotlib.axes.Axes.fill()`, eg:

```python
xv, yv = poly_below(0, x, y)
ax.fill(xv, yv)
```

**matplotlib.mlab.poly_between**($x$, $ylower$, $yupper$)
Given a sequence of $x$, $ylower$ and $yupper$, return the polygon that fills the regions between them. $ylower$ or $yupper$ can be scalar or iterable. If they are iterable, they must be equal in length to $x$.

Return value is $x$, $y$ arrays for use with `matplotlib.axes.Axes.fill()`.

**matplotlib.mlab.prctile**($x$, $p$=$(0.0, 25.0, 50.0, 75.0, 100.0)$)
Return the percentiles of $x$. $p$ can either be a sequence of percentile values or a scalar. If $p$ is a sequence, the $i$th element of the return sequence is the $p*(i)$-th percentile of $x$. If $p$ is a scalar, the largest value of $x$ less than or equal to the $p$ percentage point in the sequence is returned.

$p$ is either an array of percentiles in [0..100] or a scalar which indicates how many quantiles of data you want ranked.

**matplotlib.mlab.prctile_rank**($x$, $p$)
Return the rank for each element in $x$, return the rank 0..len($p$). e.g., if $p$ = (25, 50, 75), the return value will be a len($x$) array with values in [0,1,2,3] where 0 indicates the value is less than the 25th percentile, 1 indicates the value is $\geq$ the 25th and $< 50$th percentile, ... and 3 indicates the value is above the 75th percentile cutoff.

$p$ is either an array of percentiles in [0..100] or a scalar which indicates how many quantiles of data you want ranked.

**matplotlib.mlab.prepca**($P$, $frac$=$(0, 0.1, 0.2, 0.3)$)
WARNING: this function is deprecated – please see class PCA instead

Compute the principal components of $P$. $P$ is a (numVars, numObs) array. $frac$ is the minimum fraction of variance that a component must contain to be included.

Return value is a tuple of the form (Pcomponents, Trans, fracVar) where:
- **Pcomponents** : a (numVars, numObs) array
- **Trans** [the weights matrix, ie, Pcomponents = Trans *] $P$
- **fracVar** [the fraction of the variance accounted for by each] component returned

A similar function of the same name was in the MATLAB R13 Neural Network Toolbox but is not found in later versions; its successor seems to be called “processpcs”.

**matplotlib.mlab.psd**($x$, $NFFT$=256, $Fs$=2, $detrend$=\texttt{function detrend_none at 0x24627d0},
window=\texttt{function window_hanning at 0x2462578}, $noverlap$=0,
$pad_to$=None, $sides$='default', $scale_by_freq$=None)
The power spectral density by Welch’s average periodogram method. The vector $x$ is divided into $NFFT$ length blocks. Each block is detrended by the function $detrend$ and windowed by the function $window$. $noverlap$ gives the length of the overlap between blocks. The absolute(fft(block))*2 of each segment are averaged to compute $Pxx$, with a scaling to correct for power loss due to windowing.

If len($x$) < $NFFT$, it will be zero padded to $NFFT$. 

68.1. matplotlib.mlab
Array or sequence containing the data

Keyword arguments:

**NFFT**: integer The number of data points used in each block for the FFT. Must be even; a power 2 is most efficient. The default value is 256. This should NOT be used to get zero padding, or the scaling of the result will be incorrect. Use pad_to for this instead.

**Fs**: scalar The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, freqs, in cycles per time unit. The default value is 2.

**detrend**: callable 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 pylab module defines detrend_none(), detrend_mean(), and detrend_linear(), but you can use a custom function as well.

**window**: callable or ndarray 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. The default is window_hanning(). 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.

**pad_to**: integer 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 psd (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.

**sides**: [‘default’ | ‘onesided’ | ‘twosided’] Specifies which sides of the PSD to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. ‘onesided’ forces the return of a one-sided PSD, while ‘twosided’ forces two-sided.

**scale_by_freq**: boolean Specifies 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**: integer The number of points of overlap between blocks. The default value is 0 (no overlap).

Returns the tuple (Pxx, freqs).

Refs:


**matplotlib.mlab.quad2cubic(q0x, q0y, q1x, q1y, q2x, q2y)**

Converts a quadratic Bezier curve to a cubic approximation.

The inputs are the x and y coordinates of the three control points of a quadratic curve, and the output is a tuple of x and y coordinates of the four control points of the cubic curve.

**matplotlib.mlab.rec2csv(r, fname, delimiter=', ', formatd=None, missing='', missingd=None, withheader=True)**

Save the data from numpy recarray r into a comma-/space-/tab-delimited file. The record array dtype
names will be used for column headers.

**fname**: can be a filename or a file handle. Support for gzipped files is automatic, if the filename ends in `.gz`

**withheader**: if withheader is False, do not write the attribute names in the first row for format type FormatFloat, we override the precision to store full precision floats in the CSV file

**See also:**

**csv2rec()** For information about missing and missingd, which can be used to fill in masked values into your CSV file.

**matplotlib.mlab.rec2txt(r, header=None, padding=3, precision=3, fields=None)**

Returns a textual representation of a record array.

- **r**: numpy recarray
- **header**: list of column headers
- **padding**: space between each column
- **precision**: number of decimal places to use for floats. Set to an integer to apply to all floats. Set to a list of integers to apply precision individually. Precision for non-floats is simply ignored.
- **fields**: if not None, a list of field names to print. fields can be a list of strings like ['field1', 'field2'] or a single comma separated string like ‘field1,field2’

Example:

```python
precision=[0,2,3]
```

Output:

<table>
<thead>
<tr>
<th>ID</th>
<th>Price</th>
<th>Return</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABC</td>
<td>12.54</td>
<td>0.234</td>
</tr>
<tr>
<td>XYZ</td>
<td>6.32</td>
<td>-0.076</td>
</tr>
</tbody>
</table>

**matplotlib.mlab.rec_append_fields(rec, names, arrs, dtypes=None)**

Return a new record array with field names populated with data from arrays in arrs. If appending a single field, then names, arrs and dtypes do not have to be lists. They can just be the values themselves.

**matplotlib.mlab.rec_drop_fields(rec, names)**

Return a new numpy record array with fields in names dropped.

**matplotlib.mlab.rec_groupby(r, groupby, stats)**

- **r**: a numpy record array
- **groupby**: is a sequence of record array attribute names that together form the grouping key. eg (‘date’, ‘productcode’)
- **stats**: is a sequence of (attr, func, outname) tuples which will call `x = func(attr)` and assign `x` to the record array output with attribute outname. For example:

```python
stats = ( ('sales', len, 'numsales'), ('sales', np.mean, 'avgsale') )
```
Return record array has dtype names for each attribute name in the the groupby argument, with the associated group values, and for each outname name in the stats argument, with the associated stat summary output.

```python
matplotlib.mlab.rec_join(key, r1, r2, jointype='inner', defaults=None, r1postfix='1', r2postfix='2')
```

Join record arrays r1 and r2 on key; key is a tuple of field names – if key is a string it is assumed to be a single attribute name. If r1 and r2 have equal values on all the keys in the key tuple, then their fields will be merged into a new record array containing the intersection of the fields of r1 and r2.

r1 (also r2) must not have any duplicate keys.

The jointype keyword can be ‘inner’, ‘outer’, ‘leftouter’. To do a rightouter join just reverse r1 and r2.

The defaults keyword is a dictionary filled with {column_name:default_value} pairs.

The keywords r1postfix and r2postfix are postfixed to column names (other than keys) that are both in r1 and r2.

```python
matplotlib.mlab.rec_keep_fields(rec, names)
```

Return a new numpy record array with only fields listed in names.

```python
matplotlib.mlab.rec_summarize(r, summaryfuncs)
```

r is a numpy record array

summaryfuncs is a list of (attr, func, outname) tuples which will apply func to the the array r*[attr] and assign the output to a new attribute name *outname. The returned record array is identical to r, with extra arrays for each element in summaryfuncs.

```python
matplotlib.mlab.recs_join(key, name, recs, jointype='outer', missing=0.0, postfixes=None)
```

Join a sequence of record arrays on single column key.

This function only joins a single column of the multiple record arrays

key is the column name that acts as a key

name is the name of the column that we want to join

recs is a list of record arrays to join

jointype is a string ‘inner’ or ‘outer’

missing is what any missing field is replaced by

postfixes if not None, a len recs sequence of postfixes returns a record array with columns [rowkey, name0, name1, ... namen-1]. or if postfixes [PF0, PF1, ..., PFN-1] are supplied, [rowkey, namePF0, namePF1, ... namePFN-1].

Example:

```python
r = recs_join("date", "close", recs=[r0, r1], missing=0.)
```

```python
matplotlib.mlab.rk4(derivs, y0, t)
```

Integrate 1D or ND system of ODEs using 4-th order Runge-Kutta. This is a toy implementation which may be useful if you find yourself stranded on a system w/o scipy. Otherwise use scipy.integrate().

y0 initial state vector

t sample times
derivs returns the derivative of the system and has the signature dy = derivs(yi, ti)

Example 1

## 2D system

```python
def derivs6(x, t):
    d1 = x[0] + 2*x[1]
    d2 = -3*x[0] + 4*x[1]
    return (d1, d2)
dt = 0.0005
T = arange(0.0, 2.0, dt)
y0 = (1, 2)
yout = rk4(derivs6, y0, T)
```

Example 2:

## 1D system

```python
alpha = 2
def derivs(x, t):
    return -alpha*x + exp(-t)
y0 = 1
yout = rk4(derivs, y0, T)
```

If you have access to scipy, you should probably be using the scipy.integrate tools rather than this function.

```python
matplotlib.mlab.rms_flat(a)
return the root mean square of all the elements of a, flattened out.
```

```python
matplotlib.mlab.safe_isinf(x)
numpy.isinf() for arbitrary types
```

```python
matplotlib.mlab.safe_isnan(x)
numpy.isnan() for arbitrary types
```

```python
matplotlib.mlab.segments_intersect(s1, s2)
return True if s1 and s2 intersect. s1 and s2 are defined as:

s1: (x1, y1), (x2, y2)
s2: (x3, y3), (x4, y4)
```

```python
matplotlib.mlab.slopes(x, y)
slopes() calculates the slope y'(x)
The slope is estimated using the slope obtained from that of a parabola through any three consecutive points.
```

This method should be superior to that described in the appendix of A CONSISTENTLY WELL BEHAVED METHOD OF INTERPOLATION by Russel W. Stineman (Creative Computing July 1980) in at least one aspect:
Circles for interpolation demand a known aspect ratio between $x$- and $y$-values. For many functions, however, the abscissa are given in different dimensions, so an aspect ratio is completely arbitrary.

The parabola method gives very similar results to the circle method for most regular cases but behaves much better in special cases.

Norbert Nemec, Institute of Theoretical Physics, University or Regensburg, April 2006 Norbert.Nemec at physik.uni-regensburg.de

(inspired by a original implementation by Halldor Bjornsson, Icelandic Meteorological Office, March 2006 halldor at vedur.is)

```python
matplotlib.mlab.specgram(x, NFFT=256, Fs=2, detrend=<function detrend_none at 0x24627d0>, window=<function window_hanning at 0x2462578>, noverlap=128, pad_to=None, sides='default', scale_by_freq=None)
```

Compute a spectrogram of data in $x$. Data are split into $NFFT$ length segments and the PSD 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 $noverlap$.

If $x$ is real (i.e. non-complex) only the spectrum of the positive frequencies is returned. If $x$ is complex then the complete spectrum is returned.

Keyword arguments:

- **$NFFT$: integer** The number of data points used in each block for the FFT. Must be even; a power 2 is most efficient. The default value is 256. This should NOT be used to get zero padding, or the scaling of the result will be incorrect. Use $pad_to$ for this instead.

- **$Fs$: scalar** The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, fregs, in cycles per time unit. The default value is 2.

- **$detrend$: callable** 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 matplotib it is a function. The pylab module defines $detrend_none()$, $detrend_mean()$, and $detrend_linear()$, but you can use a custom function as well.

- **$window$: callable or ndarray** 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. The default is $window_hanning()$. 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.

- **$pad_to$: integer** 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 psd (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 $NFFT$.

- **$sides$: [‘default’ | ‘onesided’ | ‘twosided’]** Specifies which sides of the PSD to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. ‘onesided’ forces the return of a one-sided PSD, while
'twosided' forces two-sided.

**scale_by_freq**: boolean Specify 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**: integer The number of points of overlap between blocks. The default value is 128.

Returns a tuple (Pxx, freqs, t):
- Pxx: 2-D array, columns are the periodograms of successive segments
- freqs: 1-D array of frequencies corresponding to the rows in Pxx
- t: 1-D array of times corresponding to midpoints of segments.

See also:

psd() psd() differs in the default overlap; in returning the mean of the segment periodograms; and in not returning times.

```python
def stineman_interp(xi, x, y, yp=None):
    # Implementation of Stineman interpolation
```

Here's an example that generates a coarse sine curve, then interpolates over a finer abscissa:

```python
x = linspace(0, 2*pi, 20); y = sin(x); yp = cos(x)
xi = linspace(0, 2*pi, 40);
yi = stineman_interp(xi, x, y, yp);
plot(x, y, 'o', xi, yi)
```

The interpolation method is described in the article A CONSISTENTLY WELL BEHAVED METHOD OF INTERPOLATION by Russell W. Stineman. The article appeared in the July 1980 issue of Creative Computing with a note from the editor stating that while they were:

- not an academic journal but once in a while something serious and original comes in
- adding that this was "apparently a real solution" to a well known problem.

For yp = None, the routine automatically determines the slopes using the slopes() routine.

x is assumed to be sorted in increasing order.

For values xi[i] < x[0] or xi[i] > x[-1], the routine tries an extrapolation. The relevance of the data obtained from this, of course, is questionable...

Original implementation by Halldor Bjornsson, Icelandic Meteorolocial Office, March 2006 halldor at vedur.is

Completely reworked and optimized for Python by Norbert Nemec, Institute of Theoretical Physics, University or Regensburg, April 2006 Norbert.Nemec at physik.uni-regensburg.de

```python
def vector_lengths(X, P=2.0, axis=None):
    # Implementation of vector length calculation
```

Finds the length of a set of vectors in n dimensions. This is like the numpy.norm() function for vectors, but has the ability to work over a particular axis of the supplied array or matrix.

Computes \( \sum((x_i)^P)^{1/P} \) for each \( x_i \) being the elements of \( X \) along the given axis. If axis is None, compute over all elements of \( X \).
matplotlib.mlab.window_hanning(x)
    return x times the hanning window of len(x)

matplotlib.mlab.window_none(x)
    No window function; simply return x
69.1 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 Paths somewhere in the drawing pipeline.

Whilst a Path instance itself cannot be drawn, there exists Artist subclasses which can be used for convenient Path visualisation - the two most frequently used of these are PathPatch and PathCollection.

class matplotlib.path.Path(vertices, codes=None, _interpolation_steps=1, closed=False, read-only=False)

Bases: object

Path represents 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

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.

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 is important, since many Path objects, as an optimization, do not store a codes at all, but have a default one provided for them by iter_segments().

Note: The vertices and codes arrays should be treated as immutable – there are a number of optimiza-
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**: \{None, array_like\}, 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.

- **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 an arc on the unit circle from angle *theta1* to angle *theta2* (in degrees).

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.*
**cleaned***(transform=None, remove_nans=False, clip=None, quantize=False, simplify=False, curves=False, stroke_width=1.0, snap=False, sketch=None)***

Cleans up the path according to the parameters returning a new Path instance.

See also:

See *iter_segments()* for details of the keyword arguments.

**Returns** Path instance with cleaned up vertices and codes.

**clip_to_bbox***(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 uint8

**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***(path, transform=None)***

Returns True if this path completely contains the given path.

If *transform* is not None, the path will be transformed before performing the test.

**contains_point***(point, transform=None, radius=0.0)***

Returns True if the path contains the given point.

If *transform* is not None, the path will be transformed before performing the test.

*radius* allows the path to be made slightly larger or smaller.

**contains_points***(points, transform=None, radius=0.0)***

Returns a bool array which is True if the path contains the corresponding point.

If *transform* is not None, the path will be transformed before performing the test.

*radius* allows the path to be made slightly larger or smaller.

**copy**()

Returns a shallow copy of the Path, which will share the vertices and codes with the source Path.

**deepcopy**()

Returns a deepcopy of the Path. The Path will not be readonly, even if the source Path is.

**get_extents***(transform=None)***

Returns the extents (xmin, ymin, xmax, ymax) of the path.

Unlike computing the extents on the vertices alone, this algorithm will take into account the curves and deal with control points appropriately.
has_nonfinite
True if the vertices array has nonfinite values.

classmethod 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(steps)
Returns a new path resampled to length N x steps. Does not currently handle interpolating curves.

intersects_bbox(bbox, filled=True)
Returns True if this path intersects a given Bbox.

filled, when True, treats the path as if it was filled. That is, if one path completely encloses the other, intersects_path() will return True.

intersects_path(other, filled=True)
Returns True if this path intersects another given path.

filled, when True, treats the paths as if they were filled. That is, if one path completely encloses the other, intersects_path() will return True.

iter_segments(transform=None, remove_nans=True, clip=None, snap=False, stroke_width=1.0, simplify=None, curves=True, sketch=None)
Iterates over all of the curve segments in the path. Each iteration returns a 2-tuple (vertices, code), where vertices is a sequence of 1 - 3 coordinate pairs, and code is one of the Path codes.

Additionally, this method can provide a number of standard cleanups and conversions to the path.

Parameters transform : None or Transform instance
If not None, the given affine transformation will be applied to the path.

remove_nans : {False, True}, optional
If True, will remove all NaNs from the path and insert MOVETO commands to skip over them.

clip : None or sequence, 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 None, auto-snap to pixels, to reduce fuzziness of rectilinear lines. If True, force snapping, and if False, don’t snap.

stroke_width : float, optional
The width of the stroke being drawn. Needed as a hint for the snapping algorithm.

simplify : None or bool, optional
If True, perform simplification, to remove vertices that do not affect the appearance of the path. If False, perform no simplification. If None, use the should_simplify member variable.

curves : {True, False}, 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.

**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

**readonly**
  True if the Path is read-only.

**should_simplify**
  True if the vertices array should be simplified.

**simplify_threshold**
  The fraction of a pixel difference below which vertices will be simplified out.

**to_polygons(transform=None, width=0, height=0)**
  Convert this path to a list of polygons. Each polygon 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, such as GDK.
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.

**transformed(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 a `Path` of the unit circle. The circle is approximated using cubic Bezier curves. This uses 8 splines around the circle using the approach presented here:

Lancaster, Don. Approximating a Circle or an Ellipse Using Four Bezier Cubic Splines.

**classmethod** `unit_circle_righthalf()`

Return a `Path` of the right half of a unit circle. The circle is approximated using cubic Bezier curves. This uses 4 splines around the circle using the approach presented here:

Lancaster, Don. Approximating a Circle or an Ellipse Using Four Bezier Cubic Splines.

**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 and radius of 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).

**vertices**

The list of vertices in the `Path` as an Nx2 numpy array.

**classmethod** `wedge(theta1, theta2, n=None)`

Return a wedge of the unit circle from angle `theta1` to angle `theta2` (in degrees).

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`.

`matplotlib.path.cleanup_path(*args, **kwargs)`

Deprecated since version 1.3: The cleanup_path function was deprecated in version 1.3. Use `path.Path.cleaned` instead.

`cleanup_path(path, trans, remove_nans, clip, snap, simplify, curves, sketch_params)`

`matplotlib.path.clip_path_to_rect(*args, **kwargs)`

Deprecated since version 1.3: The clip_path_to_rect function was deprecated in version 1.3. Use
Matplotlib, Release 1.3.0

path.Path.clip_to_bbox instead.
clip_path_to_rect(path, bbox, inside)

matplotlib.path.convert_path_to_polygons(*args, **kwargs)
Deprecation since version 1.3: The convert_path_to_polygons function was deprecated in version 1.3. Use path.Path.to_polygons instead.
convert_path_to_polygons(path, trans, width, height)

matplotlib.path.get_path_collection_extents(master_transform, paths, transforms, off-sets, offset_transform)
Given a sequence of Path objects, Transform objects and offsets, as found in a PathCollection, returns the bounding box that encapsulates all of them.
master_transform is a global transformation to apply to all paths
paths is a sequence of Path instances.
transforms is a sequence of Affine2D instances.
offsets is a sequence of (x, y) offsets (or an Nx2 array)
offset_transform is a Affine2D to apply to the offsets before applying the offset to the path.
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)

matplotlib.path.get_path_extents(*args, **kwargs)
Deprecation since version 1.3: The get_path_extents function was deprecated in version 1.3. Use path.Path.get_extents instead.
get_path_extents(path, trans)

matplotlib.path.get_paths_extents(paths, transforms=[]) Given a sequence of Path objects and optional Transform objects, returns the bounding box that encapsulates all of them.
paths is a sequence of Path instances.
transforms is an optional sequence of Affine2D instances to apply to each path.

matplotlib.path.path_in_path(*args, **kwargs)
Deprecation since version 1.3: The path_in_path function was deprecated in version 1.3. Use path.Path.contains_path instead.
path_in_path(a, atrans, b, btrans)

matplotlib.path.path_intersects_path(*args, **kwargs)
Deprecation since version 1.3: The path_intersects_path function was deprecated in version 1.3. Use path.Path.intersects_path instead.
path_intersects_path(p1, p2)


```python
matplotlib.path.point_in_path(*args, **kwargs)
    Deprecated since version 1.3: The point_in_path function was deprecated in version 1.3. Use path.Path.contains_point instead.
    point_in_path(x, y, path, trans)

matplotlib.path.point_in_path_collection(*args, **kwargs)
    Deprecated since version 1.3: The point_in_path_collection function was deprecated in version 1.3. Use collection.Collection.contains instead.
    point_in_path_collection(x, y, r, trans, paths, transforms, offsets, offsetTrans, filled)

matplotlib.path.points_in_path(*args, **kwargs)
    Deprecated since version 1.3: The points_in_path function was deprecated in version 1.3. Use path.Path.contains_points instead.
    points_in_path(points, path, trans)
```
70.1 matplotlib.pyplot

Provides a MATLAB-like plotting framework.

pylab combines pyplot with numpy into a single namespace. This is convenient for interactive work, but for programming it is recommended that the namespaces be kept separate, e.g.:

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

x = np.arange(0, 5, 0.1);
y = np.sin(x)
plt.plot(x, y)

matplotlib.pyplot.acorr(x, hold=None, **kwargs)
```

Plot the autocorrelation of `x`.

Call signature:

```python
acorr(x, normed=True, detrend=mlab.detrend_none, usevlines=True, maxlags=10, **kwargs)
```

If `normed = True`, normalize the data by the autocorrelation at 0-th lag. `x` is detrended by the `detrend` callable (default no normalization).

Data are plotted as `plot(lags, c, **kwargs)`

Return value is a tuple `(lags, c, line)` where:

- `lags` are a length 2*maxlags+1 lag vector
- `c` is the 2*maxlags+1 auto correlation vector
- `line` is a Line2D instance returned by `plot()`

The default linestyle is None and the default marker is 'o', though these can be overridden with keyword args. The cross correlation is performed with `numpy.correlate()` with `mode = 2`.

If `usevlines = True`, `vlines()` rather than `plot()` is used to draw vertical lines from the origin to the acorr. Otherwise, the plot style is determined by the kwargs, which are Line2D properties.

`maxlags` is a positive integer detailing the number of lags to show. The default value of `None` will return all (2*len(x)-1) lags.
The return value is a tuple \((lags, c, linecol, b)\) where

- \(linecol\) is the \texttt{LineCollection}
- \(b\) is the \(x\)-axis.

\textbf{See also:}

\texttt{plot()} or \texttt{vlines()} For documentation on valid kwargs.

\textbf{Example:}

\texttt{xcorr()} is top graph, and \texttt{acorr()} is bottom graph.

Additional kwargs: \texttt{hold} = [True|False] overrides default hold state

\texttt{matplotlib.pyplot.annotate(**args, **kwargs)}

Create an annotation: a piece of text referring to a data point.

\textbf{Call signature:}

\texttt{annotate(s, xy, xytext=\texttt{None}, xycoords='data',}
\texttt{ textcoords='data', arrowprops=\texttt{None}, **kwargs)}

\textbf{Keyword arguments:}

Annotate the \(x, y\) point \(xy\) with text \(s\) at \(x, y\) location \texttt{xytext}. (If \texttt{xytext} = \texttt{None}, defaults to \(xy\), and if \texttt{textcoords} = \texttt{None}, defaults to \texttt{xycoords}).
arrowprops, if not None, is a dictionary of line properties (see matplotlib.lines.Line2D) for the arrow that connects annotation to the point.

If the dictionary has a key arrowstyle, a FancyArrowPatch instance is created with the given dictionary and is drawn. Otherwise, a YAArow patch instance is created and drawn. Valid keys for YAArow are

<table>
<thead>
<tr>
<th>Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>width</td>
<td>the width of the arrow in points</td>
</tr>
<tr>
<td>frac</td>
<td>the fraction of the arrow length occupied by the head</td>
</tr>
<tr>
<td>head-width</td>
<td>the width of the base of the arrow head in points</td>
</tr>
<tr>
<td>shrink</td>
<td>oftentimes it is convenient to have the arrowtip and base a bit away from the text and point being annotated. If $d$ is the distance between the text and annotated point, shrink will shorten the arrow so the tip and base are shrink percent of the distance $d$ away from the endpoints. ie, shrink=0.05 is 5%</td>
</tr>
</tbody>
</table>

Valid keys for FancyArrowPatch are

<table>
<thead>
<tr>
<th>Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>arrowstyle</td>
<td>the arrow style</td>
</tr>
<tr>
<td>connectionstyle</td>
<td>the connection style</td>
</tr>
<tr>
<td>relpos</td>
<td>default is (0.5, 0.5)</td>
</tr>
<tr>
<td>patchA</td>
<td>default is bounding box of the text</td>
</tr>
<tr>
<td>patchB</td>
<td>default is None</td>
</tr>
<tr>
<td>shrinkA</td>
<td>default is 2 points</td>
</tr>
<tr>
<td>shrinkB</td>
<td>default is 2 points</td>
</tr>
<tr>
<td>mutation_scale</td>
<td>default is text size (in points)</td>
</tr>
<tr>
<td>mutation_aspect</td>
<td>default is 1.</td>
</tr>
</tbody>
</table>

? any key for matplotlib.patches.polygon

Valid keys for matplotlib.patches.PathPatch

xycoords and textcoords are strings that indicate the coordinates of xy and xytext.
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'figure points'</td>
<td>points from the lower left corner of the figure</td>
</tr>
<tr>
<td>'figure pixels'</td>
<td>pixels from the lower left corner of the figure</td>
</tr>
<tr>
<td>'figure fraction'</td>
<td>0,0 is lower left of figure and 1,1 is upper right</td>
</tr>
<tr>
<td>'axes points'</td>
<td>points from lower left corner of axes</td>
</tr>
<tr>
<td>'axes pixels'</td>
<td>pixels from lower left corner of axes</td>
</tr>
<tr>
<td>'axes fraction'</td>
<td>0,0 is lower left of axes and 1,1 is upper right</td>
</tr>
<tr>
<td>'data'</td>
<td>use the coordinate system of the object being annotated (default)</td>
</tr>
<tr>
<td>'offset points'</td>
<td>Specify an offset (in points) from the xy value</td>
</tr>
<tr>
<td>'polar'</td>
<td>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 &quot;data&quot; coordinate system.</td>
</tr>
</tbody>
</table>

If a 'points' or 'pixels' option is specified, values will be added to the bottom-left and if negative, values will be subtracted from the top-right. e.g.:  

```python  
# 10 points to the right of the left border of the axes and 
# 5 points below the top border  
xy=(10,-5), xycoords='axes points'  
```

You may use an instance of `Transform` or `Artist`. See `Annotating Axes` for more details.

The `annotation_clip` attribute controls the visibility of the annotation when it goes outside the axes area. If True, the annotation will only be drawn when the `xy` is inside the axes. If False, the annotation will always be drawn regardless of its position. The default is `None`, which behave as True only if `xycoords` is "data".

Additional kwargs are Text properties:
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>color</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>family or fontfamily or fontname or name</td>
<td>[FONTNAME</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fontproperties or font_properties</td>
<td>a matplotlib.font_manager.FontProperties instance</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>horizontalalignment or ha</td>
<td>['center'</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linespacing</td>
<td>float (multiple of font size)</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>multialignment</td>
<td>['left'</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None]float</td>
</tr>
<tr>
<td>position</td>
<td>(x,y)</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>rotation</td>
<td>[ angle in degrees</td>
</tr>
<tr>
<td>rotation_mode</td>
<td>unknown</td>
</tr>
<tr>
<td>size or fontsize</td>
<td>[size in points</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>stretch or fontstretch</td>
<td>[a numeric value in range 0-1000</td>
</tr>
<tr>
<td>style or fontstyle</td>
<td>['normal'</td>
</tr>
<tr>
<td>text</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>variant or fontvariant</td>
<td>['normal’</td>
</tr>
<tr>
<td>verticalalignment or va or ma</td>
<td>['center’</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>x</td>
<td>float</td>
</tr>
<tr>
<td>y</td>
<td>float</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>
matplotlib.pyplot.arrow(x, y, dx, dy, hold=None, **kwargs)

Add an arrow to the axes.
Call signature:

```
arrow(x, y, dx, dy, **kwargs)
```

Draws arrow on specified axis from \((x, y)\) to \((x + dx, y + dy)\). Uses FancyArrow patch to construct the arrow.

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()`.

Optional kwargs control the arrow construction and properties:

**Constructor arguments**

- `width`: float (default: 0.001) width of full arrow tail
- `length_includes_head`: [True | False] (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`: [True | False] (default: False) if True, the head starts being drawn at coordinate 0 instead of ending at coordinate 0.

Other valid kwargs (inherited from `Patch`) are:
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased  or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor   or ec</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>facecolor   or fc</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[‘/’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle   or ls</td>
<td>[‘solid’</td>
</tr>
<tr>
<td>linewidth   or lw</td>
<td>float or None for default</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

Example:
Additional kwargs: hold = [True|False] overrides default hold state

**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.

**enable:** [True | False | None] True (default) turns autoscaling on, False turns it off. None leaves the autoscaling state unchanged.

**axis:** ['x' | 'y' | 'both'] which axis to operate on; default is 'both'

**tight:** [True | False | None] If True, set view limits to data limits; if False, let the locator and margins expand the view limits; if None, use tight scaling if the only artist is an image, otherwise treat tight as False. The tight setting is retained for future autoscaling until it is explicitly changed.

Returns None.

**matplotlib.pyplot.autumn()**
set the default colormap to autumn and apply to current image if any. See help(colormaps) for more information

**matplotlib.pyplot.axes(***args, **kwargs)**
Add an axes to the figure.
The axes is added at position rect specified by:
- **axes()** by itself creates a default full subplot(111) window axis.
- **axes(rect, axisbg='w')** where rect = [left, bottom, width, height] in normalized (0, 1) units. axisbg is the background color for the axis, default white.
- **axes(h)** where h is an axes instance makes h the current axis. An Axes instance is returned.
Matplotlib, Release 1.3.0

<table>
<thead>
<tr>
<th>kwarg</th>
<th>Accepts</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>axisbg</td>
<td>color</td>
<td>the axes background color</td>
</tr>
<tr>
<td>frameon</td>
<td>[True</td>
<td>False]</td>
</tr>
<tr>
<td>sharex</td>
<td>otherax</td>
<td>current axes shares xaxis attribute with otherax</td>
</tr>
<tr>
<td>sharey</td>
<td>otherax</td>
<td>current axes shares yaxis attribute with otherax</td>
</tr>
<tr>
<td>polar</td>
<td>[True</td>
<td>False]</td>
</tr>
</tbody>
</table>

Examples:
• examples/pylab_examples/axes_demo.py places custom axes.
• examples/pylab_examples/shared_axis_demo.py uses sharex and sharey.

matplotlib.pyplot.axhline(y=0, xmin=0, xmax=1, hold=None, **kwargs)

Add a horizontal line across the axis.

Call signature:

axhline(y=0, xmin=0, xmax=1, **kwargs)

Draw a horizontal line at y from xmin to xmax. With the default values of xmin = 0 and xmax = 1, this line will always span the horizontal extent of the axes, regardless of the xlim settings, even if you change them, e.g., with the set_xlim() command. That is, the horizontal extent is in axes coords: 0=left, 0.5=middle, 1.0=right but the y location is in data coordinates.

Return value is the Line2D instance. kwargs are the same as kwargs to plot, and can be used to control the line properties. e.g.,
• 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 the middle half of the xrange:

        >>> axhline(y=.5, xmin=0.25, xmax=0.75)

Valid kwargs are Line2D properties, with the exception of ‘transform’:
Table 70.2 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dash_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>data</td>
<td>2D array (rows are x, y) or two 1D arrays</td>
</tr>
<tr>
<td>drawstyle</td>
<td>['default'</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>['full'</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with '%s' conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['-']</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>marker</td>
<td>unknown</td>
</tr>
<tr>
<td>markeredgcolor or mec</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markeredgewidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

See also:

- `axhspan()` for example plot and source code
  
  Additional kwargs: `hold = [True|False]` overrides default hold state

`matplotlib.pyplot.axhspan(ymin, ymax, xmin=0, xmax=1, hold=None, **kwargs)`

Add a horizontal span (rectangle) across the axis.

Call signature:
axhspan(ymin, ymax, xmin=0, xmax=1, **kwargs)

y coords are in data units and x coords are in axes (relative 0-1) units.

Draw a horizontal span (rectangle) from ymin to ymax. With the default values of xmin = 0 and xmax = 1, this always spans the xrange, regardless of the xlim settings, even if you change them, e.g., with the set_xlim() command. That is, the horizontal extent is in axes coords: 0=left, 0.5=middle, 1.0=right but the y location is in data coordinates.

Return value is a matplotlib.patches.Polygon instance.

Examples:

- draw a gray rectangle from y = 0.25-0.75 that spans the horizontal extent of the axes:

  ```python
  >>> axhspan(0.25, 0.75, facecolor='0.5', alpha=0.5)
  ```

Valid kwargs are Polygon properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[Path, Transform]</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or ec</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>facecolor or fc</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[’/’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>[‘solid’</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float or None for default</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

Example:
Additional kwargs: hold = [True|False] overrides default hold state

matplotlib.pyplot.axis(*v, **kwargs)
Convenience method to get or set axis properties.

Calling with no arguments:

```python
>>> axis()
```
returns the current axes limits [xmin, xmax, ymin, ymax].:

```python
>>> axis(v)
```
sets the min and max of the x and y axes, with v = [xmin, xmax, ymin, ymax].:

```python
>>> axis('off')
```
turns off the axis lines and labels.:

```python
>>> axis('equal')
```
changes limits of x or y axis so that equal increments of x and y have the same length; a circle is circular.:
achieves the same result by changing the dimensions of the plot box instead of the axis data limits:

changes \( x \) and \( y \) axis limits such that all data is shown. If all data is already shown, it will move it to the center of the figure without modifying \( (x_{\text{max}} - x_{\text{min}}) \) or \( (y_{\text{max}} - y_{\text{min}}) \). Note this is slightly different than in MATLAB:

is ‘scaled’ with the axis limits equal to the data limits:

and:

are deprecated. They restore default behavior; axis limits are automatically scaled to make the data fit comfortably within the plot box.

if \( \text{len(*v)} = 0 \), you can pass in \( x_{\text{min}}, x_{\text{max}}, y_{\text{min}}, y_{\text{max}} \) as kwargs selectively to alter just those limits without changing the others.

The \( x_{\text{min}}, x_{\text{max}}, y_{\text{min}}, y_{\text{max}} \) tuple is returned

See also:

\( \text{xlim()}, \text{ylim()} \) For setting the \( x \)- and \( y \)-limits individually.

Add a vertical line across the axes.

Call signature:

\[
\text{axvline}(x=0, ymin=0, ymax=1, hold=None, **kwargs)
\]

Draw a vertical line at \( x \) from \( y_{\text{min}} \) to \( y_{\text{max}} \). With the default values of \( y_{\text{min}} = 0 \) and \( y_{\text{max}} = 1 \), this line will always span the vertical extent of the axes, regardless of the ylim settings, even if you change them, e.g., with the \( \text{set_ylim()} \) command. That is, the vertical extent is in axes coords: 0=bottom, 0.5=middle, 1.0=top but the \( x \) location is in data coordinates.

Return value is the Line2D instance. kwargs are the same as kwargs to plot, and can be used to control the line properties. e.g.,

- draw a thick red vline at \( x = 0 \) that spans the yrange:

\[
>> \text{axvline}(\text{linewidth}=4, \text{color}='r')
\]

- draw a default vline at \( x = 1 \) that spans the yrange:
```python
>>> axvline(x=1)
• draw a default vline at x = .5 that spans the the middle half of the yrange:

```n```python
>>> axvline(x=.5, ymin=0.25, ymax=0.75)
```

Valid kwargs are Line2D properties, with the exception of ‘transform’:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>data</td>
<td>2D array (rows are x, y) or two 1D arrays</td>
</tr>
<tr>
<td>drawstyle</td>
<td>['default'</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>['full'</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['-'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>marker</td>
<td>unknown</td>
</tr>
<tr>
<td>markeredgecolor or mec</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markeredgewidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
</tbody>
</table>
Table 70.3 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

See also:

axhspan() for example plot and source code

Additional kwargs: hold = [True]False] overrides default hold state

matplotlib.pyplot.axvspan(xmin, xmax, ymin=0, ymax=1, hold=None, **kwargs)

Add a vertical span (rectangle) across the axes.

Call signature:

```python
axvspan(xmin, xmax, ymin=0, ymax=1, **kwargs)
```

x coords are in data units and y coords are in axes (relative 0-1) units.

Draw a vertical span (rectangle) from xmin to xmax. With the default values of ymin = 0 and ymax = 1, this always spans the yrange, regardless of the ylim settings, even if you change them, e.g., with the set_xlim() command. That is, the vertical extent is in axes coords: 0=bottom, 0.5=middle, 1.0=top but the y location is in data coordinates.

Return value is the matplotlib.patches.Polygon instance.

Examples:

- draw a vertical green translucent rectangle from x=1.25 to 1.55 that spans the yrange of the axes:

  ```python
  >>> axvspan(1.25, 1.55, facecolor='g', alpha=0.5)
  ```

Valid kwargs are Polygon properties:
## Property Description

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an <code>Axes</code> instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or ec</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>facecolor or fc</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>['/||||---</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['solid’</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float or None for default</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td><code>Transform</code> instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

See also:

`axhspan()` for example plot and source code

Additional kwargs: `hold = [True|False]` overrides default hold state

### `matplotlib.pyplot.bar(left, height, width=0.8, bottom=None, hold=None, **kwargs)`

Make a bar plot.

Make a bar plot with rectangles bounded by:

`left, left + width, bottom, bottom + height` (left, right, bottom and top edges)

#### Parameters

- `left` : sequence of scalars
  
  the x coordinates of the left sides of the bars

- `height` : sequence of scalars
  
  the heights of the bars

- `width` : scalar or array-like, optional, default: 0.8
  
  the width(s) of the bars

- `bottom` : scalar or array-like, optional, default: None
the y coordinate(s) of the bars

color : scalar or array-like, optional
    the colors of the bars
edgcolor : scalar or array-like, optional
    the colors of the bar edges
linewidth : scalar or array-like, optional, default: None
    width of bar edge(s). If None, use default linewidth; If 0, don’t draw edges.
xerr : scalar or array-like, optional, default: None
    if not None, will be used to generate errorbar(s) on the bar chart
yerr : scalar or array-like, optional, default: None :
    if not None, will be used to generate errorbar(s) on the bar chart
color : scalar or array-like, optional, default: None
    specifies the color of errorbar(s)
capsize : integer, optional, default: 3
    determines the length in points of the error bar caps

error_kw :
    dictionary of kwargs to be passed to errorbar method. ecolor and capsize may be specified here rather than as independent kwargs.

align : ['edge' | 'center'], optional, default: ‘edge’
    If edge, aligns bars by their left edges (for vertical bars) and by their bottom edges (for horizontal bars). If center, interpret the left argument as the coordinates of the centers of the bars.
orientation : ‘vertical’ | ‘horizontal’, optional, default: ‘vertical’
    The orientation of the bars.

log : boolean, optional, default: False
    If true, sets the axis to be log scale

Returns :class:`matplotlib.patches.Rectangle` instances.

Notes

The optional arguments color, edgcolor, linewidth, xerr, and yerr can be either scalars or sequences of length equal to the number of bars. This enables you to use bar as the basis for stacked bar charts, or candlestick plots. Detail: xerr and yerr are passed directly to errorbar(), so they can also have shape 2xN for independent specification of lower and upper errors.

Other optional kwargs:
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>facecolor</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[‘/’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestype</td>
<td>[‘solid’</td>
</tr>
<tr>
<td>linewidth</td>
<td>float or None for default</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

**Example:** A stacked bar chart.
Scores by group and gender

Additional kwargs: hold = [True|False] overrides default hold state

```
matplotlib.pyplot.barbs(*args, **kw)
```

Plot a 2-D field of barbs.

Call signatures:

```
barb(U, V, **kw)
barb(U, V, C, **kw)
barb(X, Y, U, V, **kw)
barb(X, Y, U, V, C, **kw)
```

Arguments:

- **X, Y:** The x and y coordinates of the barb locations (default is head of barb; see `pivot` kwarg)
- **U, V:** Give the x and y components of the barb shaft
- **C:** An optional array used to map colors to the barbs

All arguments may be 1-D or 2-D arrays or sequences. If X and Y are absent, they will be generated as a uniform grid. If U and V are 2-D arrays but X and Y are 1-D, and if len(X) and len(Y) match the column and row dimensions of U, then X and Y will be expanded with `numpy.meshgrid()`.

**U, V, C** may be masked arrays, but masked X, Y are not supported at present.

Keyword arguments:
length: Length of the barb in points; the other parts of the barb are scaled against this. Default is 9
pivot: [‘tip’ | ‘middle’] The part of the arrow that is at the grid point; the arrow rotates about this point, hence the name pivot. Default is ‘tip’
barbcolor: [color | color sequence] Specifies the color all parts of the barb except any flags. This parameter is analogous to the edgecolor parameter for polygons, which can be used instead. However this parameter will override facecolor.
flagcolor: [color | color sequence] Specifies 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: 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: A flag on whether the empty barbs (circles) that are drawn should be filled with the flag color. If they are not filled, they will be drawn such that no color is applied to the center. Default is False
rounding: A flag to indicate 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. Default is True
barb_increments: 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: Either a single boolean flag or an array of booleans. Single boolean indicates whether the lines and flags should point opposite to normal for all barbs. An array (which should be the same size as the other data arrays) indicates whether to flip for each individual barb. 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.) Default is False

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 shown schematically below:

```plaintext
: / \ \
: /  \ \\
: /   \ \ \\
: /     \ \\
: ------------------------------
```
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.

 linewidths and edgecolors can be used to customize the barb. Additional PolyCollection keyword arguments:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or antialiaseds</td>
<td>Boolean or sequence of booleans</td>
</tr>
<tr>
<td>array</td>
<td>unknown</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clim</td>
<td>a length 2 sequence of floats</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>cmap</td>
<td>a colormap or registered colormap name</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>colorbar</td>
<td>unknown</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or edgecolors</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>facecolor or facecolors</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
</tbody>
</table>
| hatch                 | [ '/', ' \', '
', '
', ' -', ' +', ' x', ' o', ' O', ' .', ' *' ] |
| label                 | string or anything printable with ‘%s’ conversion. |
| linestyle or linestyles or dashes | ['solid', 'dashed', 'dashdot', 'dotted'] | (offset, on-off-dash-seq) |
| linewidth or lw or linewidths | float or sequence of floats                      |
| lod                   | [True | False]                                      |
| norm                  | unknown                                          |
| offset_position       | unknown                                          |
| offsets               | float or sequence of floats                      |
| path_effects          | unknown                                          |
| paths                 | unknown                                          |
| picker                | [None|float|boolean|callable]                   |
| pickradius            | unknown                                          |
| rasterized            | [True | False | None]                                   |
| sketch_params         | unknown                                          |
| snap                  | unknown                                          |
| transform             | Transform instance                               |
| url                   | a url string                                      |
| urls                  | unknown                                          |
| visible               | [True | False]                                      |
Table 70.4 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

Example:

![Example diagram](image-url)
Additional kwargs: hold = [True|False] overrides default hold state

```python
matplotlib.pyplot.barh(bottom, width, height=0.8, left=None, hold=None, **kwargs)
```

Make a horizontal bar plot.

Call signature:

```python
barh(bottom, width, height=0.8, left=0, **kwargs)
```

Make a horizontal bar plot with rectangles bounded by:

- `left, left + width, bottom, bottom + height` (left, right, bottom and top edges)
- `bottom, width, height`, and `left` can be either scalars or sequences

Return value is a list of `matplotlib.patches.Rectangle` instances.

Required arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bottom</code></td>
<td>the vertical positions of the bottom edges of the bars</td>
</tr>
<tr>
<td><code>width</code></td>
<td>the lengths of the bars</td>
</tr>
</tbody>
</table>

Optional keyword arguments:
### Key-Word | Description
--- | ---
**height** | the heights (thicknesses) of the bars
**left** | the x coordinates of the left edges of the bars
**color** | the colors of the bars
**edge-color** | the colors of the bar edges
**linewidth** | width of bar edges; None means use default linewidth; 0 means don’t draw edges.
**xerr** | if not None, will be used to generate errorbars on the bar chart
**yerr** | if not None, will be used to generate errorbars on the bar chart
**ecolor** | specifies the color of any errorbar
**capsize** | (default 3) determines the length in points of the error bar caps
**align** | ‘edge’ (default) | ‘center’
**log** | [False|True] False (default) leaves the horizontal axis as-is; True sets it to log scale

Setting `align = ‘edge’` aligns bars by their bottom edges in bottom, while `align = ‘center’` interprets these values as the y coordinates of the bar centers.

The optional arguments `color`, `edgecolor`, `linewidth`, `xerr`, and `yerr` can be either scalars or sequences of length equal to the number of bars. This enables you to use barh as the basis for stacked bar charts, or candlestick plots.

Other optional kwargs:
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[ (Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or cc</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>facecolor or fc</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[‘/’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>[‘solid’</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float or None for default</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

Additional kwargs: hold = [True|False] overrides default hold state

`matplotlib.pyplot.bone()`

set the default colormap to bone and apply to current image if any. See help(colormaps) for more information

`matplotlib.pyplot.box(on=None)`

Turn the axes box on or off. on may be a boolean or a string, ‘on’ or ‘off’.

If on is None, toggle state.

`matplotlib.pyplot.boxplot(x, notch=False, sym='b+', vert=True, whis=1.5, positions=None, widths=None, patch_artist=False, bootstrap=None, usermedians=None, conf_intervals=None, hold=None)`

Make a box and whisker plot.

Call signature:

```
boxplot(x, notch=False, sym='+', vert=True, whis=1.5,
```
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.

Function Arguments:

- \( x \): Array or a sequence of vectors.
  - \( notch \): If False (default), produces a rectangular box plot. If True, produces a notched box plot.
  - \( sym \): The default symbol for flier points. Enter an empty string ("") if you don’t want to show fliers.
  - \( vert \): If True (default), makes the boxes vertical. If False, makes horizontal boxes.
  - \( whis \): Defines the length of the whiskers as a function of the inner quartile range. They extend to the most extreme data point within \( \text{whis} \times (75\%-25\%) \) data range.
  - \( bootstrap \): Specifies whether to bootstrap the confidence intervals around the median for notched boxplots. If bootstrap==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.
  - \( usermedians \): An array or sequence whose first dimension (or length) is compatible with \( x \). This overrides the medians computed by matplotlib for each element of \( usermedians \) that is not None. When an element of \( usermedians \) is None, the median will be computed directly as normal.
  - \( conf_intervals \): Array or sequence whose first dimension (or length) is compatible with \( x \) and whose second dimension is 2. When the current element of \( conf_intervals \) is not None, the notch locations computed by matplotlib are overridden (assuming notch is True). When an element of \( conf_intervals \) is None, boxplot compute notches the method specified by the other kwargs (e.g., \( bootstrap \)).
  - \( positions \): Sets the horizontal positions of the boxes. The ticks and limits are automatically set to match the positions.
  - \( widths \): Either a scalar or a vector and sets the width of each box. The default is 0.5, or \( 0.15 \times \text{distance between extreme positions} \) if that is smaller.
  - \( patch_artist \): If False (default), produces boxes with the Line2D artist. If True, produces boxes with the Patch artist.

Returns a dictionary mapping each component of the boxplot to a list of the \texttt{matplotlib.lines.Line2D} instances created. That dictionary has the following keys (assuming vertical boxplots):

- \textit{boxes}: the main body of the boxplot showing the quartiles and the median’s confidence intervals if enabled.
- \textit{medians}: horizontal lines at the median of each box.
- \textit{whiskers}: the vertical lines extending to the most extreme, n-outlier data points.
- **caps**: the horizontal lines at the ends of the whiskers.
- **fliers**: points representing data that extend beyond the whiskers (outliers).

**Example:**

![Box plot example](image-url)
1

-100 -50 0 50 100 150 200
Additional kwargs: hold = [True|False] overrides default hold state

`matplotlib.pyplot.broken_barh(xranges, yrange, hold=None, **kwargs)`
Plot horizontal bars.

Call signature:

`broken_barh(self, xranges, yrange, **kwargs)`

A collection of horizontal bars spanning `yrange` with a sequence of `xranges`.

Required arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>xranges</code></td>
<td>sequence of <code>(xmin, xwidth)</code></td>
</tr>
<tr>
<td><code>yrange</code></td>
<td>sequence of <code>(ymin, ywidth)</code></td>
</tr>
</tbody>
</table>

kwargs are `matplotlib.collections.BrokenBarHCollection` properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>agg_filter</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>alpha</code></td>
<td>float or None</td>
</tr>
<tr>
<td><code>animated</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>antialiased</code></td>
<td>Boolean or sequence of booleans</td>
</tr>
<tr>
<td><code>array</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>axes</code></td>
<td>an <code>Axes</code> instance</td>
</tr>
</tbody>
</table>

Continued on next page
Table 70.5 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clim</td>
<td>a length 2 sequence of floats</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>cmap</td>
<td>a colormap or registered colormap name</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>colorbar</td>
<td>unknown</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or edgecolors</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>facecolor or facecolors</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>['/'</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or linestyles or dashes</td>
<td>['solid'</td>
</tr>
<tr>
<td>linewidth or lw or linewidths</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>norm</td>
<td>unknown</td>
</tr>
<tr>
<td>offset_position</td>
<td>unknown</td>
</tr>
<tr>
<td>offsets</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>paths</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None]float[boolean]callable]</td>
</tr>
<tr>
<td>pickradius</td>
<td>unknown</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>urls</td>
<td>unknown</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

these can either be a single argument, ie:

    facecolors = 'black'

or a sequence of arguments for the various bars, ie:

    facecolors = ('black', 'red', 'green')

Example:
Additional kwargs: hold = [True|False] overrides default hold state

```
matplotlin.pyplot.cla()

Clear the current axes.
```

```
matplotlin.pyplot.clabel(CS, *args, **kwargs)

Label a contour plot.

Call signature:

clabel(cs, **kwargs)
```

Adds labels to line contours in `cs`, where `cs` is a ContourSet object returned by `contour`.

```
clabel(cs, v, **kwargs)
```

only labels contours listed in `v`.

Optional keyword arguments:

- `fontsize`: size in points or relative size eg ‘smaller’, ‘x-large’
- `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 matplotlib color args (string, float, rgb, etc), different labels will be plotted in different colors in the order specified

**inline**: controls whether the underlying contour is removed or not. Default is `True`.

**inline_spacing**: space in pixels to leave on each side of label when placing inline. Defaults to 5. This spacing will be exact for labels at locations where the contour is straight, less so for labels on curved contours.

**fmt**: a format string for the label. Default is `‘%1.3f’` 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**: 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).

 Manuel can be an iterable object of x.y tuples. Contour labels will be created as if mouse is clicked at each x.y positions.

**rightside_up**: if `True` (default), label rotations will always be plus or minus 90 degrees from level.

**use_clabeltext**: if `True` (default is False), `ClabelText` class (instead of `matplotlib.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.
Simplest default with labels
labels at selected locations
Single color - negative contours dashed
Single color - negative contours solid
Crazy lines
Additional kwargs: hold = [True|False] overrides default hold state

matplotlib.pyplot.clf()
Clear the current figure.

matplotlib.pyplot.clim(vmin=None, vmax=None)
Set the color limits of the current image.

To apply clim to all axes images do:

clim(0, 0.5)

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, for example:

for im in gca().get_images():
    im.set_clim(0, 0.05)

matplotlib.pyplot.close(*args)
Close a figure window.

close() by itself closes the current figure

close(h) where h is a Figure instance, closes that figure

close(num) closes figure number num
**close**(<code>name</code>) where <code>name</code> is a string, closes figure with that label

**close('all')** closes all the figure windows

**matplotlib.pyplot.cohere**(<code>x</code>, <code>y</code>, <code>NFFT=256</code>, <code>Fs=2</code>, <code>Fc=0</code>, <code>detrend=<function detrend_none at 0x24627d0</function></code>, <code>window=<function window_hanning at 0x2462578</function></code>, <code>noverlap=0</code>, <code>pad_to=None</code>, <code>sides='default'</code>, <code>scale_by_freq=None</code>, <code>hold=None</code>, **<code>kwargs</code>**)

Plot the coherence between <code>x</code> and <code>y</code>.

Call signature:

```python
cohere(x, y, NFFT=256, Fs=2, Fc=0, detrend = mlab.detrend_none,
       window = mlab.window_hanning, noverlap=0, pad_to=None,
       sides='default', scale_by_freq=None, **kwargs)
```

Plot the coherence between <code>x</code> and <code>y</code>. Coherence is the normalized cross spectral density:

$$C_{xy} = \frac{|P_{xy}|^2}{P_{xx}P_{yy}}$$

(70.1)

Keyword arguments:

**NFFT**: integer The number of data points used in each block for the FFT. Must be even; a power 2 is most efficient. The default value is 256. This should NOT be used to get zero padding, or the scaling of the result will be incorrect. Use pad_to for this instead.

**Fs**: scalar The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, freqs, in cycles per time unit. The default value is 2.

**detrend**: callable 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 pyplot module defines detrend_none(), detrend_mean(), and detrend_linear(), but you can use a custom function as well.

**window**: callable or ndarray A function or a vector of length <code>NFFT</code>. To create window vectors see window_hanning(), window_none(), numpy.blackman(), numpy.hamming(), numpy.bartlett(), scipy.signal(), scipy.signal.get_window(), etc. The default is window_hanning(). 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.

**pad_to**: integer The number of points to which the data segment is padded when performing the FFT. This can be different from <code>NFFT</code>, which specifies the number of data points used. While not increasing the actual resolution of the psd (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the <code>n</code> parameter in the call to fft(). The default is None, which sets pad_to equal to <code>NFFT</code>

**sides**: ['default' | 'onesided' | 'twosided'] Specifies which sides of the PSD to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. ‘onesided’ forces the return of a one-sided PSD, while ‘twosided’ forces two-sided.

**scale_by_freq**: boolean Specifies 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:** integer  The number of points of overlap between blocks. The default value is 0 (no overlap).

**Fc:** integer  The center frequency of \( x \) (defaults to 0), 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.

The return value is a tuple \((C_{xy}, f)\), where \( f \) are the frequencies of the coherence vector.

kwargs are applied to the lines.

References:


kwargs control the **Line2D** properties of the coherence plot:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>data</td>
<td>2D array (rows are x, y) or two 1D arrays</td>
</tr>
<tr>
<td>drawstyle</td>
<td>['default'</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>['full'</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['- '</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>marker</td>
<td>unknown</td>
</tr>
<tr>
<td>markeredgecolor or mec</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markeredgewidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
</tbody>
</table>
Table 70.6 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function ( \text{fn(artist, event)} )</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>[‘butt’</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>[‘miter’</td>
</tr>
<tr>
<td>transform</td>
<td>a \text{matplotlib.transforms.Transform} instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

Example:

![Plot](image_url)

Additional kwargs: hold = [True|False] overrides default hold state

\text{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:

```python
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(s) 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` False | 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 grid_spec module.

Additional keyword arguments are of two kinds:

- **axes properties**:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>orientation</td>
<td>vertical or horizontal</td>
</tr>
<tr>
<td>fraction</td>
<td>0.15; fraction of original axes to use for colorbar</td>
</tr>
<tr>
<td>pad</td>
<td>0.05 if vertical, 0.15 if horizontal; fraction of original axes between colorbar and new image axes</td>
</tr>
<tr>
<td>shrink</td>
<td>1.0; fraction by which to shrink the colorbar</td>
</tr>
<tr>
<td>aspect</td>
<td>20; ratio of long to short dimensions</td>
</tr>
<tr>
<td>anchor</td>
<td>(0.0, 0.5) if vertical; (0.5, 1.0) if horizontal; the anchor point of the colorbar axes</td>
</tr>
<tr>
<td>panchor</td>
<td>(1.0, 0.5) if vertical; (0.5, 0.0) if horizontal; the anchor point of the colorbar parent axes. If False, the parent axes’ anchor will be unchanged</td>
</tr>
</tbody>
</table>

- **colorbar properties**:
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>extend</td>
<td>[ 'neither'</td>
</tr>
<tr>
<td>extendfrac</td>
<td>[ None</td>
</tr>
<tr>
<td>extendrect</td>
<td>[ False</td>
</tr>
<tr>
<td>spacing</td>
<td>[ 'uniform'</td>
</tr>
<tr>
<td>ticks</td>
<td>[ None</td>
</tr>
<tr>
<td>format</td>
<td>[ None</td>
</tr>
<tr>
<td>drawedges</td>
<td>False</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boundaries</td>
<td>None or a sequence</td>
</tr>
<tr>
<td>values</td>
<td>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.</td>
</tr>
</tbody>
</table>

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.

```
matplotlib.pyplot.colors()
```

This is a do-nothing function to provide you with help on how matplotlib handles colors.

Commands which take color arguments can use several formats to specify the colors. For the basic built-in colors, you can use a single letter

<table>
<thead>
<tr>
<th>Alias</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>'b'</td>
<td>blue</td>
</tr>
<tr>
<td>'g'</td>
<td>green</td>
</tr>
<tr>
<td>'r'</td>
<td>red</td>
</tr>
<tr>
<td>'c'</td>
<td>cyan</td>
</tr>
<tr>
<td>'m'</td>
<td>magenta</td>
</tr>
<tr>
<td>'y'</td>
<td>yellow</td>
</tr>
<tr>
<td>'k'</td>
<td>black</td>
</tr>
<tr>
<td>'w'</td>
<td>white</td>
</tr>
</tbody>
</table>

For a greater range of colors, you have two options. You can specify the color using an html hex string, as in:

```
color = '#eefeff'
```

or you can pass an R,G,B tuple, where each of R,G,B are in the range [0,1].

You can also use any legal html name for a color, for example:

```
color = 'red'
color = 'burlywood'
color = 'chartreuse'
```

The example below creates a subplot with a dark slate gray background:

```
subplot((111, axisbg=(0.1843, 0.3098, 0.3098))
```

Here is an example that creates a pale turquoise title:

```
title('Is this the best color?', color='#afefee')
```
```python
matplotlib.pyplot.connect(s, func)

Connect event with string s to func. The signature of func is:

def func(event)

where event is a matplotlib.backend_bases.Event. The following events are recognized

- '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'

For the location events (button and key press/release), if the mouse is over the axes, the variable event.inaxes will be set to the Axes the event occurs is over, and additionally, the variables event.xdata and event.ydata will be defined. This is the mouse location in data coords. See KeyEvent and MouseEvent for more info.

Return value is a connection id that can be used with mpl_disconnect().

Example usage:

def on_press(event):
    print('you pressed', event.button, event.xdata, event.ydata)

cid = canvas.mpl_connect('button_press_event', on_press)
```

```python
matplotlib.pyplot.contour(*args, **kwargs)

Plot contours.

contour() and contourf() draw contour lines and filled contours, respectively. Except as noted, function signatures and return values are the same for both versions.

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().

Call signatures:

contour(Z)

make a contour plot of an array Z. The level values are chosen automatically.

contour(X, Y, Z)
```
$X$, $Y$ specify the (x, y) coordinates of the surface

```
contour(Z,N)
contour(X,Y,Z,N)
```

contour $N$ automatically-chosen levels.

```
contour(Z,V)
contour(X,Y,Z,V)
```

draw contour lines at the values specified in sequence $V$

```
contourf(..., V)
```

fill the $\text{len}(V)-1$ regions between the values in $V$

```
contour(Z, **kwargs)
```

Use keyword args to control colors, linewidth, origin, cmap ... see below for more details.

$X$ and $Y$ must both be 2-D with the same shape as $Z$, or they must both be 1-D such that $\text{len}(X)$ is the number of columns in $Z$ and $\text{len}(Y)$ is the number of rows in $Z$.

$C = \text{contour}(...) \text{ returns a QuadContourSet object.}$

Optional keyword arguments:

- `colors`: [ `None` | string | (mpl_colors) ] If `None`, the colormap specified by cmap will be used.
  
  - If a string, like ‘r’ or ‘red’, all levels will be plotted in this color.
  
  - If a tuple of matplotlib color args (string, float, rgb, etc), different levels will be plotted in different colors in the order specified.

- `alpha`: float The alpha blending value

- `cmap`: [ `None` | Colormap ] A cm Colormap instance or `None`. If cmap is `None` and colors is `None`, a default Colormap is used.

- `norm`: [ `None` | Normalize ] A matplotlib.colors.Normalize instance for scaling data values to colors. If norm is `None` and colors is `None`, the default linear scaling is used.

- `vmin, vmax`: [ `None` | scalar ] If not `None`, either or both of these values will be supplied to the matplotlib.colors.Normalize instance, overriding the default color scaling based on levels.

- `levels`: [level0, level1, ..., leveln] A list of floating point numbers indicating the level curves to draw; eg to draw just the zero contour pass levels=[0]

- `origin`: [ `None` | ‘upper’ | ‘lower’ | ‘image’ ] If None, the first value of Z will correspond to the lower left corner, location (0,0). If ‘image’, the rc value for image.origin will be used.
  
  This keyword is not active if $X$ and $Y$ are specified in the call to contour.

- `extent`: [ `None` | (x0,x1,y0,y1) ]
If `origin` is not `None`, then `extent` is interpreted as in `matplotlib.pyplot.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 keyword is not active if $X$ and $Y$ are specified in the call to `contour`.

**locator**: `[None | ticker.Locator subclass]` If `locator` is `None`, the default `MaxNLocator` is used. The locator is used to determine the contour levels if they are not given explicitly via the `V` argument.

**extend**: `[‘neither’ | ‘both’ | ‘min’ | ‘max’]` Unless this is `‘neither’`, contour levels are automatically added to one or both ends of the range so that all data are included. These added ranges are then mapped to the special colormap values which default to the ends of the colormap range, but can be set via `matplotlib.colors.Colormap.set_under()` and `matplotlib.colors.Colormap.set_over()` methods.

**xunits, yunits**: `[None | registered units]` Override axis units by specifying an instance of a `matplotlib.units.ConversionInterface`.

**antialiased**: `[True | False]` enable antialiasing, overriding the defaults. For filled contours, the default is `True`. For line contours, it is taken from `rcParams[‘lines.antialiased’]`.

**contour-only keyword arguments**:

**linewidths**: `[None | number | tuple of numbers]` If `linewidths` is `None`, the default width in `lines.linewidth` in `matplotlibrc` is used.

If a number, all levels will be plotted with this linewidth.

If a tuple, different levels will be plotted with different linewidths in the order specified.

**linestyles**: `[None | ‘solid’ | ‘dashed’ | ‘dashdot’ | ‘dotted’]` If `linestyles` is `None`, the default is ‘solid’ unless the lines are monochrome. In that case, negative contours will take their linestyle from the `matplotlibrc contour.negative_linestyle` 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.

**contourf-only keyword arguments**:

**nchunk**: `[0 | integer]` If 0, no subdivision of the domain. Specify a positive integer to divide the domain into subdomains of roughly `nchunk` by `nchunk` points. This may never actually be advantageous, so this option may be removed. Chunking introduces artifacts at the chunk boundaries unless `antialiased` is `False`.

**hatches**: 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.

Note: contourf fills intervals that are closed at the top; that is, for boundaries $z1$ and $z2$, the filled region is:
$z_1 < z \leq z_2$

There is one exception: if the lowest boundary coincides with the minimum value of the $z$ array, then that minimum value will be included in the lowest interval.

Examples:
labels at selected locations
Single color - negative contours dashed
Single color - negative contours solid
Crazy lines
Nonsense (3 masked regions)

![Diagram showing the relationship between word length anomaly and sentence length anomaly with a color scale for verbosity coefficient.](image-url)
Matplotlib, Release 1.3.0

Additional kwargs: hold = [True|False] overrides default hold state

```
import matplotlib.pyplot as plt
plt.contourf(*args, **kwargs)
```

Plot contours.

- `contour()` and `contourf()` draw contour lines and filled contours, respectively. Except as noted, function signatures and return values are the same for both versions.

- `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()`

Call signatures:

- `contour(Z)`

  make a contour plot of an array `Z`. The level values are chosen automatically.

- `contour(X, Y, Z)`

  `X, Y` specify the (x, y) coordinates of the surface

- `contour(Z, N)`

- `contour(X, Y, Z, N)`

  contour `N` automatically-chosen levels.
contour(Z, V)
contour(X, Y, Z, V)

draw contour lines at the values specified in sequence V

contourf(..., V)

fill the \text{len(V)}-1 regions between the values in V

contour(Z, **kwargs)

Use keyword args to control colors, linewidth, origin, cmap ... see below for more details.

\(X\) and \(Y\) must both be 2-D with the same shape as \(Z\), or they must both be 1-D such that \text{len}(X) is the number of columns in \(Z\) and \text{len}(Y) is the number of rows in \(Z\).

\(C = \text{contour}(\ldots)\) returns a QuadContourSet object.

Optional keyword arguments:

\textit{colors}: \([\text{None} | \text{string} | (\text{mpl\_colors})]\) If \text{None}, the colormap specified by \text{cmap} will be used.

If a string, like ‘r’ or ‘red’, all levels will be plotted in this color.

If a tuple of matplotlib color args (string, float, rgb, etc), different levels will be plotted in different colors in the order specified.

\textit{alpha}: \text{float} The alpha blending value

\textit{cmap}: \([\text{None} | \text{Colormap}]\) A \text{cm Colormap} instance or \text{None}. If \text{cmap} is \text{None} and \text{colors is None}, a default Colormap is used.

\textit{norm}: \([\text{None} | \text{Normalize}]\) A \text{matplotlib.colors.Normalize} instance for scaling data values to colors. If \text{norm is None} and \text{colors is None}, the default linear scaling is used.

\textit{vmin, vmax}: \([\text{None} | \text{scalar}]\) If not \text{None}, either or both of these values will be supplied to the \text{matplotlib.colors.Normalize} instance, overriding the default color scaling based on \text{levels}.

\textit{levels}: \([\text{level0, level1, ...}, \text{leveln}]\) A list of floating point numbers indicating the level curves to draw; eg to draw just the zero contour pass \text{levels=\[0\]}

\textit{origin}: \([\text{None} | \text{‘upper’ | ‘lower’ | ‘image’}]\) If \text{None}, the first value of \(Z\) will correspond to the lower left corner, location (0,0). If ‘image’, the rc value for \text{image.origin} will be used.

This keyword is not active if \(X\) and \(Y\) are specified in the call to \text{contour}.

\textit{extent}: \([\text{None} | \text{(x0,x1,y0,y1)}]\)

If \text{origin} is not \text{None}, then \text{extent} is interpreted as in \text{matplotlib.pyplot.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 \text{origin is None}, then \((x0, y0)\) is the position of \(Z[0,0]\), and \((xl, yl)\) is the position of \(Z[-1,-1]\).

This keyword is not active if \(X\) and \(Y\) are specified in the call to \text{contour}. 
locator: [ None | ticker.Locator subclass ] If locator is None, the default MaxNLocator is used. The locator is used to determine the contour levels if they are not given explicitly via the V argument.

extend: [ ‘neither’ | ‘both’ | ‘min’ | ‘max’ ] Unless this is ‘neither’, contour levels are automatically added to one or both ends of the range so that all data are included. These added ranges are then mapped to the special colormap values which default to the ends of the colormap range, but can be set via matplotlib.colors.Colormap.set_under() and matplotlib.colors.Colormap.set_over() methods.

xunits, yunits: [ None | registered units ] Override axis units by specifying an instance of a matplotlib.units.ConversionInterface.

antialiased: [ True | False ] enable antialiasing, overriding the defaults. For filled contours, the default is True. For line contours, it is taken from rc-Params[‘lines.antialiased’).

countour-only keyword arguments:

linewidens: [ None | number | tuple of numbers ] If linewidens is None, the default width in lines.linewidth in matplotlibrc is used.

If a number, all levels will be plotted with this linewidth.

If a tuple, different levels will be plotted with different linewidens in the order specified

linestyles: [ None | ‘solid’ | ‘dashed’ | ‘dashdot’ | ‘dotted’ ] If linestyles is None, the default is ‘solid’ unless the lines are monochrome. In that case, negative contours will take their linestyle from the matplotlibrc contour.negative_linestyle 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.

countourf-only keyword arguments:

nchunk: [ 0 | integer ] If 0, no subdivision of the domain. Specify a positive integer to divide the domain into subdomains of roughly nchunk by nchunk points. This may never actually be advantageous, so this option may be removed. Chunking introduces artifacts at the chunk boundaries unless antialiased is False.

hatches: 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.

Note: contourf fills intervals that are closed at the top; that is, for boundaries z1 and z2, the filled region is:

z1 < z <= z2

There is one exception: if the lowest boundary coincides with the minimum value of the z array, then that minimum value will be included in the lowest interval.

Examples:
Simplest default with labels
labels at selected locations
Single color - negative contours dashed
Single color - negative contours solid
Nonsense (3 masked regions)

Verbosity coefficient

Word length anomaly

Sentence length anomaly
Additional kwargs: hold = [True|False] overrides default hold state

**matplotlib.pyplot.cool()**

set the default colormap to cool and apply to current image if any. See help(colormaps) for more information

**matplotlib.pyplot.copper()**

set the default colormap to copper and apply to current image if any. See help(colormaps) for more information

**matplotlib.pyplot.csd**(x, y, NFFT=256, Fs=2, Fc=0, detrend=\texttt{<function detrend\_none at 0x24627d0>}, window=\texttt{<function window\_hanning at 0x2462578>}, noverlap=0, pad_to=None, sides='default', scale\_by\_freq=None, hold=None, **kwargs)

Plot cross-spectral density.

Call signature:

csd(x, y, NFFT=256, Fs=2, Fc=0, detrend=\texttt{mlab.detrend\_none}, window=\texttt{mlab.window\_hanning}, noverlap=0, pad_to=None, sides='default', scale\_by\_freq=None, hold=None, **kwargs)

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 $\text{detrend}$ and windowed by function $\text{window}$. 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.
Returns the tuple \((P_{xy}, \text{freqs})\). \(P\) is the cross spectrum (complex valued), and \(10 \log_{10} |P_{xy}|\) is plotted.

Keyword arguments:

**NFFT:** integer The number of data points used in each block for the FFT. Must be even; a power 2 is most efficient. The default value is 256. This should NOT be used to get zero padding, or the scaling of the result will be incorrect. Use `pad_to` for this instead.

**Fs:** scalar The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, `freqs`, in cycles per time unit. The default value is 2.

**detrend:** callable 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 it is a function. The `pylab` module defines `detrend_none()`, `detrend_mean()`, and `detrend_linear()`, but you can use a custom function as well.

**window:** callable or ndarray A function or a vector of length `NFFT`. To create window vectors see `window_hanning()`, `window_none()`, `numpy.blackman()`, `numpy.hanning()`, `numpy.hamming()`, `numpy.bartlett()`, `scipy.signal()`, `scipy.signal.get_window()`, etc. The default is `window_hanning()`. 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.

**pad_to:** integer 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 psd (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`.

**sides:** [‘default’ | ‘onesided’ | ‘twosided’] Specifies which sides of the PSD to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. ‘onesided’ forces the return of a one-sided PSD, while ‘twosided’ forces two-sided.

**scale_by_freq:** boolean Specifies 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:** integer The number of points of overlap between blocks. The default value is 0 (no overlap).

**Fc:** integer The center frequency of \(x\) (defaults to 0), 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.


kwargs control the Line2D properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>agg_filter</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>alpha</code></td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td><code>animated</code></td>
<td>[True</td>
</tr>
</tbody>
</table>
Table 70.7 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>data</td>
<td>2D array (rows are x, y) or two 1D arrays</td>
</tr>
<tr>
<td>drawstyle</td>
<td>['default'</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>['full'</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with '%%s' conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['-'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>marker</td>
<td>unknown</td>
</tr>
<tr>
<td>markeredgewidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

Example:
Additional kwargs: hold = [True|False] overrides default hold state

```
matplotlib.pyplot.delaxes(*args)
```

Remove an axes from the current figure. If ax doesn’t exist, an error will be raised.

delaxes() : delete the current axes

```
matplotlib.pyplot.disconnect(cid)
```

Disconnect callback id cid

Example usage:

```
cid = canvas.mpl_connect('button_press_event', on_press)
    #...later
    canvas.mpl_disconnect(cid)
```

```
matplotlib.pyplot.draw()
```

Redraw the current figure.

This is used in interactive mode to update a figure that has been altered using one or more plot object method calls; it is not needed if figure modification is done entirely with pyplot functions, if a sequence of modifications ends with a pyplot function, or if matplotlib is in non-interactive mode and the sequence of modifications ends with show() or savefig().

A more object-oriented alternative, given any Figure instance, fig, that was created using a pyplot function, is:
fig.canvas.draw()

matplotlib.pyplot.errorbar(x, y, yerr=None, xerr=None, fmt='-', ecolor=None, elinewidth=None, capsize=3, barsabove=False, lolims=False, uplims=False, xlolims=False, xuplims=False, errorevery=1, capthick=None, hold=None, **kwargs)

Plot an errorbar graph.

Call signature:

```python
errorbar(x, y, yerr=None, xerr=None, fmt='-', ecolor=None, elinewidth=None, capsize=3, barsabove=False, lolims=False, uplims=False, xlolims=False, xuplims=False, errorevery=1, capthick=None, hold=None, **kwargs)
```

Plot \( x \) versus \( y \) with error deltas in \( yerr \) and \( xerr \). Vertical errorbars are plotted if \( yerr \) is not \( None \). Horizontal errorbars are plotted if \( xerr \) is not \( None \).

\( x, y, xerr, \) and \( yerr \) can all be scalars, which plots a single error bar at \( x, y \).

Optional keyword arguments:

- **xerr/yerr:** [ scalar | N, Nx1, or 2xN array-like ] If a scalar number, len(N) array-like object, or an Nx1 array-like object, errorbars are drawn at +/-value relative to the data.

  If a sequence of shape 2xN, errorbars are drawn at -row1 and +row2 relative to the data.

- **fmt:** `-` The plot format symbol. If \( fmt \) is \( None \), only the errorbars are plotted. This is used for adding errorbars to a bar plot, for example.

- **ecolor:** [ None | mpl color ] A matplotlib color arg which gives the color the errorbar lines; if \( None \), use the marker color.

- **elinewidth:** scalar The linewidth of the errorbar lines. If \( None \), use the linewidth.

- **capsize:** scalar The length of the error bar caps in points

- **capthick:** scalar An alias kwarg to `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:** [ True | False ] if True, will plot the errorbars above the plot symbols. Default is below.

- **lolims / uplims / xlolims / xuplims:** [ False | True ] 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 of the same type as \( xerr \) and \( yerr \).

- **errorevery:** positive integer subsamples the errorbars. e.g., if everyerror=5, errorbars for every 5-th datapoint will be plotted. The data plot itself still shows all data points.

All other keyword arguments are passed on to the plot command for 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

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased (or aa)</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color (or c)</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>data</td>
<td>2D array (rows are x, y) or two 1D arrays</td>
</tr>
<tr>
<td>drawstyle</td>
<td>['default'</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.pyplot.figure.Figure instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>['full'</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with '%s' conversion.</td>
</tr>
<tr>
<td>linestyle (or ls)</td>
<td>['-.'</td>
</tr>
<tr>
<td>linewidth (or lw)</td>
<td>float value in points</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>marker</td>
<td>unknown</td>
</tr>
<tr>
<td>markeredgewidth (or mew)</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor (or mec)</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markerfacecoloralt (or mfcalt)</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize (or ms)</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt'</td>
</tr>
</tbody>
</table>
Table 70.8 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>solid_joinstyle</td>
<td>['miter', 'round', 'bevel']</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

Returns (plotline, caplines, barlinecols):

- **plotline**: Line2D instance x, y plot markers and/or line
- **caplines**: list of error bar cap Line2D instances
- **barlinecols**: list of LineCollection instances for the horizontal and vertical error ranges.

Example:

Additional kwargs: hold = [True|False] overrides default hold state

```
matplotlib.pyplot.eventplot(positions, orientation='horizontal', lineoffsets=1, line-lengths=1, linewidths=None, colors=None, linestyles='solid', hold=None, **kwargs)
```

Plot identical parallel lines at specific positions.
Call signature:

```python
eventplot(positions, orientation='horizontal', lineoffsets=0,
          linelengths=1, linewidths=None, color=None,
          linestyles='solid')
```

Plot parallel lines at the given positions. positions should be a 1D or 2D array-like object, with each row corresponding to a row or column of lines.

This type of plot is commonly used in neuroscience for representing neural events, where it is commonly 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.

- **orientation**
  - 'horizontal': the lines will be vertical and arranged in rows
  - 'vertical': lines will be horizontal and arranged in columns

- **lineoffsets**: A float or array-like containing floats.
- **linelengths**: A float or array-like containing floats.
- **linewidths**: A float or array-like containing floats.
- **colors** must be a sequence of RGBA tuples (eg arbitrary color strings, etc, not allowed) or a list of such sequences
- **linestyles**: [‘solid’ | ‘dashed’ | ‘dashdot’ | ‘dotted’] or an array of these values

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 or column in positions.

Returns a list of `matplotlib.collections.EventCollection` objects that were added.

kwars are `LineCollection` properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or antialiaseds</td>
<td>Boolean or sequence of booleans</td>
</tr>
<tr>
<td>array</td>
<td>unknown</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clim</td>
<td>a length 2 sequence of floats</td>
</tr>
<tr>
<td>clip_box</td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>cmap</td>
<td>a colormap or registered colormap name</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>colorbar</td>
<td>unknown</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or edgecolors</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>facecolor or facecolors</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>figure</td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
</tbody>
</table>

Continued on next page
Table 70.9 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[ ‘/’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or linestyles or dashes</td>
<td>[‘solid’</td>
</tr>
<tr>
<td>linewidth or lw or linwidths</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>norm</td>
<td>unknown</td>
</tr>
<tr>
<td>offset_position</td>
<td>unknown</td>
</tr>
<tr>
<td>offsets</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>paths</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>pickradius</td>
<td>unknown</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>segments</td>
<td>unknown</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>urls</td>
<td>unknown</td>
</tr>
<tr>
<td>verts</td>
<td>unknown</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

Example:
Additional kwargs: hold = [True|False] overrides default hold state

matplotlib.pyplot.figimage(*args, **kwargs)

Adds a non-resampled image to the figure.

Call signatures:

figimage(X, **kwargs)

adds a non-resampled array X to the figure.

figimage(X, xo, yo)

with pixel offsets xo, yo,

X must be a float array:

• If X is MxN, assume luminance (grayscale)
• If X is MxN3, assume RGB
• If X is MxN4, assume RGBA

Optional keyword arguments:
<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xo or yo</td>
<td>An integer, the x and y image offset in pixels</td>
</tr>
<tr>
<td>cmap</td>
<td>a <code>matplotlib.colors.Colormap</code> instance, eg cm.jet. If <code>None</code>, default to the rc image cmap value</td>
</tr>
<tr>
<td>norm</td>
<td>a <code>matplotlib.colors.Normalize</code> instance. The default is normalization(). This scales luminance -&gt; 0-1</td>
</tr>
<tr>
<td>vmin</td>
<td><code>None</code> used to scale a luminance image to 0-1. If either is <code>None</code>, the min and max of the luminance values will be used. Note if you pass a norm instance, the settings for vmin and vmax will be ignored.</td>
</tr>
<tr>
<td>alpha</td>
<td>the alpha blending value, default is <code>None</code></td>
</tr>
<tr>
<td>origin</td>
<td>[‘upper’</td>
</tr>
</tbody>
</table>

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 size [0,1,0,1]. An `matplotlib.image.FigureImage` instance is returned.

Additional kwargs are Artist kwargs passed on to :class:`matplotlib.image.FigureImage` Addition kwags: hold = [True|False] overrides default hold state

`matplotlib.pyplot.figlegend(handles, labels, loc, **kwargs)`
Place a legend in the figure.

-labels- a sequence of strings
- handles - a sequence of Line2D or Patch instances
- loc - can be a string or an integer specifying the legend location

A matplotlib.legend.Legend instance is returned.

Example:

```python
figlegend( (line1, line2, line3),
    ('label1', 'label2', 'label3'),
    'upper right' )
```

See also:

legend()

matplotlib.pyplot.figtext(*args, **kwargs)

Add text to figure.

Call signature:

```python
text(x, y, s, fontdict=None, **kwargs)
```

Add text to figure at location x, y (relative 0-1 coords). See text() for the meaning of the other arguments.

kwargs control the Text properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>backgroundcolor</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>bbox</td>
<td>rectangle prop dict</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>family or fontfamily or fontname or name</td>
<td>[FONTNAME</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fontproperties or font_properties</td>
<td>a matplotlib.font_manager.FontProperties instance</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>horizontalalignment or ha</td>
<td>[‘center’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linespacing</td>
<td>float (multiple of font size)</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>multialignment</td>
<td>[‘left’</td>
</tr>
</tbody>
</table>
Table 70.10 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>position</td>
<td>(x,y)</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>rotation</td>
<td>[ angle in degrees</td>
</tr>
<tr>
<td>rotation_mode</td>
<td>unknown</td>
</tr>
<tr>
<td>size or fontsize</td>
<td>[size in points</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>stretch or fontstretch</td>
<td>[a numeric value in range 0-1000</td>
</tr>
<tr>
<td>style or fontstyle</td>
<td>[ ‘normal’</td>
</tr>
<tr>
<td>text</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>variant or fontvariant</td>
<td>[ ‘normal’</td>
</tr>
<tr>
<td>verticalalignment or va or ma</td>
<td>[ ‘center’</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>x</td>
<td>float</td>
</tr>
<tr>
<td>y</td>
<td>float</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

```
matplotlib.pyplot.figure(num=None, figsize=None, dpi=None, facecolor=None, edgecolor=None, frameon=True, FigureClass=<class 'matplotlib.figure.Figure'>, **kwargs)
```

Creates a new figure.

**Parameters**
- **num** : integer or string, optional, default: none
  If not provided, a new figure will be created, and a the figure number will be incremented. The figure objects holds this number in a number attribute. If num is provided, and a figure with this id already exists, make it active, and returns a reference to it. If this figure does not exists, create it and returns it. If num is a string, the window title will be set to this figure’s num.
- **figsize** : tuple of integers, optional, default
  width, height in inches. If not provided, defaults to rc figure.figsize.
- **dpi** : integer, optional, default ; None
  resolution of the figure. If not provided, defaults to rc figure.dpi.
- **facecolor** : 
  the background color; If not provided, defaults to rc figure.facecolor
- **edgecolor** : 
  the border color. If not provided, defaults to rc figure.edgecolor

**Returns**
- **figure** : 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 pylab interface. Additional kwargs will be passed to the figure init function.

```python
matplotlib.pyplot.fill(*args, **kwargs)
```

Plot filled polygons.

Call signature:

```python
fill(*args, **kwargs)
```

`args` is a variable length argument, allowing for multiple x, y pairs with an optional color format string; see `plot()` for details on the argument parsing. For example, to plot a polygon with vertices at x, y in blue:

```python
ax.fill(x, y, 'b')
```

An arbitrary number of x, y, color groups can be specified:

```python
ax.fill(x1, y1, 'g', x2, y2, 'r')
```

Return value is a list of `Patch` instances that were added.

The same color strings that `plot()` supports are supported by the fill format string.

If you would like to fill below a curve, e.g., shade a region between 0 and y along x, use `fill_between()`

The `closed` kwarg will close the polygon when `True` (default).

kwargs control the `Polygon` properties:
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or ec</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>facecolor or fc</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[‘/’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>[‘solid’</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float or None</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

Example:
Additional kwargs: hold = [True|False] overrides default hold state

`matplotlib.pyplot.fill_between(x, y1, y2=0, where=None, interpolate=False, hold=None, **kwargs)`

Make filled polygons between two curves.

Call signature:

```
fill_between(x, y1, y2=0, where=None, **kwargs)
```

Create a `PolyCollection` filling the regions between `y1` and `y2` where `where==True`

- `x`: An N-length array of the x data
- `y1`: An N-length array (or scalar) of the y data
- `y2`: An N-length array (or scalar) of the y data
- `where`: If `None`, default to fill between everywhere. If not `None`, it is an N-length numpy boolean array and the fill will only happen over the regions where `where==True`.
- `interpolate`: If `True`, interpolate between the two lines to find the precise point of intersection. Otherwise, the start and end points of the filled region will only occur on explicit values in the `x` array.
- `kwargs`: Keyword args passed on to the `PolyCollection`. `kwargs` control the `Polygon` properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>agg_filter</code></td>
<td>unknown</td>
</tr>
</tbody>
</table>

Continued on next page
Table 70.11 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or anti_aliaseds</td>
<td>Boolean or sequence of booleans</td>
</tr>
<tr>
<td>array</td>
<td>unknown</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clim</td>
<td>a length 2 sequence of floats</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>cmap</td>
<td>a colormap or registered colormap name</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>colorbar</td>
<td>unknown</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or edgecolors</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>facecolor or facecolors</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[''/'</td>
</tr>
<tr>
<td>linestyle or linestyles or dashes</td>
<td>['solid'</td>
</tr>
<tr>
<td>linewidth or lw or linewidths</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>norm</td>
<td>unknown</td>
</tr>
<tr>
<td>offset_position</td>
<td>unknown</td>
</tr>
<tr>
<td>offsets</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>paths</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None]float</td>
</tr>
<tr>
<td>pickradius</td>
<td>unknown</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>urls</td>
<td>unknown</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>
between y1 and 0
0.0 0.5 1.0 1.5 2.0
x
1.5
1.0
0.5
0.0
0.5
1.0
1.5
between y1 and y2
1.0
0.5
0.0
0.5
1.0
between y1 and 0
1.0
0.5
0.0
fill between where

Now regions with y2>1 are masked
See also:

`fill_betweenx()` for filling between two sets of x-values

Additional kwargs: `hold = [True|False]` overrides default hold state

```python
matplotlib.pyplot.fill_betweenx(y, x1, x2=0, where=None, hold=None, **kwargs)
```

Make filled polygons between two horizontal curves.

Call signature:

```python
fill_betweenx(y, x1, x2=0, where=None, **kwargs)
```

Create a `PolyCollection` filling the regions between `x1` and `x2` where `where==True`

- `y`: An N-length array of the y data
- `x1`: An N-length array (or scalar) of the x data
- `x2`: An N-length array (or scalar) of the x data
- `where`: If `None`, default to fill between everywhere. If not `None`, it is a N length numpy boolean array and the fill will only happen over the regions where `where==True`
- `**kwargs`: keyword args passed on to the `PolyCollection`

`kwargs` control the `Polygon` properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>agg_filter</code></td>
<td>unknown</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or antialiaseds</td>
<td>Boolean or sequence of booleans</td>
</tr>
<tr>
<td>array</td>
<td>unknown</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clim</td>
<td>a length 2 sequence of floats</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>cmap</td>
<td>a colormap or registered colormap name</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>colorbar</td>
<td>unknown</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgcolor or edgecolors</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>facecolor or facecolors</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>['/'</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or linestyles or dashes</td>
<td>['solid'</td>
</tr>
<tr>
<td>linewidth or lw or linewidths</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>norm</td>
<td>unknown</td>
</tr>
<tr>
<td>offset_position</td>
<td>unknown</td>
</tr>
<tr>
<td>offsets</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>paths</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>pickradius</td>
<td>unknown</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>urls</td>
<td>unknown</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>
between y1 and 0
between y1 and 1
between y1 and y2
See also:

`fill_between()` for filling between two sets of y-values

Additional kwargs: `hold = [True|False]` overrides default hold state

`matplotlib.pyplot.findobj(o=None, match=None, include_self=True)`

Find artist objects.

Recursively find all `Artist` instances contained in self.

`match` can be

- `None`: return all objects contained in artist.
- `function with signature boolean = match(artist)` used to filter matches
- `class instance`: e.g., `Line2D`. Only return artists of class type.

If `include_self` is True (default), include self in the list to be checked for a match.

`matplotlib.pyplot.flag()`

Set the default colormap to flag and apply to current image if any. See help(colormaps) for more information.

`matplotlib.pyplot.gca(**kwargs)`

Return the current axis instance. This can be used to control axis properties either using set or the `Axes` methods, for example, setting the xaxis range:
```python
plot(t,s)
set(gca(), 'xlim', [0,10])

or:

plot(t,s)
a = gca()
a.set_xlim([0,10])
```

```python
matplotlib.pyplot.gcf()
Return a reference to the current figure.
```

```python
matplotlib.pyplot.gci()
Get the current colorable artist. Specifically, returns the current ScalarMappable instance (image or patch collection), or None if no images or patch collections have been defined. The commands imshow() and figimage() create Image instances, and the commands pcolor() and scatter() create Collection instances. The current image is an attribute of the current axes, or the nearest earlier axes in the current figure that contains an image.
```

```python
matplotlib.pyplot.get_current_fig_manager()
```

```python
matplotlib.pyplot.get_figlabels()
Return a list of existing figure labels.
```

```python
matplotlib.pyplot.get_fignums()
Return a list of existing figure numbers.
```

```python
matplotlib.pyplot.get_plot_commands()
Get a sorted list of all of the plotting commands.
```

```python
matplotlib.pyplot.ginput(*args, **kwargs)
Call signature:

    ginput(self, n=1, timeout=30, show_clicks=True,
           mouse_add=1, mouse_pop=3, mouse_stop=2)

Blocking call to interact with the figure.
This will wait for n clicks from the user and return a list of the coordinates of each click.
If timeout is zero or negative, does not timeout.
If n is zero or negative, accumulate clicks until a middle click (or potentially both mouse buttons at once) terminates the input.
Right clicking cancels last input.
The buttons used for the various actions (adding points, removing points, terminating the inputs) can be overridden via the arguments mouse_add, mouse_pop and mouse_stop, that give the associated mouse button: 1 for left, 2 for middle, 3 for right.
```
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.

**matplotlib.pyplot.gray()**

set the default colormap to gray and apply to current image if any. See help(colormaps) for more information

**matplotlib.pyplot.grid(b=None, which='major', axis='both', **kwargs)**

Turn the axes grids on or off.

Call signature:

```
grid(self, b=None, which='major', axis='both', **kwargs)
```

Set the axes grids on or off; `b` is a boolean. (For MATLAB compatibility, `b` may also be a string, ‘on’ or ‘off’.)

If `b` is `None` and `len(kwargs)==0`, toggle the grid state. If `kwargs` are supplied, it is assumed that you want a grid and `b` is thus set to `True`.

`which` can be ‘major’ (default), ‘minor’, or ‘both’ to control whether major tick grids, minor tick grids, or both are affected.

`axis` can be ‘both’ (default), ‘x’, or ‘y’ to control which set of gridlines are drawn.

`kwargs` are used to set the grid line properties, eg:

```
ax.grid(color='r', linestyle='-', linewidth=2)
```

Valid `Line2D` `kwargs` are

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an <code>Axes</code> instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>any <code>matplotlib</code> color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>data</td>
<td>2D array (rows are x, y) or two 1D arrays</td>
</tr>
<tr>
<td>drawstyle</td>
<td>['default'</td>
</tr>
<tr>
<td>figure</td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>['full'</td>
</tr>
</tbody>
</table>
Table 70.13 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['-','--','-.',' :', 'None',' ',''] and any drawstyle in combination with a</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>marker</td>
<td>unknown</td>
</tr>
<tr>
<td>markeredgewidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt', 'round', 'projecting']</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>['miter', 'round', 'bevel']</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

```python
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='none', reduce_C_function=np.mean, mincnt=None, marginals=True, hold=None, **kwargs)
```

Make a hexagonal binning plot.

Call signature:

```python
hexbin(x, y, C = None, gridsize = 100, bins = None, xscale = 'linear', yscale = 'linear', cmap = None, norm = None, vmin = None, vmax = None, alpha = None, linewidths = None, edgecolors = 'none', reduce_C_function = np.mean, mincnt = None, marginals = True, **kwargs)
```
Make a hexagonal binning plot of $x$ versus $y$, where $x$, $y$ are 1-D sequences of the same length, $N$. If $C$ is $None$ (the default), this is a histogram of the number of occurrences of the observations at $(x[i],y[i])$.

If $C$ is specified, it specifies values at the coordinate $(x[i],y[i])$. These values are accumulated for each hexagonal bin and then reduced according to $reduce_C_function$, which defaults to numpy’s mean function (np.mean). (If $C$ is specified, it must also be a 1-D sequence of the same length as $x$ and $y$.) $x$, $y$ and/or $C$ may be masked arrays, in which case only unmasked points will be plotted.

Optional keyword arguments:

- **gridsize**: [100 | integer] The number of hexagons in the $x$-direction, default is 100. The corresponding number of hexagons in the $y$-direction is chosen such that the hexagons are approximately regular. Alternatively, gridsize can be a tuple with two elements specifying the number of hexagons in the $x$-direction and the $y$-direction.

- **bins**: [None | ‘log’ | integer | sequence] 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.

  - 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’] Use a linear or log10 scale on the horizontal axis.

- **yscale**: [‘linear’ | ‘log’] Use a linear or log10 scale on the vertical axis.

- **mincnt**: [None | a positive integer] If not None, only display cells with more than mincnt number of points in the cell.

- **marginals**: [True | False] if marginals is True, plot the marginal density as color-mapped rectangles along the bottom of the $x$-axis and left of the $y$-axis.

- **extent**: [None | scalars (left, right, bottom, top)] The limits of the bins. The default assigns the limits based on gridsize, $x$, $y$, xscale and yscale.

Other keyword arguments controlling color mapping and normalization arguments:

- **cmap**: [None | Colormap] a matplotlib.colors.Colormap instance. If None, defaults to rc image.cmap.

- **norm**: [None | Normalize] matplotlib.colors.Normalize instance is used to scale luminance data to 0,1.

- **vmin / vmax**: scalar vmin and vmax are used in conjunction with norm to normalize luminance data.

  - If either are None, the min and max of the color array $C$ is used. Note if you pass a norm instance, your settings for vmin and vmax will be ignored.

- **alpha**: scalar between 0 and 1, or None the alpha value for the patches

- **linewidths**: [None | scalar] If None, defaults to rc lines.linewidth. Note that this is a tuple, and if you set the linewidths argument you must set it as a sequence of floats, as required by RegularPolyCollection.

Other keyword arguments controlling the Collection properties:

- **edgecolors**: [None | ‘none’ | mpl color | color sequence] If ’none’, draws the edges in the same color as the fill color. This is the default, as it avoids unsightly unpainted pixels between the hexagons.

  - If None, draws the outlines in the default color.

  - If a matplotlib color arg or sequence of rgba tuples, draws the outlines in the specified color.
Here are the standard descriptions of all the `Collection` kwargs:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased</td>
<td>Boolean or sequence of booleans</td>
</tr>
<tr>
<td>array</td>
<td>unknown</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clim</td>
<td>a length 2 sequence of floats</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>cmap</td>
<td>a colormap or registered colormap name</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>colorbar</td>
<td>unknown</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>facecolor</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure Figure instance</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[‘/’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle</td>
<td>[‘solid’</td>
</tr>
<tr>
<td>linewidth</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>norm</td>
<td>unknown</td>
</tr>
<tr>
<td>offset_position</td>
<td>unknown</td>
</tr>
<tr>
<td>offsets</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>paths</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>pickradius</td>
<td>unknown</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>urls</td>
<td>unknown</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

The return value is a `PolyCollection` instance; use `get_array()` on this `PolyCollection` to get the counts in each hexagon. If `marginals` is `True`, horizontal bar and vertical bar (both `PolyCollections`) will be attached to the return collection as attributes `hbar` and `vbar`. 
Example:

```
Hexagon binning

With a log color scale

Additional kwargs: hold = [True|False] overrides default hold state

```

```
matplotlib.pyplot.hist(x, bins=10, range=None, normed=False, weights=None, cumulative=False, bottom=None, histtype='bar', align='mid', orientation='vertical', rwidth=None, log=False, color=None, label=None, stacked=False, hold=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.

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 at present.

Parameters

- **x**: array_like, shape (n, )
  
  Input values.

- **bins**: integer or array_like, optional, default: 10
  
  If an integer is given, bins + 1 bin edges are returned, consistently with `numpy.histogram()` for numpy version >= 1.3.

  Unequally spaced bins are supported if bins is a sequence.

- **range**: tuple, optional, default: None
The lower and upper range of the bins. Lower and upper outliers are ignored. If not provided, range is \((x\text{.min()}, x\text{.max()})\). 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 \(x\).

**normed** : boolean, optional, default: False
If True, the first element of the return tuple will be the counts normalized to form a probability density, i.e., \(n/\text{len}(x) \times \text{dbin}\), ie the integral of the histogram will sum to 1. If stacked is also True, the sum of the histograms is normalized to 1.

**weights** : array_like, shape (n,), optional, 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 normed is True, the weights are normalized, so that the integral of the density over the range remains 1.

**cumulative** : boolean, optional, default
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 normed is also True then the histogram is normalized such that the last bin equals 1. If cumulative evaluates to less than 0 (e.g., -1), the direction of accumulation is reversed. In this case, if normed is also True, then the histogram is normalized such that the first bin equals 1.

**histtype** : [‘bar’ | ‘barstacked’ | ‘step’ | ‘stepfilled’], optional
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’], optional, default: ‘mid’
Controls how the histogram is plotted.
- ‘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** : [‘horizontal’ | ‘vertical’], optional
If ‘horizontal’, barh will be used for bar-type histograms and the bottom kwarg will be the left edges.

**rwidth** : scalar, optional, default: None
The relative width of the bars as a fraction of the bin width. If None, automatically compute the width. Ignored if histtype = ‘step’ or ‘stepfilled’.

**log** : boolean, optional, default
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 (\(n, \text{bins}, \text{patches}\)) will be returned.
color : color or array_like of colors, optional, default: None
    Color spec or sequence of color specs, one per dataset. Default (None)
    uses the standard line color sequence.
label : string, optional, default: ‘’
    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 the legend command will work as expected.
stacked : boolean, optional, default
    If True, multiple data are stacked on top of each other If False mul-
    tiple data are aranged side by side if histtype is ‘bar’ or on top of each
    other if histtype is ‘step’
Returns tuple : (n, bins, patches) or ([n0, n1, ...], bins, [patches0, patches1,...])
Other Parameters kwargs : Patch properties
See also:

hist2d 2D histograms

Notes

Until numpy release 1.5, the underlying numpy histogram function was incorrect with
normed=’True if bin sizes were unequal. MPL inherited that error. It is now corrected within
MPL when using earlier numpy versions.
Examples

Additional kwargs: hold = [True]False] overrides default hold state

```
import matplotlib.pyplot as plt
plt.hist2d(x, y, bins=10, range=None, normed=False, weights=None,
cmin=None, cmax=None, hold=None, **kwargs)
```

Make a 2D histogram plot.

**Parameters** `x, y: array_like, shape (n, )`:

Input values

- **bins**: [None | int | [int, int] | array_like | [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, default: None

  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.

- **normed**: boolean, optional, default: False
  Normalize histogram.

- **weights**: array_like, shape (n, ), optional, default: None
  An array of values w_i weighing each sample (x_i, y_i).

- **cmin**: scalar, optional, default: None
  All bins that has count less than cmin will not be displayed and these count values in the return value count histogram will also be set to nan upon return.

- **cmax**: scalar, optional, default: None
  All bins that has count more than cmax will not be displayed (set to none before passing to imshow) and these count values in the return value count histogram will also be set to nan upon return.

**Returns**
The return value is ``(counts, xedges, yedges, Image)``.

**Other Parameters**
- **kwargs**: pcolorfast() properties.

**See also:**
- **hist**: 1D histogram

**Notes**
Rendering the histogram with a logarithmic color scale is accomplished by passing a colors.LogNorm instance to the norm keyword argument.
Examples

Additional kwargs: hold = [True|False] overrides default hold state

```python
matplotlib.pyplot.hlines(y, xmin, xmax, colors='k', linestyles='solid', label='', hold=None, **kwargs)
```

Plot horizontal lines at each y from xmin to xmax.

**Parameters**
- `y`: scalar or 1D array_like
  - y-indexes where to plot the lines.
- `xmin, xmax`: scalar or 1D array_like
  - Respective beginning and end of each line. If scalars are provided, all lines will have same length.
- `colors`: array_like of colors, optional, default: ‘k’
- `linestyles`: ['solid' | ‘dashed’ | ‘dashdot’ | ‘dotted’], optional
- `label`: string, optional, default: ‘’

**Returns**
- `lines`: LineCollection

**Other Parameters**
- `kwargs`: LineCollection properties.

See also:
- `vlines` vertical lines
Additional kwargs: hold = [True|False] overrides default hold state

```python
matplotlib.pyplot.hold(b=None)
```
Set the hold state. If `b` is None (default), toggle the hold state, else set the hold state to boolean value `b`:

```python
hold() # toggle hold
hold(True) # hold is on
hold(False) # hold is off
```

When `hold` is `True`, subsequent plot commands will be added to the current axes. When `hold` is `False`, the current axes and figure will be cleared on the next plot command.

```python
matplotlib.pyplot.hot()
```
Set the default colormap to hot and apply to current image if any. See help(colormaps) for more information

```python
matplotlib.pyplot.hsv()
```
Set the default colormap to hsv and apply to current image if any. See help(colormaps) for more information

```python
matplotlib.pyplot.imread(*args, **kwargs)
```
Read an image from a file into an array.

`fname` may be a string path or a Python file-like object. If using a file object, it must be opened in binary mode.

If `format` is provided, will try to read file of that type, otherwise the format is deduced from the filename. If nothing can be deduced, PNG is tried.

Return value is a numpy.array. For grayscale images, the return array is MxN. For RGB images, the return value is MxNx3. For RGBA images the return value is MxNx4.
Matplotlib can only read PNGs natively, but if PIL is installed, it will use it to load the image and return an array (if possible) which can be used with imshow().

```
matplotlib.pyplot.imsave(*args, **kwargs)
```
Save an array as in image file.

The output formats available depend on the backend being used.

**Arguments:**

- `fname`: A string containing a path to a filename, or a Python file-like object. If `format` is `None` and `fname` is a string, the output format is deduced from the extension of the filename.
- `arr`: An MxN (luminance), MxNx3 (RGB) or MxNx4 (RGBA) array.

**Keyword arguments:**

- `vmin/vmax`: `[None | scalar]` 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`: cmap is a colors.Colormap instance, eg cm.jet. If `None`, default to the rc image.cmap value.
- `format`: One of the file extensions supported by the active backend. Most backends support png, pdf, ps, eps and svg.
- `origin`: [‘upper’ | ‘lower’] Indicates where the [0,0] index of the array is in the upper left or lower left corner of the axes. Defaults to the rc image.origin value.
- `dpi`: The DPI to store in the metadata of the file. This does not affect the resolution of the output image.

```
matplotlib.pyplot.imshow(X, cmap=None, norm=None, aspect=None, interpolation=None, alpha=None, vmin=None, vmax=None, origin=None, extent=None, shape=None, filternorm=1, filterrad=4.0, imlim=None, resample=None, url=None, hold=None, **kwargs)
```
Display an image on the axes.

**Parameters**

- `X`: array_like, shape (n, m) or (n, m, 3) or (n, m, 4)
  Display the image in `X` to current axes. `X` may be a float array, a uint8 array or a PIL image. If `X` is an array, it can have the following shapes:
  - MxN – luminance (grayscale, float array only)
  - MxNx3 – RGB (float or uint8 array)
  - MxNx4 – RGBA (float or uint8 array)

  The value for each component of MxNx3 and MxNx4 float arrays should be in the range 0.0 to 1.0; MxN float arrays may be normalised.

- `cmap`: Colormap, optional, default: `None`
  If `None`, default to rc image.cmap value. cmap is ignored when `X` has RGB(A) information

- `aspect`: [‘auto’ | ‘equal’ | scalar], optional, default: `None`
  If ‘auto’, changes the image aspect ratio to match that of the axes.

  If ‘equal’, and `extent` is `None`, changes the axes aspect ratio to match that of the image. If `extent` is not `None`, the axes aspect ratio is changed to match that of the extent.

  If `None`, default to rc image.aspect value.

- `interpolation`: string, optional, default: `None`
If interpolation is None, default to rc `image.interpolation`. See also the `filternorm` and `filterrad` parameters. If interpolation is ‘none’, then no interpolation is performed on the Agg, ps and pdf backends. Other backends will fall back to ‘nearest’.

**norm**: `Normalize`, optional, default: None

A `Normalize` instance is used to scale luminance data to 0, 1. If None, use the default function: `normalize`. `norm` is only used if `X` is an array of floats.

**vmin, vmax**: scalar, optional, default: None

`vmin` and `vmax` are used in conjunction with `norm` to normalize luminance data. Note if you pass a `norm` instance, your settings for `vmin` and `vmax` will be ignored.

**alpha**: scalar, optional, default: None

The alpha blending value, between 0 (transparent) and 1 (opaque)

**origin**: ['upper'] | ['lower'], optional, default: None

Place the [0,0] index of the array in the upper left or lower left corner of the axes. If None, default to rc `image.origin`.

**extent**: scalars (left, right, bottom, top), optional, default: None

Data limits for the axes. The default assigns zero-based row, column indices to the x, y centers of the pixels.

**shape**: scalars (columns, rows), optional, default: None

For raw buffer images

**filternorm**: scalar, optional, default: 1

A parameter for the antigrain image resize filter. From the antigrain documentation, if `filternorm` = 1, 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**: scalar, optional, default: 4.0

The filter radius for filters that have a radius parameter, i.e. when interpolation is one of: ‘sinc’, ‘lanczos’ or ‘blackman’

Returns `image`: `AxesImage`

Other Parameters `**kwargs`: `Artist` properties.

See also:

`matshow` Plot a matrix or an array as an image.
Examples

Additional kwargs: hold = [True|False] overrides default hold state

```python
matplotlib.pyplot.ion()
    Turn interactive mode on.
```

```python
matplotlib.pyplot.ishold()
    Return the hold status of the current axes.
```

```python
matplotlib.pyplot.isinteractive()
    Return status of interactive mode.
```

```python
matplotlib.pyplot.jet()
    set the default colormap to jet and apply to current image if any. See help(colormaps) for more information
```

```python
matplotlib.pyplot.legend(*args, **kwargs)
    Place a legend on the current axes.
```

Call signature:
legend(*args, **kwargs)

Places legend at location `loc`. Labels are a sequence of strings and `loc` can be a string or an integer specifying the legend location.

To make a legend with existing lines:

```python
legend()
```

`legend()` by itself will try and build a legend using the label property of the lines/patches/collections. You can set the label of a line by doing:

```python
plot(x, y, label='my data')
```
or:

```python
line.set_label('my data').
```

If label is set to `'_nolegend_'`, the item will not be shown in legend.

To automatically generate the legend from labels:

```python
legend( ('label1', 'label2', 'label3') )
```

To make a legend for a list of lines and labels:

```python
legend( (line1, line2, line3), ('label1', 'label2', 'label3') )
```

To make a legend at a given location, using a location argument:

```python
legend( ('label1', 'label2', 'label3'), loc='upper left')
```
or:

```python
legend((line1, line2, line3), ('label1', 'label2', 'label3'), loc=2)
```

The location codes are:

<table>
<thead>
<tr>
<th>Location String</th>
<th>Location Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>'best'</td>
<td>0</td>
</tr>
<tr>
<td>'upper right'</td>
<td>1</td>
</tr>
<tr>
<td>'upper left'</td>
<td>2</td>
</tr>
<tr>
<td>'lower left'</td>
<td>3</td>
</tr>
<tr>
<td>'lower right'</td>
<td>4</td>
</tr>
<tr>
<td>'right'</td>
<td>5</td>
</tr>
<tr>
<td>'center left'</td>
<td>6</td>
</tr>
<tr>
<td>'center right'</td>
<td>7</td>
</tr>
<tr>
<td>'lower center'</td>
<td>8</td>
</tr>
<tr>
<td>'upper center'</td>
<td>9</td>
</tr>
<tr>
<td>'center'</td>
<td>10</td>
</tr>
</tbody>
</table>
Users can specify any arbitrary location for the legend using the `bbox_to_anchor` keyword argument. `bbox_to_anchor` can be an instance of BboxBase(or its derivatives) or a tuple of 2 or 4 floats. For example:

```python
loc = 'upper right', bbox_to_anchor = (0.5, 0.5)
```

will place the legend so that the upper right corner of the legend at the center of the axes.

The legend location can be specified in other coordinate, by using the `bbox_transform` keyword.

The `loc` itself can be a 2-tuple giving x,y of the lower-left corner of the legend in axes coords (`bbox_to_anchor` is ignored).

Keyword arguments:

- `prop`: [None | FontProperties | dict] A matplotlib.font_manager.FontProperties instance. If `prop` is a dictionary, a new instance will be created with `prop`. If `None`, use rc settings.
- `fontsize`: [size in points | ‘xx-small’ | ‘x-small’ | ‘small’ | ‘medium’ | ‘large’ | ‘x-large’ | ‘xx-large’] Set the font size. May be either a size string, relative to the default font size, or an absolute font size in points. This argument is only used if `prop` is not specified.
- `numpoints`: integer The number of points in the legend for line
- `scatterpoints`: integer The number of points in the legend for scatter plot
- `scatteryoffsets`: list of floats a list of yo offsets for scatter symbols in legend
- `markerscale`: [None | scalar] The relative size of legend markers vs. original. If `None`, use rc settings.
- `frameon`: [True | False] if `True`, draw a frame around the legend. The default is set by the rcParam ‘legend.frameon’
- `fancybox`: [None | False | True] if `True`, draw a frame with a round fancybox. If `None`, use rc settings
- `shadow`: [None | False | True] If `True`, draw a shadow behind legend. If `None`, use rc settings.
- `framealpha`: [None | float] If not `None`, alpha channel for legend frame. Default `None`.
- `ncol`: [integer] number of columns. default is 1
- `mode`: [“expand” | `None` ] if mode is “expand”, the legend will be horizontally expanded to fill the axes area (or `bbox_to_anchor`)
- `bbox_to_anchor`: an instance of BboxBase or a tuple of 2 or 4 floats the bbox that the legend will be anchored.
- `bbox_transform`: an instance of Transform | `None` ] the transform for the bbox. transAxes if `None`.
- `title`: [string] the legend title

Padding and spacing between various elements use following keywords parameters. These values are measure in font-size units. e.g., a fontsize of 10 points and a handlelength=5 implies a handlelength of 50 points. Values from rcParams will be used if `None`.  

70.1. `matplotlib.pyplot` 1123
**Keyword** | **Description**
---|---
borderpad | the fractional whitespace inside the legend border
labelspacing | the vertical space between the legend entries
handlelength | the length of the legend handles
handletextpad | the pad between the legend handle and text
borderaxespad | the pad between the axes and legend border
columnspacing | the spacing between columns

*Note:* Not all kinds of artist are supported by the legend command. See *Legend guide* for details.

**Example:**

```
0.0 0.5 1.0 1.5 2.0 2.5 3.0
0
5
10
15
20
25
Model length
Data length
Total message length
```

**See also:**

*Legend guide.*

```
matplotlib.pyplot.locator_params(axis='both', tight=None, **kwargs)
```

Control behavior of tick locators.

Keyword arguments:

- **axis** ['x' | 'y' | 'both'] Axis on which to operate; default is ‘both’.
- **tight** [True | False | None] Parameter passed to autoscale_view(). Default is None, for no change. Remaining keyword arguments are passed to directly to the `set_params()` method.

Typically one might want to reduce the maximum number of ticks and use tight bounds when plotting small subplots, for example:
ax.locator_params(tight=True, nbins=4)

Because the locator is involved in autoscaling, autoscale_view() is called automatically after the parameters are changed.

This presently works only for the MaxNLocator used by default on linear axes, but it may be generalized.

matplotlib.pyplot.loglog(*args, **kwargs)

Make a plot with log scaling on both the x and y axis.

Call signature:

```
loglog(*args, **kwargs)
```

loglog() supports all the keyword arguments of plot() and matplotlib.axes.Axes.set_xscale() / matplotlib.axes.Axes.set_yscale().

Notable keyword arguments:

- **base** or basey: scalar > 1 Base of the x/y logarithm
- **subsx** or subsy: [None | sequence] The location of the minor x/y ticks;
  None defaults to autosubs, which depend on the number of decades in the plot; see matplotlib.axes.Axes.set_xscale() / matplotlib.axes.Axes.set_yscale() for details
- **nonposx** or nonposy: ['mask' | 'clip'] Non-positive values in x or y can be masked as invalid, or clipped to a very small positive number

The remaining valid kwargs are Line2D properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased</td>
<td>True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>data</td>
<td>2D array (rows are x, y) or two 1D arrays</td>
</tr>
<tr>
<td>drawstyle</td>
<td>['default'</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>['full'</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
</tbody>
</table>
Table 70.15 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>linestyle or ls</td>
<td>['-'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>marker</td>
<td>unknown</td>
</tr>
<tr>
<td>markeredgewidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any <em>matplotlib</em> color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function \fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>transform</td>
<td>a <em>matplotlibtransforms.Transform</em> instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

Example:
Additional kwargs: hold = [True|False] overrides default hold state

matplotlib.pyplot.margins(*args, **kw)
Set or retrieve autoscaling margins.
signatures:
margins()
returns xmargin, ymargin
margins(margin)
margins(xmargin, ymargin)
margins(x=xmargin, y=ymargin)
margins(..., tight=False)

All three forms above set the xmargin and ymargin parameters. All keyword parameters are optional. A single argument specifies both xmargin and ymargin. 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.

```
matplotlib.pyplot.matshow(A, fignum=None, **kw)
```

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.
```

With the exception of fignum, keyword arguments are passed to imshow(). You may set the origin kwarg to “lower” if you want the first row in the array to be at the bottom instead of the top.

```
fignum: [ None | integer | False ] By default, matshow() creates a new figure window with automatic numbering. If fignum is given as an integer, the created figure will use this figure number. Because of how matshow() tries to set the figure aspect ratio to be the one of the array, if you provide the number of an already existing figure, strange things may happen.
```

If fignum is False or 0, a new figure window will NOT be created.

```
matplotlib.pyplot.minorticks_off()
```

Remove minor ticks from the current plot.

```
matplotlib.pyplot.minorticks_on()
```

Display minor ticks on the current plot.

```
Displaying minor ticks reduces performance; turn them off using minorticks_off() if drawing speed is a problem.
```

```
matplotlib.pyplot.over(func, *args, **kwargs)
```

Call a function with hold(True).

```
Calls:
```

```
func(*args, **kwargs)
```

with hold(True) and then restores the hold state.

```
matplotlib.pyplot.pause(interval)
```

Pause for interval seconds.

```
If there is an active figure it will be updated and displayed, and the GUI event loop will run during the pause.
```

```
If there is no active figure, or if a non-interactive backend is in use, this executes time.sleep(interval).
```

```
This can be used for crude animation. For more complex animation, see matplotlib.animation.
```

This function is experimental; its behavior may be changed or extended in a future release.

```
matplotlib.pyplot.pcolor(*args, **kwargs)
```

Create a pseudocolor plot of a 2-D array.

**Note:** pcolor can be very slow for large arrays; consider using the similar but much faster pcolormesh() instead.
Call signatures:

```python
pcolor(C, **kwargs)
pcolor(X, Y, C, **kwargs)
```

C is the array of color values.

X and Y, if given, specify the (x, y) coordinates of the colored quadrilaterals; the quadrilateral for
C[i,j] has corners at:

\[(X[i, j], Y[i, j]), (X[i, j+1], Y[i, j+1]), (X[i+1, j], Y[i+1, j]), (X[i+1, j+1], Y[i+1, j+1]).\]

Ideally the dimensions of X and Y should be one greater than those of C; if the dimensions are the
same, then the last row and column of C will be ignored.

Note that the the column index corresponds to the x-coordinate, and the row index corresponds to y;
for details, see the Grid Orientation section below.

If either or both of X and Y are 1-D arrays or column vectors, they will be expanded as needed into
the appropriate 2-D arrays, making a rectangular grid.

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.

Keyword arguments:

- **cmp**: [ None | Colormap ] A `matplotlib.colors.Colormap` instance. If None, use
colorbar by default.
- **norm**: [ None | Normalize ] An `matplotlib.colors.Normalize` instance is used to
scale luminance data to 0,1. If None, defaults to `normalize()`.
- **vmin/vmax**: [ None | scalar ] vmin and vmax are used in conjunction with norm to nor-
malize luminance data. If either is None, it is autoscaled to the respective min or
max of the color array C. If not None, vmin or vmax passed in here override any
pre-existing values supplied in the norm instance.
- **shading**: [ ‘flat’ | ‘faceted’ ] If ‘faceted’, a black grid is drawn around each rectangle; if
‘flat’, edges are not drawn. Default is ‘flat’, contrary to MATLAB.

This kwarg is deprecated; please use ‘edgecolors’ instead:

- shading='flat' – edgecolors='none'
- shading='faceted’ – edgecolors='k'
- **edgecolors**: [ None | 'none' | color | color sequence] If None, the rc setting is used by
default.

If 'none', edges will not be visible.

An mpl color or sequence of colors will set the edge color

- **alpha**: 0 <= scalar <= 1 or None the alpha blending value

Return value is a `matplotlib.collections.Collection` instance. The grid orientation follows
the MATLAB convention: an array C with shape (nrows, ncolumns) is plotted with the column number

---

70.1. matplotlib.pyplot

---
as $X$ and the row number as $Y$, increasing up; hence it is plotted the way the array would be printed, except that the $Y$ axis is reversed. That is, $C$ is taken as $C^{y(*)x}$.

Similarly for `meshgrid()`:

```python
x = np.arange(5)
y = np.arange(3)
X, Y = np.meshgrid(x, y)
```

is equivalent to:

```python
X = array([[0, 1, 2, 3, 4],
           [0, 1, 2, 3, 4],
           [0, 1, 2, 3, 4]])
Y = array([[0, 0, 0, 0, 0],
           [1, 1, 1, 1, 1],
           [2, 2, 2, 2, 2]])
```

so if you have:

```python
C = rand(len(x), len(y))
```

then you need to transpose $C$:

```python
pcolor(X, Y, C.T)
```

or:

```python
pcolor(C.T)
```

MATLAB `pcolor()` always discards the last row and column of $C$, but matplotlib displays the last row and column if $X$ and $Y$ are not specified, or if $X$ and $Y$ have one more row and column than $C$.

`kwargs` can be used to control the `PolyCollection` properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or antialiaseds</td>
<td>Boolean or sequence of booleans</td>
</tr>
<tr>
<td>array</td>
<td>unknown</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clim</td>
<td>a length 2 sequence of floats</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[ (Path, Transform)</td>
</tr>
<tr>
<td>cmap</td>
<td>a colormap or registered colormap name</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>colorbar</td>
<td>unknown</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or edgecolors</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>facecolor or facecolors</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or linestyles</td>
<td>['solid', 'dashed', 'dashdot', 'dotted'] [ (offset, on-off-dash-seq) ]</td>
</tr>
<tr>
<td>linewidth or lw or linewidths</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>norm</td>
<td>unknown</td>
</tr>
<tr>
<td>offset_position</td>
<td>unknown</td>
</tr>
<tr>
<td>offsets</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>paths</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None,float,boolean,callable]</td>
</tr>
<tr>
<td>pickradius</td>
<td>unknown</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>urls</td>
<td>unknown</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

**Note:** 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"], which defaults to True. Stroking the edges may be preferred if alpha is 1, but will cause artifacts otherwise.

**See also:**

`pcolormesh()` For an explanation of the differences between pcolor and pcolormesh.

Additional kwargs: hold = [True]False] overrides default hold state

`matplotlib.pyplot.pcolormesh(*args, **kwargs)`

Plot a quadrilateral mesh.

Call signatures:
Create a pseudocolor plot of a 2-D array.

`pcolormesh(C)`
`pcolormesh(X, Y, C)`
`pcolormesh(C, **kwargs)`

`pcolormesh` is similar to `pcolor()`, but uses a different mechanism and returns a different object; `pcolor` returns a `PolyCollection` but `pcolormesh` returns a `QuadMesh`. It is much faster, so it is almost always preferred for large arrays.

`C` may be a masked array, but `X` and `Y` may not. Masked array support is implemented via `cmap` and `norm`; in contrast, `pcolor()` simply does not draw quadrilaterals with masked colors or vertices.

Keyword arguments:

- `cmap`: `[None] | Colormap` A `matplotlib.colors.Colormap` instance. If `None`, use rc settings.
- `norm`: `[None] | Normalize` A `matplotlib.colors.Normalize` instance is used to scale luminance data to 0,1. If `None`, defaults to `normalize()`.
- `vmin/vmax`: `[None] | scalar` `vmin` and `vmax` are used in conjunction with `norm` to normalize luminance data. If either is `None`, it is autoscaled to the respective min or max of the color array `C`. If not `None`, `vmin` or `vmax` passed in here override any pre-existing values supplied in the `norm` instance.
- `shading`: `['flat'] | 'gouraud'` 'flat' indicates a solid color for each quad. When 'gouraud', each quad will be Gouraud shaded. When gouraud shading, edgecolors is ignored.
- `edgecolors`: `[None] | 'None' | 'face' | color | color sequence]
  If `None`, the rc setting is used by default.
  If 'None', edges will not be visible.
  If 'face', edges will have the same color as the faces.

An mpl color or sequence of colors will set the edge color

- `alpha`: `0 <= scalar <= 1 or None` the alpha blending value

Return value is a `matplotlib.collections.QuadMesh` object.

`kwargs` can be used to control the `matplotlib.collections.QuadMesh` properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>agg_filter</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>alpha</code></td>
<td>float or None</td>
</tr>
<tr>
<td><code>animated</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>antialiased</code> or <code>antialiaseds</code></td>
<td>Boolean or sequence of booleans</td>
</tr>
<tr>
<td><code>array</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>axes</code></td>
<td>an <code>Axes</code> instance</td>
</tr>
<tr>
<td><code>clim</code></td>
<td>a length 2 sequence of floats</td>
</tr>
<tr>
<td><code>clip_box</code></td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td><code>clip_on</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>clip_path</code></td>
<td>[(Path, Transform)</td>
</tr>
</tbody>
</table>

Continued on next page
Table 70.17 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmap</td>
<td>a colormap or registered colormap name</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>colorbar</td>
<td>unknown</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or edgecolors</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>facecolor or facecolors</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[ '/', '', '</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with '%s' conversion.</td>
</tr>
<tr>
<td>linestyle or linestyles or dashes</td>
<td>['solid', 'dashed', 'dashdot', 'dotted', (offset, on-off-dash-seq) ]</td>
</tr>
<tr>
<td>linewidth or lw or linewidths</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>norm</td>
<td>unknown</td>
</tr>
<tr>
<td>offset_position</td>
<td>unknown</td>
</tr>
<tr>
<td>offsets</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>paths</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None]float[boolean]callable</td>
</tr>
<tr>
<td>pickradius</td>
<td>unknown</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>urls</td>
<td>unknown</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

See also:

**pcolor()**: For an explanation of the grid orientation and the expansion of 1-D $X$ and/or $Y$ to 2-D arrays.

Additional kwargs: hold = [True|False] overrides default hold state

```python
matplotlib.pyplot.pie(x, explode=None, labels=None, colors=None, autopct=None, pctdistance=0.6, shadow=False, labeldistance=1.1, startangle=None, radius=None, hold=None)
```

Plot a pie chart.

Call signature:

```python
pie(x, explode=None, labels=None, colors=('b', 'g', 'r', 'c', 'm', 'y', 'k', 'w'),
```

70.1. matplotlib.pyplot.pyplot 1133
Make a pie chart of array \(x\). The fractional area of each wedge is given by \(x/\text{sum}(x)\). If \(\text{sum}(x) \leq 1\), then the values of \(x\) give the fractional area directly and the array will not be normalized. The wedges are plotted counterclockwise, by default starting from the \(x\)-axis.

Keyword arguments:

- **explode**: [\(None\) | len(\(x\)) sequence] If not \(None\), is a \(\text{len}(x)\) array which specifies the fraction of the radius with which to offset each wedge.
- **colors**: [\(None\) | color sequence] A sequence of matplotlib color args through which the pie chart will cycle.
- **labels**: [\(None\) | len(\(x\)) string sequence of strings] A sequence of strings providing the labels for each wedge.
- **autopct**: [\(None\) | format string | format function] 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 \(\text{fmt\%pct}\). If it is a function, it will be called.
- **pctdistance**: scalar The ratio between the center of each pie slice and the start of the text generated by \(\text{autopct}\). Ignored if \(\text{autopct}\) is \(None\); default is 0.6.
- **labeldistance**: scalar The radial distance at which the pie labels are drawn
- **shadow**: [\(False\) | \(True\)] Draw a shadow beneath the pie.
- **startangle**: [\(None\) | Offset angle] If not \(None\), rotates the start of the pie chart by \(angle\) degrees counterclockwise from the \(x\)-axis.
- **radius**: [\(None\) | scalar] The radius of the pie, if \(radius\) is \(None\) it will be set to 1.

The pie chart will probably look best if the figure and axes are square, or the Axes aspect is equal. e.g.:

```python
figure(figsize=(8,8))
ax = axes([0.1, 0.1, 0.8, 0.8])
```

or:

```python
axes(aspect=1)
```

**Return value**: If \(\text{autopct}\) is \(None\), return the tuple \((\text{patches}, \text{texts})\):

- \(\text{patches}\) is a sequence of `matplotlib.patches.Wedge` instances
- \(\text{texts}\) is a list of the label `matplotlib.text.Text` instances.

If \(\text{autopct}\) is not \(None\), return the tuple \((\text{patches}, \text{texts}, \text{autotexts})\), where \(\text{patches}\) and \(\text{texts}\) are as above, and \(\text{autotexts}\) is a list of `Text` instances for the numeric labels.

Additional kwargs: `hold` = [True,False] overrides default hold state

```python
matplotlib.pyplot.pink()
```

set the default colormap to pink and apply to current image if any. See help(colormaps) for more information

```python
matplotlib.pyplot.plot(*args, **kwargs)
```

Plot lines and/or markers to the \(Axes\). \(args\) is a variable length argument, allowing for multiple \(x, y\)
pairs with an optional format string. For example, each of the following is legal:

```python
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
```

If x and/or y is 2-dimensional, then the corresponding columns will be plotted.

An arbitrary number of x, y, fmt groups can be specified, as in:

```python
a.plot(x1, y1, 'g^', x2, y2, 'g-')
```

Return value is a list of lines that were added.

By default, each line is assigned a different color specified by a ‘color cycle’. To
give a list of different colors, you can edit the axes.color_cycle rcParam. Alternatively, you can use
`set_default_color_cycle()`.

The following format string characters are accepted to control the line style or marker:

<table>
<thead>
<tr>
<th>character</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'-'</td>
<td>solid line style</td>
</tr>
<tr>
<td>'--'</td>
<td>dashed line style</td>
</tr>
<tr>
<td>'-.'</td>
<td>dash-dot line style</td>
</tr>
<tr>
<td>':'</td>
<td>dotted line style</td>
</tr>
<tr>
<td>','</td>
<td>point marker</td>
</tr>
<tr>
<td>'.'</td>
<td>pixel marker</td>
</tr>
<tr>
<td>'o'</td>
<td>circle marker</td>
</tr>
<tr>
<td>'v'</td>
<td>triangle_down marker</td>
</tr>
<tr>
<td>'^'</td>
<td>triangle_up marker</td>
</tr>
<tr>
<td>'&lt;'</td>
<td>triangle_left marker</td>
</tr>
<tr>
<td>'&gt;'</td>
<td>triangle_right marker</td>
</tr>
<tr>
<td>'1'</td>
<td>tri_down marker</td>
</tr>
<tr>
<td>'2'</td>
<td>tri_up marker</td>
</tr>
<tr>
<td>'3'</td>
<td>tri_left marker</td>
</tr>
<tr>
<td>'4'</td>
<td>tri_right marker</td>
</tr>
<tr>
<td>'s'</td>
<td>square marker</td>
</tr>
<tr>
<td>'p'</td>
<td>pentagon marker</td>
</tr>
<tr>
<td>'<em>r</em>'</td>
<td>star marker</td>
</tr>
<tr>
<td>'h'</td>
<td>hexagon1 marker</td>
</tr>
<tr>
<td>'H'</td>
<td>hexagon2 marker</td>
</tr>
<tr>
<td>'+'</td>
<td>plus marker</td>
</tr>
<tr>
<td>'x'</td>
<td>x marker</td>
</tr>
<tr>
<td>'D'</td>
<td>diamond marker</td>
</tr>
<tr>
<td>'d'</td>
<td>thin_diamond marker</td>
</tr>
<tr>
<td>'</td>
<td>'</td>
</tr>
<tr>
<td>'_'</td>
<td>hline marker</td>
</tr>
</tbody>
</table>

The following color abbreviations are supported:
<table>
<thead>
<tr>
<th>character</th>
<th>color</th>
</tr>
</thead>
<tbody>
<tr>
<td>'b'</td>
<td>blue</td>
</tr>
<tr>
<td>'g'</td>
<td>green</td>
</tr>
<tr>
<td>'r'</td>
<td>red</td>
</tr>
<tr>
<td>'c'</td>
<td>cyan</td>
</tr>
<tr>
<td>'m'</td>
<td>magenta</td>
</tr>
<tr>
<td>'y'</td>
<td>yellow</td>
</tr>
<tr>
<td>'k'</td>
<td>black</td>
</tr>
<tr>
<td>'w'</td>
<td>white</td>
</tr>
</tbody>
</table>

In addition, you can specify colors in many weird and wonderful ways, including full names ('green'), hex strings ('#008000'), RGB or RGBA tuples ((0,1,0,1)) or grayscale intensities as a string ('0.8'). Of these, the string specifications can be used in place of a fmt group, but the tuple forms can be used only as kwargs.

Line styles and colors are combined in a single format string, as in 'bo' for blue circles.

The kwargs can be used to set line properties (any property that has a set_ method). You can use this to set a line label (for auto legends), linewidth, antialiased, marker face color, etc. Here is an example:

```python
plot([1,2,3], [1,2,3], 'go-', label='line 1', linewidth=2)
plot([1,2,3], [1,4,9], 'rs', label='line 2')
axis([0, 4, 0, 10])
legend()
```

If you make multiple lines with one plot command, the kwargs apply to all those lines, e.g.:

```python
plot(x1, y1, x2, y2, antialiased=False)
```

Neither line will be antialiased.

You do not need to use format strings, which are just abbreviations. All of the line properties can be controlled by keyword arguments. For example, you can set the color, marker, linestyle, and markerfacecolor with:

```python
plot(x, y, color='green', linestyle='dashed', marker='o',
     markerfacecolor='blue', markersize=12).
```

See Line2D for details.

The kwargs are Line2D properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>data</td>
<td>2D array (rows are x, y) or two 1D arrays</td>
</tr>
<tr>
<td>drawstyle</td>
<td>['default'</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>['full'</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['-.'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>marker</td>
<td>unknown</td>
</tr>
<tr>
<td>markeredgecolor or mec</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markeredgewidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

kwarg scalex and scaley, if defined, are passed on to autoscale_view() to determine whether the x and y axes are autoscaled; the default is True.

Additional kwarg: hold = [True|False] overrides default hold state
matplotlib.pyplot.plot_date(x, y, fmt='bo', tz=None, xdate=True, ydate=False, hold=None, **kwargs)

Plot with data with dates.

Call signature:

plot_date(x, y, fmt='bo', tz=None, xdate=True, ydate=False, hold=None, **kwargs)

Similar to the plot() command, except the x or y (or both) data is considered to be dates, and the axis is labeled accordingly.

x and/or y can be a sequence of dates represented as float days since 0001-01-01 UTC.

Keyword arguments:

fmt: string The plot format string.
tz: [ None | timezone string | tzinfo instance] The time zone to use in labeling dates.
    If None, defaults to rc value.

xdate: [ True | False ] If True, the x-axis will be labeled with dates.
ydate: [ False | True ] If True, the y-axis will be labeled with dates.

Note if you are using custom date tickers and formatters, it may be necessary to set the formatters/locators after the call to plot_date() since plot_date() will set the default tick locator to matplotlib.dates.AutoDateLocator (if the tick locator is not already set to a matplotlib.dates.DateLocator instance) and the default tick formatter to matplotlib.dates.AutoDateFormatter (if the tick formatter is not already set to a matplotlib.dates.DateFormatter instance).

Valid kwarg are Line2D properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>[‘butt’</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>[‘miter’</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>data</td>
<td>2D array (rows are x, y) or two 1D arrays</td>
</tr>
<tr>
<td>drawstyle</td>
<td>[‘default’</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>[‘full’</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
</tbody>
</table>
Table 70.19 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>linestyle or ls</td>
<td>['-', '--', '-.', ':' ] and any drawstyle in combination with a linestype</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>marker</td>
<td>unknown</td>
</tr>
<tr>
<td>markeredgcolor or mec</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markeredgewidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt', 'round', 'projecting']</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>['miter', 'round', 'bevel']</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

See also:
dates for helper functions
date2num(), num2date() and drange() for help on creating the required floating point dates.

Additional kwargs: hold = [True|False] overrides default hold state

matplotlib.pyplot.plotfile(fname, cols=(0, ), plotfuncs=None, comments='#', skiprows=0, checkrows=5, delimiter=' ', names=None, subplots=True, newfig=True, **kwargs)

Plot the data in in a file.

cols is a sequence of column identifiers to plot. An identifier is either an int or a string. If it is an int, it indicates the column number. If it is a string, it indicates the column header. matplotlib will make column headers lower case, replace spaces with underscores, and remove all illegal characters; so 'Adj Close*' will have name 'adj_close'.

- If len(cols) == 1, only that column will be plotted on the y axis.
- If len(cols) > 1, the first element will be an identifier for data for the x axis and the remaining elements will be the column indexes for multiple subplots if subplots is True (the default), or for lines in a single subplot if subplots is False.
plotfuncs, if not None, is a dictionary mapping identifier to an Axes plotting function as a string. Default is ‘plot’, other choices are ‘semilogy’, ‘fill’, ‘bar’, etc. You must use the same type of identifier in the cols vector as you use in the plotfuncs dictionary, e.g., integer column numbers in both or column names in both. If subplots is False, then including any function such as ‘semilogy’ that changes the axis scaling will set the scaling for all columns.

comments, skiprows, checkrows, delimiter, and names are all passed on to matplotlib.pylab.csv2rec() to load the data into a record array.

If newfig is True, the plot always will be made in a new figure; if False, it will be made in the current figure if one exists, else in a new figure.

kwargs are passed on to plotting functions.

Example usage:

```python
# plot the 2nd and 4th column against the 1st in two subplots
plotfile(fname, (0,1,3))

# plot using column names; specify an alternate plot type for volume
plotfile(fname, ('date', 'volume', 'adj_close'),
         plotfuncs={'volume': 'semilogy'})
```

Note: plotfile is intended as a convenience for quickly plotting data from flat files; it is not intended as an alternative interface to general plotting with pyplot or matplotlib.

```python
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().

matplotlib.pyplot.prism()
set the default colormap to prism and apply to current image if any. See help(colormaps) for more information
```

```python
matplotlib.pyplot.psd(x, NFFT=256, Fs=2, Fc=0, detrend=<function detrend_none at 0x24627d0>, window=<function window_hanning at 0x2462578>, noverlap=0, pad_to=None, sides='default', scale_by_freq=None, hold=None, **kwargs)
Plot the power spectral density.
call signature:

psd(x, NFFT=256, Fs=2, Fc=0, detrend=mlab.detrend_none, window=mlab.window_hanning, noverlap=0, pad_to=None, sides='default', scale_by_freq=None, **kwargs)

The power spectral density 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 $|\text{fft}(i)|^2$ of each segment $i$ are averaged to compute $P_{xx}$, with a scaling to correct for power loss due to windowing. $Fs$ is the sampling frequency.

Keyword arguments:

- **NFFT**: integer The number of data points used in each block for the FFT. Must be even; a power 2 is most efficient. The default value is 256. This should NOT be used to get zero padding, or the scaling of the result will be incorrect. Use `pad_to` for this instead.

- **Fs**: scalar The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, `freqs`, in cycles per time unit. The default value is 2.

- **detrend**: callable 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 it is a function. The `pylab` module defines `detrend_none()`, `detrend_mean()`, and `detrend_linear()`, but you can use a custom function as well.

- **window**: callable or ndarray A function or a vector of length `NFFT`. To create window vectors see `window_hanning()`, `window_none()`, `numpy.blackman()`, `numpy.hanning()`, `numpy.hamming()`, `scipy.signal()`, `scipy.signal.get_window()`, etc. The default is `window_hanning()`. 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.

- **pad_to**: integer 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 psd (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`

- **sides**: [‘default’ | ‘onesided’ | ‘twosided’] Specifies which sides of the PSD to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. ‘onesided’ forces the return of a one-sided PSD, while ‘twosided’ forces two-sided.

- **scale_by_freq**: boolean Specifies 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**: integer The number of points of overlap between blocks. The default value is 0 (no overlap).

- **Fc**: integer The center frequency of $x$ (defaults to 0), 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 the tuple ($P_{xx}$, `freqs`).

For plotting, the power is plotted as $10 \log_{10}(P_{xx})$ for decibels, though $P_{xx}$ itself is returned.


kwargs control the `Line2D` properties:
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>data</td>
<td>2D array (rows are x, y) or two 1D arrays</td>
</tr>
<tr>
<td>drawstyle</td>
<td>['default'</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>['full'</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with <code>%s</code> conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['-.'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>marker</td>
<td>unknown</td>
</tr>
<tr>
<td>markeredgecolor or mec</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markeredgewidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>
Example:

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
ax.quiver(*args, **kw)
```

Call signatures:

```
quiver(U, V, **kw)
quiver(U, V, C, **kw)
quiver(X, Y, U, V, **kw)
quiver(X, Y, U, V, C, **kw)
```

Arguments:

- **X, Y**: The x and y coordinates of the arrow locations (default is tail of arrow; see `pivot` kwarg)
- **U, V**: Give the x and y components of the arrow vectors
- **C**: An optional array used to map colors to the arrows

All arguments may be 1-D or 2-D arrays or sequences. If X and Y are absent, they will be generated as a uniform grid. If U and V are 2-D arrays but X and Y are 1-D, and if len(X) and len(Y) match the column and row dimensions of U, then X and Y will be expanded with `numpy.meshgrid()`.

**U, V, C** may be masked arrays, but masked X, Y are not supported at present.

Keyword arguments:
units: ['width' | 'height' | 'dots' | 'inches' | 'x' | 'y' | 'xy'] Arrow units; the arrow dimensions except for length are in multiples of this unit.

- 'width' or 'height': the width or height of the axes
- 'dots' or 'inches': pixels or inches, based on the figure dpi
- 'x', 'y', or 'xy': X, Y, or sqrt(X^2+Y^2) 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' | array] With the default 'uv', the arrow aspect ratio is 1, so that if U*==*V the angle of the arrow on the plot is 45 degrees CCW from the x-axis. With 'xy', the arrow points from (x,y) to (x+u, y+v). Alternatively, arbitrary angles may be specified as an array of values in degrees, CCW from the x-axis.

scale: [None | float] Data units per arrow length unit, e.g., m/s per plot width; a smaller scale parameter makes the arrow longer. 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: None, or any of the units options. For example, if 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', then the vector will be half the width 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: 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: scalar Head width as multiple of shaft width, default is 3

headlength: scalar Head length as multiple of shaft width, default is 5

headaxislength: scalar Head length at shaft intersection, default is 4.5

minshaft: scalar Length below which arrow scales, in units of head length. Do not set this to less than 1, or small arrows will look terrible! Default is 1

minlength: scalar Minimum length as a multiple of shaft width; if an arrow length is less than this, plot a dot (hexagon) of this diameter instead. Default is 1.

pivot: ['tail' | 'middle' | 'tip'] The part of the arrow that is at the grid point; the arrow rotates about this point, hence the name pivot.

color: [color | color sequence] This is a synonym for the PolyCollection facecolor kwarg. If C has been set, color has no effect.

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.

linewdths and edgecolors can be used to customize the arrow outlines. Additional PolyCollection keyword arguments:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
</tbody>
</table>

Continued on next page
Table 70.21 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or antialiaseds</td>
<td>Boolean or sequence of booleans</td>
</tr>
<tr>
<td>array</td>
<td>unknown</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clim</td>
<td>a length 2 sequence of floats</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>cmap</td>
<td>a colormap or registered colormap name</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>colorbar</td>
<td>unknown</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgewidth or edgewidths</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>facecolor or facecolors</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>['/'</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with '%s' conversion.</td>
</tr>
<tr>
<td>linestyle or linestyle</td>
<td>['solid'</td>
</tr>
<tr>
<td>linewidth or lw or</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>linewidths</td>
<td>unknown</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>norm</td>
<td>unknown</td>
</tr>
<tr>
<td>offset_position</td>
<td>unknown</td>
</tr>
<tr>
<td>offsets</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>paths</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None]float[boolean]call{able}</td>
</tr>
<tr>
<td>pickradius</td>
<td>unknown</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>urls</td>
<td>unknown</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

Additional kwargs: hold = [True|False] overrides default hold state

matplotlib.pyplot.quiverkey(*args, **kw)
Add a key to a quiver plot.
Call signature:
quiverkey(Q, X, Y, U, label, **kw)

Arguments:

- **Q**: The Quiver instance returned by a call to quiver.
- **X, Y**: The location of the key; additional explanation follows.
- **U**: The length of the key.
- **label**: A string with the length and units of the key.

Keyword arguments:

- **coordinates**: [‘axes’ | ‘figure’ | ‘data’ | ‘inches’] 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**: 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**: Distance in inches between the arrow and the label. Default is 0.1
- **labelcolor**: defaults to default Text color.
- **fontproperties**: A dictionary with keyword arguments accepted by the FontProperties initializer: family, style, variant, size, weight

Any additional keyword arguments are used to override vector properties taken from Q.

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.

Additional kwargs: **hold** = [True|False] overrides default hold state

matplotlib.pyplot.rc(*args, **kwargs)

Set the current rc params. 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, eg:

```
rc('lines', linewidth=2, color='r')
```

sets the current rc params and is equivalent to:

```
rcParams['lines.linewidth'] = 2
rcParams['lines.color'] = 'r'
```

The following aliases are available to save typing for interactive users:
Thus you could abbreviate the above rc command as:

```python
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:

```python
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 `rcdefaults()` to restore the default rc params after changes.

```python
matplotlib.pyplot.rc_context(rc=None, fname=None)
```

Return a context manager for managing rc settings.

This allows one to do:

```python
with mpl.rc_context(fname='screen.rc'):
    plt.plot(x, a)
with mpl.rc_context(fname='print.rc'):
    plt.plot(x, b)
    plt.plot(x, c)
```

The ‘a’ vs ‘x’ and ‘c’ vs ‘x’ plots would have settings from ‘screen.rc’, while the ‘b’ vs ‘x’ plot would have settings from ‘print.rc’.

A dictionary can also be passed to the context manager:

```python
with mpl.rc_context(rc={'text.usetex': True}, fname='screen.rc'):
    plt.plot(x, a)
```

The ‘rc’ dictionary takes precedence over the settings loaded from ‘fname’. Passing a dictionary only is also valid.

```python
matplotlib.pyplot.rcdefaults()
```

Restore the default rc params. These are not the params loaded by the rc file, but mpl’s internal params. See rc_file_defaults for reloading the default params from the rc file.
matplotlib.pyplot.rgrids(*args, **kwargs)
Get or set the radial gridlines on a polar plot.

call signatures:

lines, labels = rgrids()
lines, labels = rgrids(radii, labels=None, angle=22.5, **kwargs)

When called with no arguments, rgrid() simply returns the tuple (lines, labels), where lines is an array of radial gridlines (Line2D instances) and labels is an array of tick labels (Text instances). When called with arguments, the labels will appear at the specified radial distances and angles.

labels, if not None, is a len(radii) list of strings of the labels to use at each angle.

If labels is None, the rformatter will be used

Examples:

# set the locations of the radial gridlines and labels
lines, labels = rgrids( (0.25, 0.5, 1.0) )

# set the locations and labels of the radial gridlines and labels
lines, labels = rgrids( (0.25, 0.5, 1.0), ('Tom', 'Dick', 'Harry' )

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)

The output formats available depend on the backend being used.

Arguments:

fname: A string containing a path to a filename, or a Python file-like object, or possibly some backend-dependent object such as PdfPages.

If format is None and fname is a string, the output format is deduced from the extension of the filename. If the filename has no extension, the value of the rc parameter savefig.format is used.

If fname is not a string, remember to specify format to ensure that the correct backend is used.

Keyword arguments:

dpi: [None | scalar > 0 ] The resolution in dots per inch. If None it will default to the value savefig.dpi in the matplotlibrc file.

facecolor, edgecolor: the colors of the figure rectangle

orientation: [‘landscape’ | ‘portrait’ ] not supported on all backends; currently only on postscript output

**format**: One of the file extensions supported by the active backend. Most backends support png, pdf, ps, eps and svg.

**transparent**: 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.

**frameon**: If True, the figure patch will be colored, if False, the figure background will be transparent. If not provided, the rcParam ‘savefig.frameon’ will be used.

**bbox_inches**: Bbox 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**: Amount of padding around the figure when bbox_inches is ‘tight’.

**bbox_extra_artists**: A list of extra artists that will be considered when the tight bbox is calculated.

```python
cmplib.pyplot.sca(ax)
```
Set the current Axes instance to ax.

The current Figure is updated to the parent of ax.

```python
cmplib.pyplot.scatter(x, y, s=20, c='b', marker='o', cmap=None, norm=None, vmin=None, vmax=None, alpha=None, linewidths=None, verts=None, hold=None, **kwargs)
```
Make a scatter plot of x vs y, where x and y are sequence like objects of the same lengths.

**Parameters**

- **x, y**: array_like, shape (n, )
  Input data
- **s**: scalar or array_like, shape (n, ), optional, default: 20
  size in points^2.
- **c**: color or sequence of color, optional, default
  c can be a single color format string, or a sequence of color specifications of length N, or a sequence of N numbers to be mapped to colors using the cmap and norm specified via kwargs (see below). 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. c can be a 2-D array in which the rows are RGB or RGBA, however.
- **marker**: MarkerStyle, optional, default: ‘o'
  See markers for more information on the different styles of markers scatter supports.
- **cmap**: Colormap, optional, default: None
  A Colormap instance or registered name. cmap is only used if c is an array of floats. If None, defaults to rc image.cmap.
- **norm**: Normalize, optional, default: None
  A Normalize instance is used to scale luminance data to 0, 1. norm is only used if c is an array of floats. If None, use the default normalize().
- **vmin, vmax**: scalar, optional, default: None
  vmin and vmax are used in conjunction with norm to normalize lumi-
nance data. If either are None, the min and max of the color array is used. Note if you pass a norm instance, your settings for vmin and vmax will be ignored.

**alpha** : scalar, optional, default: None
The alpha blending value, between 0 (transparent) and 1 (opaque)

**linewidths** : scalar or array_like, optional, default: None
If None, defaults to (lines.linewidth,). Note that this is a tuple, and if you set the linewidths argument you must set it as a sequence of floats, as required by RegularPolyCollection.

**Returns**

paths : PathCollection

**Other Parameters**

**kwargs** : Collection properties

**Notes**

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.

**Examples**

Additional kwargs: hold = [True]False] overrides default hold state
Matplotlib.pyplot.sci(im)
Set the current image. This image will be the target of colormap commands like jet(), hot() or clim()). The current image is an attribute of the current axes.

Matplotlib.pyplot.semilogx(*args, **kwargs)
Make a plot with log scaling on the x axis.

Call signature:

        semilogx(*args, **kwargs)

semilogx() supports all the keyword arguments of plot() and matplotlib.axes.Axes.set_xscale().

Notable keyword arguments:

        baseX: scalar > 1 Base of the x logarithm
        subsx: [ None | sequence ] The location of the minor xticks; None defaults to autosubs, which depend on the number of decades in the plot; see set_xscale() for details.
        nonposx: [ `mask` | `clip` ] Non-positive values in x can be masked as invalid, or clipped to a very small positive number

The remaining valid kwargs are Line2D properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>data</td>
<td>2D array (rows are x, y) or two 1D arrays</td>
</tr>
<tr>
<td>drawstyle</td>
<td>['default'</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>['full'</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['-.'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>marker</td>
<td>unknown</td>
</tr>
<tr>
<td>markeredgecolor or mec</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markeredgewidth or mew</td>
<td>float value in points</td>
</tr>
</tbody>
</table>
Table 70.22 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function ( fn(\text{artist, event}) )</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>[‘butt’</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>[‘miter’</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

See also:

loglog() For example code and figure

Additional kwargs: hold = [True|False] overrides default hold state

matplotlib.pyplot.semilogy(*args, **kwargs)

Make a plot with log scaling on the \( y \) axis.

call signature:

semilogy(*args, **kwargs)

semilogy() supports all the keyword arguments of plot() and matplotlib.axes.Axes.set_yscale().

Notable keyword arguments:

* basey: scalar > 1 Base of the \( y \) logarithm
* subsy: [None | sequence] The location of the minor yticks; None defaults to autosubs, which depend on the number of decades in the plot; see set_yscale() for details.
* nonposy: [‘mask’ | ‘clip’] Non-positive values in \( y \) can be masked as invalid, or clipped to a very small positive number

The remaining valid kwargs are Line2D properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>(Path, Transform)</td>
</tr>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>data</td>
<td>2D array (rows are x, y) or two 1D arrays</td>
</tr>
<tr>
<td>drawstyle</td>
<td>['default'</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>['full'</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['-.'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>marker</td>
<td>unknown</td>
</tr>
<tr>
<td>markeredgecolor or mec</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markeredgewidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>
See also:

*loglog()*. For example code and figure

Additional kwargs: hold = [True|False] overrides default hold state

```python
matplotlib.pyplot.set_cmap(cmap)
```

Set the default colormap. Applies to the current image if any. See help(colormaps) for more information.

`cmap` must be a *Colormap* instance, or the name of a registered colormap.

See *matplotlib.cm.register_cmap()* and *matplotlib.cm.get_cmap()*.

```python
matplotlib.pyplot.setp(*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:

```python
>>> 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:

```python
>>> setp(line, 'linestyle')
linestyle: [ '-' | '--' | '-.' | ':' | 'steps' | 'None' ]
```

If you want to see all the properties that can be set, and their possible values, you can do:

```python
>>> setp(line)
... long output listing omitted
```

`setp()` operates on a single instance or a list 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:

```python
>>> x = arange(0,1.0,0.01)
>>> 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:

```python
>>> setp(lines, 'linewidth', 2, 'color', 'r')  # MATLAB style
>>> setp(lines, linewidth=2, color='r')  # python style
```
Show a figure.

When running in ipython with its pylab mode, display all figures and return to the ipython prompt.

In non-interactive mode, display all figures and block until the figures have been closed; in interactive mode it has no effect unless figures were created prior to a change from non-interactive to interactive mode (not recommended). In that case it displays the figures but does not block.

A single experimental keyword argument, block, may be set to True or False to override the blocking behavior described above.

```
specgram(x, NFFT=256, Fs=2, Fc=0, detrend=<function detrend_none at 0x24627d0>, window=<function window_hanning at 0x2462578>, noverlap=128, cmap=None, xextent=None, pad_to=None, sides='default', scale_by_freq=None, hold=None, **kwargs)
```

Plot a spectrogram.

Call signature:

```
specgram(x, NFFT=256, Fs=2, Fc=0, detrend=mlab.detrend_none, window=mlab.window_hanning, noverlap=128, cmap=None, xextent=None, pad_to=None, sides='default', scale_by_freq=None, **kwargs)
```

Compute and plot a spectrogram of data in x. Data are split into NFFT length segments and the PSD 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 in decibels as a colormap (using imshow).

Keyword arguments:

- **NFFT**: integer The number of data points used in each block for the FFT. Must be even; a power 2 is most efficient. The default value is 256. This should NOT be used to get zero padding, or the scaling of the result will be incorrect. Use pad_to for this instead.
- **Fs**: scalar The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, freqs, in cycles per time unit. The default value is 2.
- **detrend**: callable 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 pylab module defines detrend_none(), detrend_mean(), and detrend_linear(), but you can use a custom function as well.
- **window**: callable or ndarray 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. The default is window_hanning(). 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.
- **pad_to**: integer 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 psd (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 `fit()`. The default is None, which sets `pad_to` equal to `NFFT`.

**sides**: [‘default’ | ‘onesided’ | ‘twosided’] Specifies which sides of the PSD to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. ‘onesided’ forces the return of a one-sided PSD, while ‘twosided’ forces two-sided.

**scale_by_freq**: boolean Specifies 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**: integer The number of points of overlap between blocks. The default value is 128.

**Fc**: integer The center frequency of $x$ (defaults to 0), which offsets the y extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

**cmap**: A `matplotlib.colors.Colormap` instance; if `None`, use default determined by `rc`.

**xextent**: The image extent along the x-axis. `xextent = (xmin, xmax)` The default is (0,max(bins)), where bins is the return value from `specgram()`.

**kwargs**: Additional kwargs are passed on to `imshow` which makes the specgram image.

Return value is `(Pxx, freqs, bins, im)`:

- `bins` are the time points the spectrogram is calculated over
- `freqs` is an array of frequencies
- `Pxx` is an array of shape `(len(times), len(freqs))` of power
- `im` is a `AxesImage` instance

**Note**: If $x$ is real (i.e. non-complex), only the positive spectrum is shown. If $x$ is complex, both positive and negative parts of the spectrum are shown. This can be overridden using the `sides` keyword argument.

Also note that while the plot is in dB, the `Pxx` array returned is linear in power.

**Example:**
Additional kwargs: hold = [True|False] overrides default hold state

```
matplotlib.pyplot.spectral()  # set the default colormap to spectral and apply to current image if any. See help(colormaps) for more information

matplotlib.pyplot.spring()  # set the default colormap to spring and apply to current image if any. See help(colormaps) for more information

matplotlib.pyplot.spy(Z, precision=0, marker=None, markersize=None, aspect='equal', hold=None, **kwargs)  # Plot the sparsity pattern on a 2-D array.

Call signature:

spy(Z, precision=0, marker=None, markersize=None, aspect='equal', **kwargs)

spy(Z) plots the sparsity pattern of the 2-D array Z.

If precision is 0, any non-zero value will be plotted; else, values of |Z| > precision will be plotted.

For scipy.sparse.spmatrix instances, there is a special case: if precision is ‘present’, any value present in the array will be plotted, even if it is identically zero.
The array will be plotted as it would be printed, with the first index (row) increasing down and the second index (column) increasing to the right.

By default aspect is ‘equal’, so that each array element occupies a square space; set the aspect kwarg to ‘auto’ to allow the plot to fill the plot box, or to any scalar number to specify the aspect ratio of an array element directly.

Two plotting styles are available: image or marker. Both are available for full arrays, but only the marker style works for scipy.sparse.spmatrix instances.

If marker and markersize are None, an image will be returned and any remaining kwargs are passed to imshow(); else, a Line2D object will be returned with the value of marker determining the marker type, and any remaining kwargs passed to the plot() method.

If marker and markersize are None, useful kwargs include:
• cmap
• alpha

See also:
imshow() For image options.

For controlling colors, e.g., cyan background and red marks, use:

cmap = mcolors.ListedColormap(['c', 'r'])

If marker or markersize is not None, useful kwargs include:
• marker
• markersize
• color

Useful values for marker include:
• 's' square (default)
• 'o' circle
• '.' point
• ',' pixel

See also:
plot() For plotting options

Additional kwargs: hold = [True|False] overrides default hold state

matplotlib.pyplot.stackplot(x, *args, **kwargs)

Draws a stacked area plot.

x : 1d array of dimension N
y [2d array of dimension MxN, OR any number 1d arrays each of 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

Keyword arguments:
baselines [[‘zero’, ‘sym’, ‘wiggle’, ‘weighted_wiggle’]] Method used to calculate the baseline. ‘zero’ is just a simple stacked plot. ‘sym’ is 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://www.leebyron.com/else/streamgraph/.

colors [A list or tuple of colors. These will be cycled through and] used to colour the
stacked areas. All other keyword arguments are passed to fill_between().

Returns r : A list of PolyCollection, one for each element in the stacked area plot.

Note that Legend does not support PolyCollection objects. To create a legend on a
stackplot, use a proxy artist: http://matplotlib.org/users/legend_guide.html#using-proxy-
artist

Additional kwargs: hold = [True|False] overrides default hold state

```python
matplotlib.pyplot.stem(*args, **kwargs)
```

Create a stem plot.

Call signatures:

```python
stem(y, linefmt='b-', markerfmt='bo', basefmt='r-')
stem(x, y, linefmt='b-', markerfmt='bo', basefmt='r-')
```

A stem plot plots vertical lines (using linefmt) at each x location from the baseline to y, and places a
marker there using markerfmt. A horizontal line at 0 is is plotted using basefmt.

If no x values are provided, the default is (0, 1, ..., len(y) - 1)

Return value is a tuple (markerline, stemlines, baseline).

See also:

This document for details.

Example:
Additional kwargs: hold = [True,False] overrides default hold state

```
import matplotlib.pyplot as plt
plt.step(x, y, *args, **kwargs)
```

Make a step plot.

Call signature:

```
step(x, y, *args, **kwargs)
```

Additional keyword args to `step()` are the same as those for `plot()`.

$x$ and $y$ must be 1-D sequences, and it is assumed, but not checked, that $x$ is uniformly increasing.

Keyword arguments:

- **where**: [‘pre’ | ‘post’ | ‘mid’] If ‘pre’, the interval from $x[i]$ to $x[i+1]$ has level $y[i+1]$
  - If ‘post’, that interval has level $y[i]$
  - If ‘mid’, the jumps in $y$ occur half-way between the $x$-values.

Additional kwargs: hold = [True,False] overrides default hold state

```
import matplotlib.pyplot as plt
plt.streamplot(x, y, u, v, density=1, linewidth=None, color=None,
               cmap=None, norm=None, arrowstyle='->', min_length=0.1, transform=None, hold=None)
```

Draws streamlines of a vector flow.

$x,y$ [1d arrays] an evenly spaced grid.
\( u, v \) [2d arrays] x and y-velocities. Number of rows should match length of \( y \), and the number of columns should match \( x \).

density [float or 2-tuple] Controls the closeness of streamlines. When density = 1, the domain is divided into a 25x25 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 [density_\( x \), density_\( y \)].

linewidth [numeric or 2d array] vary linewidth when given a 2d array with the same shape as velocities.

color [matplotlib color code, or 2d array] Streamline color. When given an array with the same shape as velocities, color values are converted to colors using cmap.

cmap [Colormap] Colormap used to plot streamlines and arrows. Only necessary when using an array input for color.

norm [Normalize] Normalize object used to scale luminance data to 0, 1. If None, stretch (min, max) to (0, 1). Only necessary when color is an array.

arrowsize [float] Factor scale arrow size.


Returns:

stream_container [StreamplotSet] Container object with attributes
  • lines: matplotlib.collections.LineCollection of streamlines
  • arrows: collection of matplotlib.patches.FancyArrowPatch objects representing 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.

Additional kwargs: hold = [True|False] overrides default hold state

matplotlib.pyplot.subplot(*args, **kwargs)

Return a subplot axes positioned by the given grid definition.

Typical call signature:

subplot(nrows, ncols, plot_number)

Where \( nrows \) and \( ncols \) are used to notionally split the figure into \( nrows \times ncols \) sub-axes, and \( plot_number \) is used to identify the particular subplot that this function is to create within the notional grid. \( plot_number \) starts at 1, increments across rows first and has a maximum of \( nrows \times ncols \).

In the case when \( nrows, ncols \) and \( plot_number \) are all less than 10, a convenience exists, such that the a 3 digit number can be given instead, where the hundreds represent \( nrows \), the tens represent \( ncols \) and the units represent \( plot_number \). For instance:

subplot(211)

produces a subaxes in a figure which represents the top plot (i.e. the first) in a 2 row by 1 column notional grid (no grid actually exists, but conceptually this is how the returned subplot has been positioned).
Note: Creating a new subplot with a position which is entirely inside a pre-existing axes will trigger the larger axes to be deleted:

```python
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)
plt.plot(range(12))
plt.subplot(212, axisbg='y') # creates 2nd subplot with yellow background
```

If you do not want this behavior, use the `add_subplot()` method or the `axes()` function instead.

Keyword arguments:

- `axisbg`: The background color of the subplot, which can be any valid color specifier. See `matplotlib.colors` for more information.
- `polar`: A boolean flag indicating whether the subplot plot should be a polar projection. Defaults to `False`.
- `projection`: A string giving the name of a custom projection to be used for the subplot. This projection must have been previously registered. See `matplotlib.projections`.

See also:

- `axes()` For additional information on `axes()` and `subplot()` keyword arguments.
- `examples/pie_and_polar_charts/polar_scatter_demo.py` For an example

Example:
A tale of 2 subplots

Damped oscillation

Undamped

time (s)

matplotlib.pyplot.subplot2grid(shape, loc, rowspan=1, colspan=1, **kwargs)

Create a subplot in a grid. The grid is specified by shape, at location of loc, spanning rowspan, colspan cells in each direction. The index for loc is 0-based.

subplot2grid(shape, loc, rowspan=1, colspan=1)

is identical to

gridspec=GridSpec(shape[0], shape[2])
subplotspec=gridspec.new_subplotspec(loc, rowspan, colspan)
 subplot(subplotspec)

matplotlib.pyplot.subplot_tool(targetfig=None)

Launch a subplot tool window for a figure.

A matplotlib.widgets.SubplotTool instance is returned.

matplotlib.pyplot.subplots(nrows=1, ncols=1, sharex=False, sharey=False, squeeze=True, subplot_kw=None, **fig_kw)

Create a figure with a set of subplots already made.

This utility wrapper makes it convenient to create common layouts of subplots, including the enclosing figure object, in a single call.

Keyword arguments:

nrows [int] Number of rows of the subplot grid. Defaults to 1.
ncols [int] Number of columns of the subplot grid. Defaults to 1.

sharex [string or bool] If True, the X axis will be shared amongst all subplots. If True and you have multiple rows, the x tick labels on all but the last row of plots will have visible set to False. If a string must be one of “row”, “col”, “all”, or “none”. “all” has the same effect as True, “none” has the same effect as False. If “row”, each subplot row will share a X axis. If “col”, each subplot column will share a X axis and the x tick labels on all but the last row will have visible set to False.

sharey [string or bool] If True, the Y axis will be shared amongst all subplots. If True and you have multiple columns, the y tick labels on all but the first column of plots will have visible set to False. If a string must be one of “row”, “col”, “all”, or “none”. “all” has the same effect as True, “none” has the same effect as False. If “row”, each subplot row will share a Y axis. If “col”, each subplot column will share a Y axis and the y tick labels on all but the last row will have visible set to False.

squeeze [bool] If True, extra dimensions are squeezed out from the returned axis object:
- if only one subplot is constructed (nrows=ncols=1), the resulting single Axis object is returned as a scalar.
- for Nx1 or 1xN subplots, the returned object is a 1-d numpy object array of Axis objects are returned as nump 1-d arrays.
- 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 axis object is always a 2-d array containing Axis instances, even if it ends up being 1x1.

subplot_kw [dict] Dict with keywords passed to the add_subplot() call used to create each subplots.

fig_kw [dict] Dict with keywords passed to the figure() call. Note that all keywords not recognized above will be automatically included here.

Returns:
fig, ax : tuple

- fig is the matplotlib.figure.Figure object
- ax can be either a single axis object or an array of axis objects if more than one subplot was created. The dimensions of the resulting array can be controlled with the squeeze keyword, see above.

Examples:

```python
x = np.linspace(0, 2*np.pi, 400)
y = np.sin(x**2)

# Just a figure and one subplot
f, ax = plt.subplots()
ax.plot(x, y)
ax.set_title('Simple plot')

# Two subplots, 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)

# Four polar axes
plt.subplots(2, 2, subplot_kw=dict(polar=True))
```
# 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 a X and Y axis with all subplots
plt.subplots(2, 2, sharex='all', sharey='all')
# same as
plt.subplots(2, 2, sharex=True, sharey=True)

matplotlib.pyplot.subplots_adjust(*args, **kwargs)
Tune the subplot layout.

call signature:

subplots_adjust(left=None, bottom=None, right=None, top=None,
                 wspace=None, hspace=None)

The parameter meanings (and suggested defaults) are:

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 blank space between subplots
hspace = 0.2  # the amount of height reserved for white space between subplots

The actual defaults are controlled by the rc file

matplotlib.pyplot.summer()
set the default colormap to summer and apply to current image if any. See help(colormaps) for more information

matplotlib.pyplot.suptitle(*args, **kwargs)
Add a centered title to the figure.

kwargs are matplotlib.text.Text properties. Using figure coordinates, the defaults are:
  x [0.5] The x location of the text in figure coords
  y [0.98] The y location of the text in figure coords
  horizontalalignment ['center'] The horizontal alignment of the text
  verticalalignment ['top'] The vertical alignment of the text
A matplotlib.text.Text instance is returned.

Example:

fig.suptitle('this is the figure title', fontsize=12)

matplotlib.pyplot.switch_backend(newbackend)
Switch the default backend. This feature is experimental, and is only expected to work switching to an image backend. e.g., if you have a bunch of PostScript scripts that you want to run from an interactive ipython session, you may want to switch to the PS backend before running them to avoid
having a bunch of GUI windows popup. If you try to interactively switch from one GUI backend to another, you will explode.

Calling this command will close all open windows.

```
import matplotlib.pyplot as plt
plt.table(kwargs)
```

Add a table to the current axes.

Call signature:

```
plt.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):
```

Returns a `matplotlib.table.Table` instance. For finer grained control over tables, use the `Table` class and add it to the axes with `add_table()`.

Thanks to John Gill for providing the class and table.

`kwargs` control the `Table` properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an <code>Axes</code> instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>figure</td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td>fontsize</td>
<td>a float in points</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td><code>Transform</code> instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

```
plt.text(x, y, s, fontdict=None, withdash=False, **kwargs)
```

Add text to the axes.

Add text in string `s` to axis at location `x`, `y`, data coordinates.

**Parameters**

- `s`: string
text

x, y : scalars
data coordinates

fontdict : dictionary, optional, default: None
A dictionary to override the default text properties. If fontdict is None, the defaults are determined by your rc parameters.

withdash : boolean, optional, default: False
 Creates a TextWithDash instance instead of a Text instance.

Other Parameters kwargs : Text properties.
Other miscellaneous text parameters.

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))

matplotlib.pyplot.thetagrids(*args, **kwargs)
Get or set the theta locations of the gridlines in a polar plot.

If no arguments are passed, return a tuple (lines, labels) where lines is an array of radial gridlines (Line2D instances) and labels is an array of tick labels (Text instances):

lines, labels = theagrids()

Otherwise the syntax is:

lines, labels = theagrids(angles, labels=None, fmt='%d', frac = 1.1)

set the angles at which to place the theta grids (these gridlines are equal along the theta dimension). angles is in degrees.

labels, if not None, is a len(angles) list of strings of the labels to use at each angle.

If labels is None, the labels will be fmt%angle.
frac is the fraction of the polar axes radius at which to place the label (1 is the edge). e.g., 1.05 is outside the axes and 0.95 is inside the axes.

Return value is a list of tuples (lines, labels):
- **lines** are Line2D instances
- **labels** are Text instances.

Note that on input, the labels argument is a list of strings, and on output it is a list of Text instances.

Examples:

```python
# set the locations of the radial gridlines and labels
lines, labels = thetagrids( range(45,360,90) )

# set the locations and labels of the radial gridlines and labels
lines, labels = thetagrids( range(45,360,90), ('NE', 'NW', 'SW', 'SE') )
```

matplotlib.pyplot.tick_params(axis='both', **kwargs)

Change the appearance of ticks and tick labels.

Keyword arguments:
- **axis** [‘x’ | ‘y’ | ‘both’] Axis on which to operate; default is ‘both’.
- **reset** [True | False] If True, set all parameters to defaults before processing other keyword arguments. Default is False.
- **which** [‘major’ | ‘minor’ | ‘both’] Default is ‘major’; apply arguments to which ticks.
- **direction** [‘in’ | ‘out’ | ‘inout’] Puts ticks inside the axes, outside the axes, or both.
- **length** Tick length in points.
- **width** Tick width in points.
- **color** Tick color; accepts any mpl color spec.
- **pad** Distance in points between tick and label.
- **labelsize** Tick label font size in points or as a string (e.g., ‘large’).
- **labelcolor** Tick label color; mpl color spec.
- **colors** Changes the tick color and the label color to the same value: mpl color spec.
- **zorder** Tick and label zorder.
- **bottom, top, left, right** [bool | ‘on’ | ‘off’] controls whether to draw the respective ticks.
- **labelbottom, labeltop, labelleft, labelright** Boolean or [‘on’ | ‘off’], controls whether to draw the respective tick labels.

Example:

```python
ax.tick_params(direction='out', length=6, width=2, colors='r')
```

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.

matplotlib.pyplot.ticklabel_format(**kwargs)

Change the ScalarFormatter used by default for linear axes.

Optional keyword arguments:
### Key-Word Description

<table>
<thead>
<tr>
<th>Key-word</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>style</strong></td>
<td>[ 'sci' (or 'scientific')</td>
</tr>
<tr>
<td><strong>scilimits</strong></td>
<td>(m, n), pair of integers; if style is 'sci', scientific notation will be used for numbers outside the range $10^m$ to $10^n$. Use (0,0) to include all numbers.</td>
</tr>
<tr>
<td><strong>useOffset</strong></td>
<td>[True</td>
</tr>
<tr>
<td><strong>axis</strong></td>
<td>[ 'x'</td>
</tr>
<tr>
<td><strong>useLocale</strong></td>
<td>If True, format the number according to the current locale. This affects things such as the character used for the decimal separator. If False, use C-style (English) formatting. The default setting is controlled by the axes.formatter.use_locale rcparam.</td>
</tr>
</tbody>
</table>

Only the major ticks are affected. If the method is called when the `ScalarFormatter` is not the `Formatter` being used, an `AttributeError` will be raised.

**matplotlib.pyplot.tight_layout**(pad=1.08, h_pad=None, w_pad=None, rect=None)

Automatically 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] padding (height/width) between edges of adjacent subplots. Defaults to `pad_inches`.
- **rect** [if rect is given, it is interpreted as a rectangle] (left, bottom, right, top) in the normalized figure coordinate that the whole subplots area (including labels) will fit into. Default is (0, 0, 1, 1).

**matplotlib.pyplot.title**(s, *args, **kwargs)

Set a title of the current 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'],
  'verticalalignment': 'baseline', 'horizontalalignment':
  'center', 'left', 'right'}
- **loc** : str, optional
  Which title to set, defaults to 'center'

Returns **text** : `Text`

The matplotlib text instance representing the title

Other Parameters Other keyword arguments are text properties, see:

:class:`matplotlib.text.Text` for a list of valid text properties.

See also:

See `func:text` for adding text to the current axes
Matplotlib, Release 1.3.0

```
matplotlib.pyplot.tricontour(*args, **kwargs)
```

Draw contours on an unstructured triangular grid. `tricontour()` and `tricontourf()` draw contour lines and filled contours, respectively. Except as noted, function signatures and return values are the same for both versions.

The triangulation can be specified in one of two ways; either:

```
tricontour(triangulation, ...)
```

where triangulation is a `matplotlib.tri.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 `Triangulation` for a explanation of these possibilities.

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, N)
```

contour `N` automatically-chosen levels.

```
tricontour(..., Z, V)
```

draw contour lines at the values specified in sequence `V`

```
tricontourf(..., Z, V)
```

fill the `(len(V)-1)` regions between the values in `V`

```
tricontour(Z, **kwargs)
```

Use keyword args to control colors, linewidth, origin, cmap ... see below for more details.

```
C = tricontour(...) returns a TriContourSet object.
```

Optional keyword arguments:

```
  colors: [ None | string | (mpl_colors) ] If None, the colormap specified by cmap will be used.
```

If a string, like ‘r’ or ‘red’, all levels will be plotted in this color.
If a tuple of matplotlib color args (string, float, rgb, etc), different levels will be plotted in different colors in the order specified.

**alpha**: float  
The alpha blending value

**cmap**: [None | Colormap]  
A cm Colormap instance or None. If cmap is None and colors is None, a default Colormap is used.

**norm**: [None | Normalize]  
A matplotlib.colors.Normalize instance for scaling data values to colors. If norm is None and colors is None, the default linear scaling is used.

**levels** [level0, level1, ..., leveln]  
A list of floating point numbers indicating the level curves to draw; eg to draw just the zero contour pass levels=[0]

**origin**: [None | ‘upper’ | ‘lower’ | ‘image’]  
If None, the first value of Z will correspond to the lower left corner, location (0,0). If ‘image’, the rc value for image.origin will be used.

This keyword is not active if X and Y are specified in the call to contour.

**extent**: [None | (x0,x1,y0,y1)]  
If origin is not None, then extent is interpreted as in matplotlib.pyplot.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 keyword is not active if X and Y are specified in the call to contour.

**locator**: [None | ticker.Locator subclass]  
If locator is None, the default MaxNLocator is used. The locator is used to determine the contour levels if they are not given explicitly via the V argument.

**extend**: [‘neither’ | ‘both’ | ‘min’ | ‘max’]  
Unless this is ‘neither’, contour levels are automatically added to one or both ends of the range so that all data are included. These added ranges are then mapped to the special colormap values which default to the ends of the colormap range, but can be set via matplotlib.colors.Colormap.set_under() and matplotlib.colors.Colormap.set_over() methods.

**xunits, yunits**: [None | registered units]  
Override axis units by specifying an instance of a matplotlib.units.ConversionInterface.

tricontour-only keyword arguments:

**linestyles**: [None | number | tuple of numbers]  
If linestyles is None, the default width in lines.linewidth in matplotlibrc is used. If a number, all levels will be plotted with this linewidth.

If a tuple, different levels will be plotted with different linewidths in the order specified.

**linestyles**: [None | ‘solid’ | ‘dashed’ | ‘dashdot’ | ‘dotted’]  
If linestyles is None, the ‘solid’ is used.

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.

If contour is using a monochrome colormap and the contour level is less than 0,
then the linestyle specified in `contour.negative_linestyle` in `matplotlibrc` will be used.

tricontourf-only keyword arguments:

- `antialiased`: [True | False] enable antialiasing
- `nchunk`: [0 | integer] If 0, no subdivision of the domain. Specify a positive integer to divide the domain into subdomains of roughly `nchunk` by `nchunk` points. This may never actually be advantageous, so this option may be removed. Chunking introduces artifacts at the chunk boundaries unless `antialiased` is `False`.

Note: tricontourf fills intervals that are closed at the top; that is, for boundaries `z1` and `z2`, the filled region is:

\[ z1 < z \leq z2 \]

There is one exception: if the lowest boundary coincides with the minimum value of the `z` array, then that minimum value will be included in the lowest interval.

Examples:
The triangulation can be specified in one of two ways; either:

```python
tricontour(triangulation, ...)
```

where triangulation is a `matplotlib.tri.Triangulation` object, or

```python
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 `Triangulation` for a explanation of these possibilities.

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, N)

t contour N automatically-chosen levels.

tricontour(..., Z, V)

draw contour lines at the values specified in sequence V

tricontourf(..., Z, V)

fill the (len(V)-1) regions between the values in V

tricontour(Z, **kwargs)

Use keyword args to control colors, linewidth, origin, cmap ... see below for more details.

C = tricontour(...) returns a TriContourSet object.

Optional keyword arguments:

  colors: [ None | string | (mpl_colors) ] If None, the colormap specified by cmap will be used.
     If a string, like ‘r’ or ‘red’, all levels will be plotted in this color.
     If a tuple of matplotlib color args (string, float, rgb, etc), different levels will be plotted in different colors in the order specified.

  alpha: float The alpha blending value

  cmap: [ None | Colormap ] A cm Colormap instance or None. If cmap is None and colors is None, a default Colormap is used.

  norm: [ None | Normalize ] A matplotlib.colors.Normalize instance for scaling data values to colors. If norm is None and colors is None, the default linear scaling is used.

  levels [level0, level1, ..., leveln] A list of floating point numbers indicating the level curves to draw; eg to draw just the zero contour pass levels=[0]

  origin: [ None | ‘upper’ | ‘lower’ | ‘image’ ] If None, the first value of Z will correspond to the lower left corner, location (0,0). If ‘image’, the rc value for image.origin will be used.
     This keyword is not active if X and Y are specified in the call to contour.

  extent: [ None | (x0,x1,y0,y1) ]
     If origin is not None, then extent is interpreted as in matplotlib.pyplot.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 keyword is not active if \(X\) and \(Y\) are specified in the call to `contour`.

**locator**: [ `None` | `ticker.Locator subclass` ] If `locator` is `None`, the default \texttt{MaxNLocator} is used. The locator is used to determine the contour levels if they are not given explicitly via the \texttt{V} argument.

**extend**: [ `‘neither’` | `‘both’` | `‘min’` | `‘max’` ] Unless this is `‘neither’`, contour levels are automatically added to one or both ends of the range so that all data are included. These added ranges are then mapped to the special colormap values which default to the ends of the colormap range, but can be set via \texttt{matplotlib.colors.Colormap.set_under()} and \texttt{matplotlib.colors.Colormap.set_over()} methods.

**xunits**, **yunits**: [ `None` | `registered units` ] Override axis units by specifying an instance of a \texttt{matplotlib.units.ConversionInterface}.

For `tricontour` only:

**linestyles**: [ `None` | `number` | `tuple of numbers` ] If `linestyles` is `None`, the default \texttt{linestyle} in \texttt{lines.linewidth} in \texttt{matplotlibrc} is used.

If a number, all levels will be plotted with this linestyle.

If a tuple, different levels will be plotted with different linestyles in the order specified.

**linewidths**: [ `None` | `number` | `tuple of numbers` ] If `linewidths` is `None`, the default width in \texttt{lines.linewidth} in \texttt{matplotlibrc} is used.

If a number, all levels will be plotted with this linewidth.

If a tuple, different levels will be plotted with different linewidths in the order specified.

For `tricontourf` only:

**antialiased**: [ `True` | `False` ] enable antialiasing

**nchunk**: [ `0` | `integer` ] If \(0\), no subdivision of the domain. Specify a positive integer to divide the domain into subdomains of roughly \(nchunk\) by \(nchunk\) points. This may never actually be advantageous, so this option may be removed. Chunking introduces artifacts at the chunk boundaries unless \texttt{antialiased} is `False`.

Note: `tricontourf` fills intervals that are closed at the top; that is, for boundaries \(z1\) and \(z2\), the filled region is:

\[ z1 < z \leq z2 \]

There is one exception: if the lowest boundary coincides with the minimum value of the \(z\) array, then that minimum value will be included in the lowest interval.

**Examples:**
Additional kwargs: hold = [True|False] overrides default hold state

```
matplotlib.pyplot.tripcolor(*args, **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 `matplotlib.tri.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`. 

70.1. `matplotlib.pyplot`
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. shading of ‘faceted’ is deprecated; please use edgecolors instead.

The remaining kwargs are the same as for pcolor().

Example:

tripcolor of Delaunay triangulation, flat shading
pcolor of Delaunay triangulation, gouraud shading

tripcolor of user-specified triangulation
Additional kwargs: hold = [True|False] overrides default hold state

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 matplotlib.tri.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().

Example:

![triplot of Delaunay triangulation](image)
Additional kwargs: hold = [True|False] overrides default hold state

```
matplotlib.pyplot.twinx(ax=None)
```
Make a second axes that shares the x-axis. The new axes will overlay ax (or the current axes if ax is None). The ticks for ax2 will be placed on the right, and the ax2 instance is returned.

See also:

```
exmaples/api_examples/two_scales.py
```
For an example

```
matplotlib.pyplot.twiny(ax=None)
```
Make a second axes that shares the y-axis. The new axis will overlay ax (or the current axes if ax is None). The ticks for ax2 will be placed on the top, and the ax2 instance is returned.

```
matplotlib.pyplot.vlines(x, ymin, ymax, colors='k', linestyles='solid', label='', hold=None, **kwargs)
```
Plot vertical lines.

Plot vertical lines at each x from ymin to ymax.

**Parameters**

- `x`: scalar or 1D array_like
  x-indexes where to plot the lines.
- `xmin, xmax`: scalar or 1D array_like
  Respective beginning and end of each line. If scalars are provided, all lines will have same length.
- `colors`: array_like of colors, optional, default: ‘k’
linestyles : ['solid' | 'dashed' | 'dashdot' | 'dotted'], optional

label : string, optional, default: ‘’

Returns lines : LineCollection

Other Parameters kwargs : LineCollection properties.

See also:

hlines horizontal lines

Examples

Additional kwargs: hold = [True]False overrides default hold state

matplotlib.pyplot.waitforbuttonpress(*args, **kwargs)

Call signature:

waitforbuttonpress(self, timeout=-1)

Blocking call to interact with the figure.

This will return True is a key was pressed, False if a mouse button was pressed and None if timeout
was reached without either being pressed.

If timeout is negative, does not timeout.

matplotlib.pyplot.winter()

set the default colormap to winter and apply to current image if any. See help(colormaps) for more
information

matplotlib.pyplot.xcorr(x, y, normed=True, detrend=<function detrend_none at 0x24627d0>,
usevlines=True, maxlags=10, hold=None, **kwargs)

Plot the cross correlation between x and y.

Call signature:
**xcorr**

```
xcorr(self, x, y, normed=True, detrend=mlab.detrend_none, usevlines=True, maxlags=10, **kwargs)
```

If *normed* = *True*, normalize the data by the cross correlation at 0-th lag. *x* and *y* are detrended by the *detrend* callable (default no normalization). *x* and *y* must be equal length.

Data are plotted as `plot(lags, c, **kwargs)`

Return value is a tuple (*lags*, *c*, *line*) where:
- *lags* are a length 2*maxlags+1 lag vector
- *c* is the 2*maxlags+1 auto correlation vector
- **line** is a **Line2D** instance returned by `plot()`.

The default linestyle is *None* and the default marker is ‘o’, though these can be overridden with keyword args. The cross correlation is performed with *numpy.correlate()* with *mode* = 2.

If *usevlines* is *True*:

- `vlines()` rather than `plot()` is used to draw vertical lines from the origin to the xcorr.
- Otherwise the plotstyle is determined by the kwargs, which are **Line2D** properties.

The return value is a tuple (*lags*, *c*, *linecol*, *b*) where *linecol* is the **matplotlib.collections.LineCollection** instance and *b* is the x-axis.

*maxlags* is a positive integer detailing the number of lags to show. The default value of *None* will return all (2*len(x)-1) lags.

**Example:**

*xcorr()* is top graph, and *acorr()* is bottom graph.
Additional kwargs: hold = [True|False] overrides default hold state

matplotlib.pyplot.xkcd(scale=1, length=100, randomness=2)

Turns 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.

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()

matplotlib.pyplot.xlabel(s, *args, **kwargs)
Set the $x$ axis label of the current axis.

Default override is:

```python
code
override = {
    'fontsize' : 'small',
    'verticalalignment' : 'top',
    'horizontalalignment' : 'center'
}
```

See also:

- `text()` For information on how `override` and the optional args work

```python
Example
```

```python
matplotlib.pyplot.xlimit(*args, **kwargs)
Get or set the $x$ limits of the current axes.
```n
```python
code
xmin, xmax = xlim()  # return the current xlim
xlim((xmin, xmax))  # set the xlim to xmin, xmax
xlim(xmin, xmax)  # set the xlim to xmin, xmax
```

If you do not specify args, you can pass the `xmin` and `xmax` as `kwargs`, e.g.:

```python
code
xlim(xmax=3)  # adjust the max leaving min unchanged
xlim(xmin=1)  # adjust the min leaving max unchanged
```

Setting limits turns autoscaling off for the $x$-axis.

The new axis limits are returned as a length 2 tuple.

```python
matplotlib.pyplot.xscale(*args, **kwargs)
Set the scaling of the $x$-axis.
```

```python
code
xscale(scale, **kwargs)
```

The available scales are: ‘linear’ | ‘log’ | ‘symlog’

Different keywords may be accepted, depending on the scale:

- ‘linear’
  - `baseX/baseY`: The base of the logarithm
  - `nonposX/nonposY`: ['mask' | 'clip'] non-positive values in $x$ or $y$ can be masked as invalid, or clipped to a very small positive number
  - `subsX/subsY`: Where to place the subticks between each major tick. Should be a sequence of integers. 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.

- ‘symlog’
basex/basey: The base of the logarithm
linthreshx/linthreshy: The range \((-x, x)\) within which the plot is linear (to avoid having the plot go to infinity around zero).
subsx/subsy: Where to place the subticks between each major tick. Should be a sequence of integers. 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.
linscalex/linscaley: This allows the linear range (\(-\text{linthresh}\) to \(\text{linthresh}\)) 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 \(\text{linscale} == 1.0\) (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.

```
matplotlib.pyplot.xticks(*args, **kwargs)
Get or set the x-limits of the current tick locations and labels.

# return locs, labels where locs is an array of tick locations and
# labels is an array of tick labels.
locs, labels = xticks()

# set the locations of the xticks
xticks( arange(6) )

# set the locations and labels of the xticks
xticks( arange(5), ('Tom', 'Dick', 'Harry', 'Sally', 'Sue') )
```

The keyword args, if any, are Text properties. For example, to rotate long labels:

```
xticks( arange(12), calendar.month_name[1:13], rotation=17 )
```

```
matplotlib.pyplot.ylabel(s, *args, **kwargs)
Set the y axis label of the current axis.

Defaults override is:

```
override = {
    'fontsize' : 'small',
    'verticalalignment' : 'center',
    'horizontalalignment' : 'right',
    'rotation'='vertical' : }
```

See also:

```
text() For information on how override and the optional args work.
```

```
matplotlib.pyplot.ylim(*args, **kwargs)
Get or set the y-limits of the current axes.
```
ymin, ymax = ylim()  # return the current ylim
ylim( (ymin, ymax) )  # set the ylim to ymin, ymax
ylim( ymin, ymax )  # set the ylim to ymin, ymax

If you do not specify args, you can pass the ymin and ymax as kwargs, e.g.:

ylim(ymax=3)    # adjust the max leaving min unchanged
ylim(ymin=1)    # adjust the min leaving max unchanged

Setting limits turns autoscaling off for the y-axis.

The new axis limits are returned as a length 2 tuple.

```
matplotlib.pyplot.yscale(*args, **kwargs)
```

Set the scaling of the y-axis.

call signature:

```
yscale(scale, **kwargs)
```

The available scales are: ‘linear’ | ‘log’ | ‘symlog’

Different keywords may be accepted, depending on the scale:

‘linear’
   ‘log’
   ‘base/baser’: The base of the logarithm
   ‘nonposx/nonposy’: [‘mask’ | ‘clip’] non-positive values in x or y can be
   masked as invalid, or clipped to a very small positive number
   ‘subsx/subsy’ Where to place the subticks between each major tick. Should
   be a sequence of integers. 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.

‘symlog’
   ‘base/baser’: The base of the logarithm
   ‘linthreshx/linthreshy’ The range (-x, x) within which the plot is linear (to
   avoid having the plot go to infinity around zero).
   ‘subsx/subsy’ Where to place the subticks between each major tick. Should
   be a sequence of integers. 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/linscaley’ This allows the linear range (-linthresh to 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.
```python
matplotlib.pyplot.yticks(*args, **kwargs)
Get or set the y-limits of the current tick locations and labels.

    # return locs, labels where locs is an array of tick locations and
    # labels is an array of tick labels.
    locs, labels = yticks()

    # set the locations of the yticks
    yticks( arange(6) )

    # set the locations and labels of the yticks
    yticks( arange(5), ('Tom', 'Dick', 'Harry', 'Sally', 'Sue') )

    The keyword args, if any, are Text properties. For example, to rotate long labels:

    yticks( arange(12), calendar.month_name[1:13], rotation=45 )
```

71.1 matplotlib.sankey

Module for creating Sankey diagrams using matplotlib

```python
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)
```

Sankey diagram in matplotlib

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.

Optional keyword arguments:
### Field | Description
---|---
**ax** | axes onto which the data should be plotted If **ax** isn’t provided, new axes will be created.

**scale** | 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** | string representing the physical unit associated with the flow quantities If **unit** is None, then none of the quantities are labeled.

**format** | 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** | space between paths that break in/break away to/from the top or bottom

**radius** | inner radius of the vertical paths

**shoulder** | size of the shoulders of output arrows

**offset** | text offset (from the dip or tip of the arrow)

**head_angle** | angle of the arrow heads (and negative of the angle of the tails) [deg]

**margin** | minimum space between Sankey outlines and the edge of the plot area

**tolerance** | acceptable maximum of the magnitude of the sum of flows The magnitude of the sum of connected flows cannot be greater than **tolerance**.

The optional arguments listed above are applied to all subdiagrams so that there is consistent alignment and formatting.

If **Sankey** is instantiated with any keyword arguments other than those explicitly listed above (**kwargs), they will be passed to **add()**, which will create the first subdiagram.

In order to draw a complex Sankey diagram, create an instance of **Sankey** by calling it without any kwargs:

```python
sankey = Sankey()
```

Then add simple Sankey sub-diagrams:

```python
sankey.add() # 1
sankey.add() # 2
#...
sankey.add() # n
```

Finally, create the full diagram:

```python
sankey.finish()
```

Or, instead, simply daisy-chain those calls:

```python
Sankey().add().add... .add().finish()
```
See also:

add() finish()

Examples:

The default settings produce a diagram like this.
add(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.

Return value is the instance of Sankey.

Optional keyword arguments:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>patchlabel</td>
<td>label to be placed at the center of the diagram Note: label (not patchlabel) will be passed to the patch through **kwargs and can be used to create an entry in the legend.</td>
</tr>
<tr>
<td>flows</td>
<td>array of flow values By convention, inputs are positive and outputs are negative.</td>
</tr>
<tr>
<td>orientations</td>
<td>list of orientations of the paths Valid values are 1 (from/to the top), 0 (from/to the left or right), or -1 (from/to the bottom). If orientations == 0, inputs will break in from the left and outputs will break away to the right.</td>
</tr>
<tr>
<td>labels</td>
<td>list of specifications of the labels for the flows Each value may be None (no labels), '' (just label the quantities), or a labeling string. If a single value is provided, it will be applied to all flows. If an entry is a non-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.</td>
</tr>
<tr>
<td>trunklength</td>
<td>length between the bases of the input and output groups</td>
</tr>
<tr>
<td>pathlengths</td>
<td>list of lengths of the 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.</td>
</tr>
<tr>
<td>prior</td>
<td>index of the prior diagram to which this diagram should be connected</td>
</tr>
<tr>
<td>connect</td>
<td>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.</td>
</tr>
<tr>
<td>rotation</td>
<td>angle of rotation of the diagram [deg] rotation is ignored if this diagram is connected to an existing one (using prior and connect). 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).</td>
</tr>
</tbody>
</table>

Valid kwargs are matplotlib.patches.PathPatch() arguments:
As examples, `fill=False` and `label='A legend entry'`. By default, `facecolor='#bfd1d4'` (light blue) and `linewidth=0.5`.

The indexing parameters (`prior` and `connect`) are zero-based.

The flows are placed along the top of the diagram from the inside out in order of their index within the `flows` list or array. They are placed along the sides of the diagram from the top down and along the bottom from the outside in.

If the 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.

**See also:**

`finish()`

**finish()**

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:
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>patch</td>
<td>Sankey outline (an instance of PathPatch)</td>
</tr>
<tr>
<td>flows</td>
<td>values of the flows (positive for input, negative for output)</td>
</tr>
<tr>
<td>angles</td>
<td>list of angles of the arrows [deg/90] For example, if the diagram has not</td>
</tr>
<tr>
<td></td>
<td>been rotated, an input to the top side will have an angle of 3 (DOWN), and</td>
</tr>
<tr>
<td></td>
<td>an output from the top side will have an angle of 1 (UP). If a flow has</td>
</tr>
<tr>
<td></td>
<td>been skipped (because its magnitude is less than tolerance), then its angle</td>
</tr>
<tr>
<td></td>
<td>will be None.</td>
</tr>
<tr>
<td>tips</td>
<td>array in which each row is an [x, y] pair indicating the positions of the</td>
</tr>
<tr>
<td></td>
<td>tips (or “dips”) of the flow paths If the magnitude of a flow is less the</td>
</tr>
<tr>
<td></td>
<td>tolerance for the instance of Sankey, the flow is skipped and its tip will</td>
</tr>
<tr>
<td></td>
<td>be at the center of the diagram.</td>
</tr>
<tr>
<td>text</td>
<td>Text instance for the label of the diagram</td>
</tr>
<tr>
<td>texts</td>
<td>list of Text instances for the labels of flows</td>
</tr>
</tbody>
</table>

See also:

add()
72.1 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 function: set_position for more information.

The default position is ('outward', 0).

Spines are subclasses of class: Patch, and inherit much of their behavior.

Spines draw a line or a circle, depending if function: set_patch_line or func-
tion: set_patch_circle has been called. Line-like is the default.
    • axes: the Axes instance containing the spine
    • spine_type: a string specifying the spine type
    • path: the path instance used to draw the spine

Valid kwargs are:
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or ec</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>facecolor or fc</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[‘/’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>[‘solid’</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float or None for default</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

classmethod **circular_spine**(axes, center, radius, **kwargs)

(staticmethod) Returns a circular Spine.

cla()

Clear the current spine

draw**(artist, renderer, *args, **kwargs)**

get_bounds()

Get the bounds of the spine.

get_patch_transform()

get_path()

get_position()

get the spine position
get_smart_bounds()
get whether the spine has smart bounds

get_spine_transform()
get the spine transform

is_frame_like()
return True if directly on axes frame

This is useful for determining if a spine is the edge of an old style MPL plot. If so, this function will return True.

classmethod linear_spine(axes, spine_type, **kwargs)
(staticmethod) Returns a linear Spine.

register_axis(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(low, high)
Set the bounds of the spine.

set_color(c)
Set the edgecolor.

ACCEPTS: matplotlib color arg or sequence of rgba tuples

See also:
set_facecolor(), set_edgecolor() For setting the edge or face color individually.

set_patch_circle(center, radius)
set the spine to be circular

set_patch_line()
set the spine to be linear

set_position(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 specify placing the spine inward.)
• ‘axes’ : place the spine at the specified Axes coordinate (from 0.0-1.0).
• ‘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(value)
set the spine and associated axis to have smart bounds
73.1 matplotlib.ticker

73.1.1 Tick locating and formatting

This module contains classes to support completely configurable tick locating and formatting. 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. Generic tick locators and formatters are provided, as well as domain specific custom ones.

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. You initialize this with a base, eg 10, and it picks axis limits and ticks that are multiples of your base.

The Locator subclasses defined here are

NullLocator  
No ticks

FixedLocator  
Tick locations are fixed

IndexLocator  
locator for index plots (e.g., where \(x = \text{range}(\text{len}(y))\))

LinearLocator  
evenly spaced ticks from min to max

LogLocator  
logarithmically ticks from min to max

MultipleLocator  

ticks and range are a multiple of base;  
either integer or float

OldAutoLocator  
choose a MultipleLocator and dynamically reassign it for intelligent ticking during navigation

MaxNLocator  
finds up to a max number of ticks at nice locations

AutoLocator  
MaxNLocator with simple defaults. This is the default tick locator for most plotting.
**AutoMinorLocator** locator for minor ticks when the axis is linear and the major ticks are uniformly spaced. It 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:

```python
ax.xaxis.set_major_locator( xmajorLocator )
ax.xaxis.set_minor_locator( xminorLocator )
ax.yaxis.set_major_locator( ymajorLocator )
ax.yaxis.set_minor_locator( yminorLocator )
```

The default minor locator is the NullLocator, eg 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

**FormatStrFormatter** use a sprintf format string

**ScalarFormatter** default formatter for scalars; autopick the fmt string

**LogFormatter** formatter for log axes

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:

```python
ax.xaxis.set_major_formatter( xmajorFormatter )
ax.xaxis.set_minor_formatter( xminorFormatter )
ax.yaxis.set_major_formatter( ymajorFormatter )
ax.yaxis.set_minor_formatter( yminorFormatter )
```

See *pylab_examples-major_minor_demo1* for an example of setting major an minor ticks. See the *matplotlib.dates* module for more information and examples of using date locators and formatters.
class matplotlib.ticker.TickHelper
    Bases: object

    axis = None

create_dummy_axis(**kwargs)

set_axis(axis)

set_bounds(vmin, vmax)

set_data_interval(vmin, vmax)

set_view_interval(vmin, vmax)

class matplotlib.ticker.Formatter
    Bases: matplotlib.ticker.TickHelper

    Convert the tick location to a string

    fix_minus(s)

        some classes may want to replace a hyphen for minus with the proper unicode symbol as described here <http://sourceforge.net/tracker/index.php?func=detail&aid=1962574&group_id=80706&atid=560720>__. The default is to do nothing

        Note, if you use this method, e.g., in format_data() or call, you probably don’t want to use it for format_data_short() since the toolbar uses this for interactive coord reporting and I doubt we can expect GUIs across platforms will handle the unicode correctly. So for now the classes that override fix_minus() should have an explicit format_data_short() method

    format_data(value)

    format_data_short(value)

        return a short string version

    get_offset()

    locs = []

set_locs(locs)
class matplotlib.ticker.FixedFormatter(seq)
    Bases: matplotlib.ticker.Formatter

    Return fixed strings for tick labels

    seq is a sequence of strings. For positions \( i < \text{len(seq)} \) return \( \text{seq}[i] \) regardless of \( x \). Otherwise return “”

    get_offset()

    set_offset_string(ofs)

class matplotlib.ticker.NullFormatter
    Bases: matplotlib.ticker.Formatter

    Always return the empty string

class matplotlib.ticker.FuncFormatter(func)
    Bases: matplotlib.ticker.Formatter

    User defined function for formatting

class matplotlib.ticker.FormatStrFormatter(fmt)
    Bases: matplotlib.ticker.Formatter

    Use a format string to format the tick

class matplotlib.ticker.ScalarFormatter(useOffset=True, useMathText=None, useLocale=None)
    Bases: matplotlib.ticker.Formatter

    Tick location is a plain old number. If useOffset==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. Scientific notation is used for data \( < 10^{-n} \) or data \( \geq 10^{m} \), where \( n \) and \( m \) are the power limits set using set_powerlimits((n,m)). The defaults for these are controlled by the axes.formatter.limits rc parameter.

    fix_minus(s)
        use a unicode minus rather than hyphen

    format_data(value)
        return a formatted string representation of a number

    format_data_short(value)
        return a short formatted string representation of a number

    get_offset()
        Return scientific notation, plus offset

    get_useLocale()

    get_useOffset()
pprint_val(x)

set_locs(locs)
    set the locations of the ticks

set_powerlimits(lims)
    Sets size thresholds for scientific notation.
    e.g., formatter.set_powerlimits((-3, 4)) sets the pre-2007 default in which scientific notation is used for numbers less than 1e-3 or greater than 1e4. See also set_scientific().

set_scientific(b)
    True or False to turn scientific notation on or off see also set_powerlimits()

set_useLocale(val)

set_useOffset(val)

useLocale

useOffset

class matplotlib.ticker.LogFormatter(base=10.0, labelOnlyBase=True)
    Bases: matplotlib.ticker.Formatter
    Format values for log axis;
    base is used to locate the decade tick, which will be the only one to be labeled if labelOnlyBase is False
    base(base)
        change the base for labeling - warning: should always match the base used for LogLocator

format_data(value)

format_data_short(value)
    return a short formatted string representation of a number

label_minor(labelOnlyBase)
    switch on/off minor ticks labeling

pprint_val(x, d)

class matplotlib.ticker.LogFormatterExponent(base=10.0, labelOnlyBase=True)
    Bases: matplotlib.ticker.LogFormatter
    Format values for log axis; using exponent = log_base(value)
    base is used to locate the decade tick, which will be the only one to be labeled if labelOnlyBase is False
class matplotlib.ticker.LogFormatterMathtext(base=10.0, labelOnlyBase=True)
    Bases: matplotlib.ticker.LogFormatter
    Format values for log axis; using exponent = log_base(value)

    base is used to locate the decade tick, which will be the only one to be labeled if labelOnlyBase is False

class matplotlib.ticker.Locator
    Bases: matplotlib.ticker.TickHelper

    Determine the tick locations;

    Note, you should not use the same locator between different Axis because the locator stores references to the Axis data and view limits

    MAXTICKS = 1000

autoscale()
    autoscale the view limits

pan(numsteps)
    Pan numticks (can be positive or negative)

raise_if_exceeds(locs)
    raise a RuntimeError if Locator attempts to create more than MAXTICKS locs

refresh()
    refresh internal information based on current lim

tick_values(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(vmin, vmax)
    select a scale for the range from vmin to vmax

    Normally this method is overridden by subclasses to change locator behaviour.

zoom(direction)
    Zoom in/out on axis; if direction is >0 zoom in, else zoom out

class matplotlib.ticker.IndexLocator(base, offset)
    Bases: matplotlib.ticker.Locator

    Place a tick on every multiple of some base number of points plotted, eg on every 5th point. It is assumed that you are doing index plotting; ie the axis is 0, len(data). This is mainly useful for x ticks.
place ticks on the i-th data points where (i-offset)%base==0

**tick_values**(vmin, vmax)

```python
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.

tick_values(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.NullLocator
Bases: matplotlib.ticker.Locator

No ticks

**tick_values**(vmin, vmax)
    ”' Return the locations of the ticks.

Note: Because the values are Null, vmin and vmax are not used in this method.

```python
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

tick_values(vmin, vmax)

view_limits(vmin, vmax)
    Try to choose the view limits intelligently
```

class matplotlib.ticker.LogLocator(base=10.0, subs=[1.0], numdec=4, numticks=15)
Bases: matplotlib.ticker.Locator

Determine the tick locations for log axes

place ticks on the location= base**i*subs[j]

**base**(base)
    set the base of the log scaling (major tick every base**i, i integer)
subs

set the minor ticks the log scaling every base**i*subs[j]

**view limits**

Try to choose the view limits intelligently

**class matplotlib.ticker.AutoLocator**

Bases: matplotlib.ticker.MaxNLocator

class matplotlib.ticker.MultipleLocator(base=1.0)

Bases: matplotlib.ticker.Locator

Set a tick on every integer that is multiple of base in the view interval

**view limits**

Set the view limits to the nearest multiples of base that contain the data

**class matplotlib.ticker.MaxNLocator(**args, **kwargs)**

Bases: matplotlib.ticker.Locator

Select no more than N intervals at nice locations.

Keyword args:

*nbins* Maximum number of intervals; one less than max number of ticks.

*steps* Sequence of nice numbers starting with 1 and ending with 10; e.g., [1, 2, 4, 5, 10]

*integer* If True, ticks will take only integer values.

*symmetric* If True, autoscaling will result in a range symmetric about zero.

*prune* ['lower' | 'upper' | 'both' | 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. 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=None, no ticks will be removed.

**bin_boundaries**

**default_params** = {'trim': True, 'nbins': 10, 'steps': None, 'prune': None, 'integer': False, 'symmetric': False}

**set_params**(**kwargs)

**tick_values**(vmin, vmax)

**view_limits**(dmin, dmax)

**class matplotlib.ticker.AutoMinorLocator**(n=None)

Bases: matplotlib.ticker.Locator
Dynamically find minor tick positions based on the positions of major ticks. Assumes the scale is linear and major ticks are 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** \((vmin, vmax)\)
74.1 `matplotlib.tight_layout`

This module provides routines to adjust subplot params so that subplots are nicely fit in the figure. In doing so, only axis labels, tick labels and axes titles are currently considered.

Internally, it 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).

```python
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 dictionary of subplot parameters so that spacing between subplots are adjusted. Note that this function ignore geometry information of subplot itself, but uses what is given by `nrows_ncols` and `num1num2_list` parameters. Also, the results could be incorrect if some subplots have `adjustable=datalim`.

Parameters:
- `nrows_ncols`: number of rows and number of columns of the grid.
- `num1num2_list`: list of numbers specifying the area occupied by the subplot.
- `subplot_list`: 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. Defaults to `pad_inches`.
- `rect` [left, bottom, right, top]: in normalized (0, 1) figure coordinates.

```python
matplotlib.tight_layout.get_renderer(fig)
```

```python
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 instance
- **axes_list**: a list of axes
- **subplotspec_list**: [a list of subplotspec associated with each] axes in axes_list
- **renderer**: renderer instance
- **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_inches.
- **rect** [if rect is given, it is interpreted as a rectangle] (left, bottom, right, top) in the normalized figure coordinate that the whole subplots area (including labels) will fit into. Default is (0, 0, 1, 1).
CHAPTER

SEVENTYFIVE

TRIANGULAR GRIDS

75.1 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.

Read-only attributes:

- **x**: array of shape (npoints). x-coordinates of grid points.
- **y**: array of shape (npoints). y-coordinates of grid points.
- **triangles**: integer array of shape (ntri,3). For each triangle, the indices of the three points that make up the triangle, ordered in an anticlockwise manner.
- **mask**: optional boolean array of shape (ntri). Which triangles are masked out.
- **edges**: integer array of shape (?,2). All edges of non-masked triangles. Each edge is the start point index and end point index. Each edge (start,end and end,start) appears only once.
- **neighbors**: integer array of shape (ntri,3). 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].

For a Triangulation to be valid it must not have duplicate points, triangles formed from colinear points, or overlapping triangles.

calculate_plane_coefficients(z)

Calculate plane equation coefficients for all unmasked triangles from the point (x,y) coordinates and specified z-array of shape (npoints). Returned array has shape (npoints,3) and allows z-value at (x,y) position in triangle tri to be calculated using \( z = \text{array[tri,0]} * x + \text{array[tri,1]} * y + \text{array[tri,2]} \).

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()
Return an array of triangles that are not masked.

get_trifinder()
Return the default matplotlib.tri.TriFinder of this triangulation, creating it if necessary. This allows the same TriFinder object to be easily shared.

set_mask(mask)
Set or clear the mask array. This is either None, or a boolean array of shape (ntri).

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 matplotlib.tri.Triangulation.get_trifinder(). Derived classes implement __call__(x,y) where x,y are array_like point coordinates of the same shape.

class matplotlib.tri.TrapezoidMapTriFinder(triangulation)
Bases: matplotlib.tri.trifinder.TriFinder
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.
__call__(x, y)
Return an array containing the indices of the triangles in which the specified x,y points lie, or -1 for points that do not lie within a triangle.
x, y are array_like x and y coordinates of the same shape and any number of dimensions.
Returns integer array with the same shape and x and y.

class matplotlib.tri.TriInterpolator(triangulation, z, trifinder=None)
Abstract base class for classes used to perform interpolation on triangular grids.
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
A LinearTriInterpolator performs linear interpolation 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 object
The triangulation to interpolate over.

\( z \): array_like of shape (npoints,)

Array of values, defined at grid points, to interpolate between.

\textit{trifinder} : TriFinder object, optional

If this is not specified, the Triangulation’s default TriFinder will be used by calling matplotlib.tri.Triangulation.get_trifinder().

### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>call</strong>(x, y)</td>
<td>Returns interpolated values at x, y points</td>
</tr>
<tr>
<td>gradient(x, y)</td>
<td>Returns interpolated derivatives at x, y points</td>
</tr>
</tbody>
</table>

**__call__(x, y)**

Returns a masked array containing interpolated values 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**

- \( z \) : np.ma.array
  
  Masked array of the same shape as \( x \) and \( y \); values corresponding to \((x, y)\) points outside of the triangulation are masked out.

**gradient(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**

- \( \text{dzdx}, \text{dzdy} \) : 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

**Parameters**

- \textit{triangulation} : Triangulation object
  
  The triangulation to interpolate over.

- \( z \) : array_like of shape (npoints,)

Bases: matplotlib.tri.triinterpolate.TriInterpolator

A CubicTriInterpolator performs cubic interpolation on triangular grids.

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**

- \textit{triangulation} : Triangulation object
  
  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 object, optional

If not specified, the Triangulation’s default TriFinder will be used by calling matplotlib.tri.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 the way a CubicTriInterpolator computes a cubic interpolation.

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 [R1]. 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 [R2] - PCG sparse solver):

$$E(z) = \frac{1}{2} \int_{\Omega} \left( \left( \frac{\partial^2 z}{\partial x^2} \right)^2 + \left( \frac{\partial^2 z}{\partial y^2} \right)^2 + 2 \left( \frac{\partial^2 z}{\partial y \partial x} \right)^2 \right) dx \, dy$$

(75.1)

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**

[R1], [R2]
Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>call</strong></td>
<td>Returns interpolated values at x,y points</td>
</tr>
<tr>
<td>gradient</td>
<td>Returns interpolated derivatives at x,y points</td>
</tr>
</tbody>
</table>

__call__(x, y)

Returns a masked array containing interpolated values 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**

- `z`: np.ma.array
  - Masked array of the same shape as x and y; values corresponding to (x, y) points outside of the triangulation are masked out.

gradient(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 $\frac{\partial z}{\partial y}$.

**Examples**

An example of effective application is shown below (plot of the direction of the vector field derivated from a known potential field):


**Class** \texttt{matplotlib.tri.TriRefiner} (\texttt{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:

- \texttt{refine_triangulation(return\_tri\_index=False, **kwargs)}, where the optional keyword arguments \texttt{kwargs} are defined in each TriRefiner concrete implementation, and which returns:
  - a refined triangulation
  - optionally (depending on \texttt{return\_tri\_index}), for each point of the refined triangulation: the index of the initial triangulation triangle to which it belongs.

- \texttt{refine\_field(z, triinterpolator=None, **kwargs)}, where:
  - \texttt{z} array of field values (to refine) defined at the base triangulation nodes
  - \texttt{triinterpolator} is a \texttt{TriInterpolator} (optional)
  - the other optional keyword arguments \texttt{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** \texttt{matplotlib.tri.UniformTriRefiner} (\texttt{triangulation})

Bases: \texttt{matplotlib.tri.trirefine.TriRefiner}

Uniform mesh refinement by recursive subdivisions.
Parameters `triangulation`: `Triangulation`  
The encapsulated triangulation (to be refined)

```
refine_field(z, triinterpolator=None, subdiv=3)
```
Refines a field defined on the encapsulated triangulation.

Returns `refi_tri` (refined triangulation), `refi_z` (interpolated values of the field at the node of the refined 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`: integer, optional  
Recursion level for the subdivision. Defaults to 3. Each triangle will be divided into \(4^{**\text{subdiv}}\) child triangles.

**Returns**

`refi_tri`: `Triangulation` object  
The returned refined triangulation

`refi_z`: 1d array of length: `refi_tri` node count.  
The returned interpolated field (at `refi_tri` nodes)

**Examples**

The main application of this method is to plot high-quality iso-contours on a coarse triangular grid (e.g., triangulation built from relatively sparse test data):
**refine_triangulation**(*return_tri_index=False, subdiv=3*)

Computes 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, recursively (level of recursion *subdiv*). In the end, each triangle is hence divided into $4^{*\text{subdiv}}$ child triangles. The default value for *subdiv* is 3 resulting in 64 refined subtriangles for each triangle of the initial triangulation.

**Parameters**

- **return_tri_index**: boolean, optional
  Boolean indicating whether an index table indicating the father triangle index of each point will be returned. Default value False.

- **subdiv**: integer, optional
  Recursion level for the subdivision. Defaults value 3. Each triangle will be divided into $4^{*\text{subdiv}}$ child triangles.

**Returns**

- **refi_triangulation**: `Triangulation`
  The returned refined triangulation

- **found_index**: array-like of integers
  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 object

The encapsulated triangulation to analyze.

**Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>scale_factors</code></td>
<td>Factors to rescale the triangulation into a unit square.</td>
</tr>
</tbody>
</table>

**circle_ratios** *(rescale=True)*

Returns 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 \( \leq 0.5 \) and equals 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**: boolean, optional
  - If True, a rescaling will be internally performed (based on `scale_factors`), so that the (unmasked) triangles fit exactly inside a unit square mesh. Default is True.

**Returns**

- **circle_ratios**: 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** *(min_circle_ratio=0.01, rescale=True)*

Eliminates 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 `matplotlib.tri.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, optional
  - Border triangles with incircle/circumcircle radii ratio \( r/R \) will be removed if \( r/R < min\_circle\_ratio \). Default value: 0.01

- **rescale**: boolean, optional
  - If True, a rescaling will first be internally performed (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. Default
(and recommended) value is True.

**Returns new_mask** : array-like of booleans

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.

**Examples**

Please refer to the following illustrating example:

```
1.0
0.5
0.0
0.5
1.0
1.0
0.5
0.0
0.5
1.0
```

Filtering a Delaunay mesh
(application to high-resolution tricontouring)

**scale_factors**

Factors to rescale the triangulation into a unit square.

Returns $k$, tuple of 2 scale factors.

**Returns** $k$ : tuple of 2 floats (kx, ky)
Tuple of floats that would rescale the triangulation:

\[\text{[triangulation.x * kx, triangulation.y * ky]}\]

fits exactly inside a unit square.
76.1 matplotlib.units

The classes here provide support for using custom classes with matplotlib, eg those that do not expose the array interface but know how to converter themselves to arrays. It also supoprts classes with units and units conversion. Use cases include converters for custom objects, eg 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 a ConversionInterface, and the register with the Registry converter dictionary. For example, here is a complete implementation which supports plotting with native datetime objects:

```python
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 value to a scalar or array'
        return dates.date2num(value)

    @staticmethod
    def axisinfo(unit, axis):
        '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 a converter
units.registry[datetime.date] = DateConverter()
```
class matplotlib.units.AxisInfo(majloc=None, minloc=None, majfmt=None, minfmt=None, label=None, default_limits=None)

information to support default axis labeling and tick labeling, and default limits

majloc and minloc: TickLocators for the major and minor ticks majfmt and minfmt: TickFormatters for the major and minor ticks label: the default axis label default_limits: the default min, max of the axis if no data is present If any of the above are None, the axis will simply use the default

class matplotlib.units.ConversionInterface

The minimal interface for a converter to take custom instances (or sequences) and convert them to values mpl can use

static axisinfo(unit, axis)

return an units.AxisInfo instance for 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. Derived conversion interfaces may opt to pass plain-ol unitless numbers through the conversion interface and this is a helper function for them.

class matplotlib.units.Registry

Bases: dict

register types with conversion interface

get_converter(x)

get the converter interface instance for x, or None
77.1 matplotlib.widgets

77.1.1 GUI Neutral widgets

Widgets that are designed to work for any of the GUI backends. All of these widgets require you to pre-
define an matplotlib.axes.Axes instance and pass that as the first arg. 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 that is connected to a single Axes.

   Attributes:
   ax [Axes] The parent axes for the widget
   canvas [FigureCanvasBase subclass] The parent figure canvas for the widget.
   active [bool] If False, the widget does not respond to events.
   connect_event(event, callback)
      Connect callback with an event.

      This should be used in lieu of figure.canvas.mpl_connect since this function stores call
      back ids for later clean up.
   disconnect_events()
      Disconnect all events created by this widget.
   ignore(event)
      Return True if event should be ignored.

      This method (or a version of it) should be called at the beginning of any event callback.

class matplotlib.widgets.Button(ax, label, image=None, color='0.85', hovercolor='0.95')
   Bases: matplotlib.widgets.AxesWidget

   A GUI neutral button

   The following attributes are accessible
      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.

Call **on_clicked()** to connect to the button

**ax** The matplotlib.axes.Axes instance the button will be placed into.

**label** The button text. Accepts string.

**image** The image to place in the button, if not **None**. Can be any legal arg to imshow (numpy array, matplotlib Image instance, or PIL image).

**color** The color of the button when not activated

**hovercolor** The color of the button when the mouse is over it

**disconnect**(cid)

remove the observer with connection id **cid**

**on_clicked**(func)

When the button is clicked, call this **func** with event

A connection id is returned which can be used to disconnect

**class** matplotlib.widgets.CheckButtons(ax, labels, actives)

**Base**s: matplotlib.widgets.AxesWidget

A GUI neutral radio button

The following attributes are exposed

**ax** The matplotlib.axes.Axes instance the buttons are located in

**labels** List of matplotlib.text.Text instances

**lines** List of (line1, line2) tuples 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.

**rectangles** List of matplotlib.patches.Rectangle instances

Connect to the CheckButtons with the **on_clicked()** method

Add check buttons to matplotlib.axes.Axes instance **ax**

**labels** A len(buttons) list of labels as strings

**actives**

A len(buttons) list of booleans indicating whether the button is active

**disconnect**(cid)

remove the observer with connection id **cid**

**on_clicked**(func)

When the button is clicked, call **func** with button label

A connection id is returned which can be used to disconnect

**class** matplotlib.widgets.Cursor(ax, horizOn=True, vertOn=True, useblit=False, **lineprops)

**Base**s: matplotlib.widgets.AxesWidget

A horizontal and vertical line span the axes that and move with the pointer. You can turn off the hline or vline spectively with the attributes

**horizOn** Controls the visibility of the horizontal line

**vertOn** Controls the visibility of the horizontal line and the visibility of the cursor itself with the **visible** attribute

Add a cursor to **ax**. If **useblit=True**, use the backend-independent blitting features for faster updates (GTKAgg only for now). **lineprops** is a dictionary of line properties.
clear(event)
clear the cursor

onmove(event)
on mouse motion draw the cursor if visible

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 [array] Coordinates of the start of the lasso.
callback [function] Whenever the lasso is released, the callback function is called and passed the vertices of the selected path.

onmove(event)

onrelease(event)
class matplotlib.widgets.LassoSelector(ax, onselect=None, useblit=True, lineprops=None)

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.

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.

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.

Example usage:

```python
ax = subplot(111)
ax.plot(x, y)

def onselect(verts):
    print verts
lasso = LassoSelector(ax, onselect)
```

class matplotlib.widgets.LockDraw

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. The module level “lock” allows someone to grab the lock and prevent other widgets from drawing. Use matplotlib.widgets.lock(someobj) to prevent drawing from available(o)

drawing is available to o

isowner(o)

Return True if o owns this lock

locked()

Return True if the lock is currently held by an owner
**release**(o)
release the lock

**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

Example usage:

```python
from matplotlib.widgets import MultiCursor
from pylab import figure, show, np

t = np.arange(0.0, 2.0, 0.01)
s1 = np.sin(2*np.pi*t)
s2 = np.sin(4*np.pi*t)
fig = figure()
ax1 = fig.add_subplot(211)
ax1.plot(t, s1)
ax2 = fig.add_subplot(212, sharex=ax1)
ax2.plot(t, s2)
multi = MultiCursor(fig.canvas, (ax1, ax2), color='r', lw=1, horizOn=False, vertOn=True)
show()
```

**clear**(event)
clear the cursor

**onmove**(event)

**class** matplotlib.widgets.RadioButtons(ax, labels, active=0, activecolor='blue')

Bases: matplotlib.widgets.AxesWidget

A GUI neutral radio button

The following attributes are exposed

- **ax** The matplotlib.axes.Axes instance the buttons are in
- **activecolor** The color of the button when clicked
- **labels** A list of matplotlib.text.Text instances
- **circles** A list of matplotlib.patches.Circle instances

Connect to the RadioButtons with the **on_clicked()** method

Add radio buttons to matplotlib.axes.Axes instance **ax**

- **labels** A len(buttons) list of labels as strings
- **active** The index into labels for the button that is active
- **activecolor** The color of the button when clicked

**disconnect**(cid)
remove the observer with connection id **cid**
on_clicked(func)
When the button is clicked, call func with button label

A connection id is returned which can be used to disconnect

class matplotlib.widgets.RectangleSelector(ax, onselect, drawtype='box',
    minspanx=None, minspany=None, useblit=False, lineprops=None, rectprops=None,
    spancoords='data', button=None)

Bases: matplotlib.widgets.AxesWidget

Select a min/max range of the x axes for a matplotlib Axes

Example usage:

    from matplotlib.widgets import RectangleSelector
    from pylab import *

    def onselect(eclick, erelease):
        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.RS.active:
            print 'RectangleSelector deactivated.'
            toggle_selector.RS.set_active(False)
        if event.key in ['A', 'a'] and not toggle_selector.RS.active:
            print 'RectangleSelector activated.'
            toggle_selector.RS.set_active(True)

    x = arange(100)/(99.0)
    y = sin(x)
    fig = figure
    ax = subplot(111)
    ax.plot(x,y)
    toggle_selector.RS = RectangleSelector(ax, onselect, drawtype='line')
    connect('key_press_event', toggle_selector)
    show()

Create a selector in ax. When a selection is made, clear the span and call onselect with:

    onselect(pos_1, pos_2)

and clear the drawn box/line. The pos_1 and pos_2 are arrays of length 2 containing the x- and y-coordinate.

If minspanx is not None then events smaller than minspanx in x direction are ignored (it’s the same for y).

The rectangle is drawn with rectprops; default:
rectprops = dict(facecolor='red', edgecolor='black', alpha=0.5, fill=False)

The line is drawn with lineprops; default:

lineprops = dict(color='black', linestyle='-', linewidth=2, alpha=0.5)

Use drawtype if you want the mouse to draw a line, a box or nothing between click and actual position by setting

drawtype = 'line', drawtype='box' or drawtype = 'none'.

spancoords is one of ‘data’ or ‘pixels’. If ‘data’, minspanx and minspanx will be interpreted in the
same coordinates as the x and y axis. If ‘pixels’, they are in pixels.

button is a list of integers indicating which mouse buttons should be used for rectangle selection. You
can also specify a single integer if only a single button is desired. Default is None, which does not
limit which button can be used.

Note, typically: 1 = left mouse button 2 = center mouse button (scroll wheel) 3 = right mouse button

get_active()
Get status of active mode (boolean variable)

ignore(event)
return True if event should be ignored

onmove(event)
on motion notify event if box/line is wanted

press(event)
on button press event

release(event)
on button release event

set_active(active)
Use this to activate / deactivate the RectangleSelector from your program with an boolean
parameter active.

update()
draw using newfangled blit or oldfangled draw depending on useblit

update_background(event)
force an update of the background

class matplotlib.widgets.Slider(ax, label, valmin, valmax, valinit=0.5, valfmt='%1.2f',
closedmin=True, closedmax=True, slidermin=None, slidermax=None, dragging=True, **kwargs)

Bases: matplotlib.widgets.AxesWidget

A slider representing a floating point range

The following attributes are defined ax: the slider matplotlib.axes.Axes instance

val: the current slider value
vline [a matplotlib.lines.Line2D instance] representing the initial value of the slider
poly [A matplotlib.patches.Polygon instance] which is the slider knob
valfmt : the format string for formatting the slider text
label [a matplotlib.text.Text instance] for the slider label
closedmin : whether the slider is closed on the minimum
closedmax : whether the slider is closed on the maximum
slidermin [another slider - if not None, this slider must be] greater than slidermin
slidermax [another slider - if not None, this slider must be] less than slidermax
dragging : allow for mouse dragging on slider

Call on_changed() to connect to the slider event

Create a slider from valmin to valmax in axes ax

valinit The slider initial position

label The slider label

valfmt Used to format the slider value

closedmin and closedmax Indicate whether the slider interval is closed

slidermin and slidermax Used to constrain the value of this slider to the values of other sliders.

additional kwargs are passed on to self.poly which is the matplotlib.patches.Rectangle
which draws the slider knob. See the matplotlib.patches.Rectangle documentation valid property names (e.g., facecolor, edgecolor, alpha, ...)

disconnect(cid)
remove the observer with connection id cid

on_changed(func)

When the slider value is changed, call func with the new slider position

A connection id is returned which can be used to disconnect

reset()
reset the slider to the initial value if needed

set_val(val)

class matplotlib.widgets.SpanSelector(ax, onselect, direction, minspan=None,
useblit=False, rectprops=None, onmove_callback=None)

Bases: matplotlib.widgets.AxesWidget

Select a min/max range of the x or y axes for a matplotlib Axes

Example usage:

ax = subplot(111)
ax.plot(x, y)

def onselect(vmin, vmax):
    print vmin, vmax
span = SpanSelector(ax, onselect, 'horizontal')

*onmove_callback is an optional callback that is called on mouse* move within the span range
Create a span selector in `ax`. When a selection is made, clear the span and call `onselect` with:

```python
onselect(vmin, vmax)
```

and clear the span.

`direction` must be ‘horizontal’ or ‘vertical’

If `minspan` is not `None`, ignore events smaller than `minspan`

The span rectangle is drawn with `rectprops`; default:

```python
rectprops = dict(facecolor='red', alpha=0.5)
```

Set the visible attribute to `False` if you want to turn off the functionality of the span selector

```python
ignore(event)
    return True if event should be ignored
```

```python
new_axes(ax)
```

```python
onmove(event)
    on motion notify event
```

```python
press(event)
    on button press event
```

```python
release(event)
    on button release event
```

```python
update()
    Draw using newfangled blit or oldfangled draw depending on `useblit`
```

```python
update_background(event)
    force an update of the background
```

```python
class matplotlib.widgets.SubplotTool(targetfig, toolfig)
    Bases: matplotlib.widgets.Widget
A tool to adjust to subplot params of a matplotlib.figure.Figure
```

```python
targetfig` The figure instance to adjust
```

```python
toolfig` The figure instance to embed the subplot tool into. If None, a default figure will be created. If you are using this from the GUI
```

```python
funcbottom(val)
```

```python
funchspace(val)
```

```python
funcleft(val)
```

```python
funcright(val)
```

```python
functop(val)
```
funcwspace(val)

class matplotlib.widgets.Widget
    Bases: object
    Abstract base class for GUI neutral widgets
    
    drawon = True

    eventson = True
Part VIII

Glossary
| **AGG** | The Anti-Grain Geometry (Agg) rendering engine, capable of rendering high-quality images |
| **Cairo** | The Cairo graphics engine |
| **dateutil** | The dateutil library provides extensions to the standard datetime module |
| **EPS** | Encapsulated Postscript (EPS) |
| **freetype** | freetype is a font rasterization library used by matplotlib which supports TrueType, Type 1, and OpenType fonts. |
| **GDK** | The Gimp Drawing Kit for GTK+ |
| **GTK** | The GIMP Toolkit (GTK) graphical user interface library |
| **JPG** | The Joint Photographic Experts Group (JPEG) compression method and file format for photographic images |
| **numpy** | numpy is the standard numerical array library for python, the successor to Numeric and numarray. numpy provides fast operations for homogeneous data sets and common mathematical operations like correlations, standard deviation, fourier transforms, and convolutions. |
| **PDF** | Adobe’s Portable Document Format (PDF) |
| **PNG** | Portable Network Graphics (PNG), a raster graphics format that employs lossless data compression which is more suitable for line art than the lossy jpg format. Unlike the gif format, png is not encumbered by requirements for a patent license. |
| **PS** | Postscript (PS) is a vector graphics ASCII text language widely used in printers and publishing. Postscript was developed by adobe systems and is starting to show its age: for example is does not have an alpha channel. PDF was designed in part as a next-generation document format to replace postscript |
| **pygtk** | pygtk provides python wrappers for the GTK widgets library for use with the GTK or GTKAgg backend. Widely used on linux, and is often packages as ‘python-gtk2’ |
| **pyqt** | pyqt provides python wrappers for the Qt widgets library and is required by the matplotlib QtAgg and Qt4Agg backends. Widely used on linux and windows; many linux distributions package this as ‘python-qt3’ or ‘python-qt4’. |
| **python** | python is an object oriented interpreted language widely used for scripting, application development, web application servers, scientific computing and more. |
| **pytz** | pytz provides the Olson tz database in Python. it allows accurate and cross platform timezone calculations and solves the issue of ambiguous times at the end of daylight savings |
| **Qt** | Qt is a cross-platform application framework for desktop and embedded development. |
| **Qt4** | Qt4 is the most recent version of Qt cross-platform application framework for desktop and embedded development. |
| **raster graphics** | Raster graphics, or bitmaps, represent an image as an array of pixels which is resolution dependent. Raster graphics are generally most practical for photo-realistic images, but do not scale easily without loss of quality. |
| **SVG** | The Scalable Vector Graphics format (SVG). An XML based vector graphics format supported by many web browsers. |
TIFF  Tagged Image File Format (TIFF) is a file format for storing images, including photographs and line art.

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.

vector graphics  vector graphics use geometrical primitives based upon mathematical equations to represent images in computer graphics. Primitives can include points, lines, curves, and shapes or polygons. Vector graphics are scalable, which means that they can be resized without suffering from issues related to inherent resolution like are seen in raster graphics. Vector graphics are generally most practical for typesetting and graphic design applications.

wxpython  wxpython provides python wrappers for the wxWidgets library for use with the WX and WX-Agg 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.

matplotlib.afm, 587
matplotlib.animation, 591
matplotlib.artist, 599
matplotlib.axes, 665
matplotlib.axis, 831
matplotlib.backend_bases, 841
matplotlib.backends.backend_pdf, 862
matplotlib.backends.backend_qt4agg, 860
matplotlib.backends.backend_wxagg, 861
matplotlib.cbook, 869
matplotlib.cm, 881
matplotlib.collections, 885
matplotlib.colorbar, 901
matplotlib.colors, 905
matplotlib.dates, 915
matplotlib.dviread, 864
matplotlib.figure, 925
matplotlib.font_manager, 947
matplotlib.fontconfig_pattern, 952
matplotlib.gridspec, 955
matplotlib.legend, 959
matplotlib.lines, 608
matplotlib.markers, 963
matplotlib.mathtext, 968
matplotlib.mlab, 983
matplotlib.patches, 616
matplotlib.path, 1005
matplotlib.projections, 407
matplotlib.projections.polar, 408
matplotlib.pyplot, 1013
matplotlib.sankey, 1189
matplotlib.scale, 405
matplotlib.sphinxext.plot_directive, 369
matplotlib.spines, 1197
matplotlib.text, 653
matplotlibticker, 1201
matplotlib.tight_layout, 1211
matplotlib.transforms, 381
matplotlib.tri, 1213
matplotlib.type1font, 867
matplotlib.units, 1225
matplotlib.widgets, 1227
mpl_toolkits.axes_grid.axes_size, 457
mpl_toolkits.mplot3d.art3d, 513
mpl_toolkits.mplot3d.axes3d, 492
mpl_toolkits.mplot3d.axis3d, 512
mpl_toolkits.mplot3d.proj3d, 517
<table>
<thead>
<tr>
<th>Module</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>matplotlib.afm</td>
<td>587</td>
</tr>
<tr>
<td>matplotlib.animation</td>
<td>591</td>
</tr>
<tr>
<td>matplotlib.artist</td>
<td>599</td>
</tr>
<tr>
<td>matplotlib.axes</td>
<td>665</td>
</tr>
<tr>
<td>matplotlib.axis</td>
<td>831</td>
</tr>
<tr>
<td>matplotlib.backend_bases</td>
<td>841</td>
</tr>
<tr>
<td>matplotlib.backends.backend_pdf</td>
<td>862</td>
</tr>
<tr>
<td>matplotlib.backends.backend_qt4agg</td>
<td>860</td>
</tr>
<tr>
<td>matplotlib.backends.backend_wxagg</td>
<td>861</td>
</tr>
<tr>
<td>matplotlib.cbook</td>
<td>869</td>
</tr>
<tr>
<td>matplotlib.cm</td>
<td>881</td>
</tr>
<tr>
<td>matplotlib.collections</td>
<td>885</td>
</tr>
<tr>
<td>matplotlib.colorbar</td>
<td>901</td>
</tr>
<tr>
<td>matplotlib.colors</td>
<td>905</td>
</tr>
<tr>
<td>matplotlib.dates</td>
<td>915</td>
</tr>
<tr>
<td>matplotlib.dviread</td>
<td>864</td>
</tr>
<tr>
<td>matplotlib.figure</td>
<td>925</td>
</tr>
<tr>
<td>matplotlib.font_manager</td>
<td>947</td>
</tr>
<tr>
<td>matplotlib.fontconfig_pattern</td>
<td>952</td>
</tr>
<tr>
<td>matplotlib.gridspec</td>
<td>955</td>
</tr>
<tr>
<td>matplotlib.legend</td>
<td>959</td>
</tr>
<tr>
<td>matplotlib.lines</td>
<td>608</td>
</tr>
<tr>
<td>matplotlib.markers</td>
<td>963</td>
</tr>
<tr>
<td>matplotlib.mathtext</td>
<td>968</td>
</tr>
<tr>
<td>matplotlib.mlab</td>
<td>983</td>
</tr>
<tr>
<td>matplotlib.patches</td>
<td>616</td>
</tr>
<tr>
<td>matplotlib.path</td>
<td>1005</td>
</tr>
<tr>
<td>matplotlib.projections</td>
<td>407</td>
</tr>
<tr>
<td>matplotlib.projections.polar</td>
<td>408</td>
</tr>
<tr>
<td>matplotlib.pyplot</td>
<td>1013</td>
</tr>
<tr>
<td>matplotlib.sankey</td>
<td>1189</td>
</tr>
<tr>
<td>matplotlib.scale</td>
<td>405</td>
</tr>
<tr>
<td>matplotlib.sphinxext.plot_directive</td>
<td>369</td>
</tr>
<tr>
<td>matplotlib.spines</td>
<td>1197</td>
</tr>
<tr>
<td>matplotlib.text</td>
<td>653</td>
</tr>
<tr>
<td>matplotlib.ticker</td>
<td>1201</td>
</tr>
<tr>
<td>matplotlib.transforms</td>
<td>381</td>
</tr>
<tr>
<td>matplotlib.tri</td>
<td>1213</td>
</tr>
<tr>
<td>matplotlib.typelfont</td>
<td>867</td>
</tr>
<tr>
<td>matplotlib.units</td>
<td>1225</td>
</tr>
<tr>
<td>matplotlib.widgets</td>
<td>1227</td>
</tr>
<tr>
<td>mpl_toolkits.axes_grid.axes_size</td>
<td>457</td>
</tr>
<tr>
<td>mpl_toolkits.mplot3d.art3d</td>
<td>513</td>
</tr>
<tr>
<td>mpl_toolkits.mplot3d.axes3d</td>
<td>492</td>
</tr>
<tr>
<td>mpl_toolkits.mplot3d.axis3d</td>
<td>512</td>
</tr>
<tr>
<td>mpl_toolkits.mplot3d.proj3d</td>
<td>517</td>
</tr>
</tbody>
</table>
Symbols

__call__() (matplotlib.tri.CubicTriInterpolator method), 1217
__call__() (matplotlib.tri.LinearTriInterpolator method), 1215
__call__() (matplotlib.tri.TrapezoidMapTriFinder method), 1214

A

Accent (class in matplotlib.mathtext), 969
accent() (matplotlib.mathtext.Parser method), 978
acorr() (in module matplotlib.pyplot), 1013
acorr() (matplotlib.axes.Axes method), 665
add() (matplotlib.figure.AxesStack method), 925
add() (matplotlib.mlab.FIFOBuﬀer method), 985
add() (matplotlib.sankey_SANkey method), 1193
add_artist() (matplotlib.axes.Axes method), 666
add_auto_adjustable_area()
(mpl_toolkits.axes_grid.axes_divider.Divider method), 458
add_axes() (matplotlib.figure.AxesStack method), 926
add_axobserver() (matplotlib.figure.Figure method), 928
add_callback() (matplotlib.artist.Artist method), 599
add_callback()
(matplotlib.backend_bases.TimerBase method), 859
add_checker() (matplotlib.cm.ScalarMappable method), 881
add_collection() (matplotlib.axes.Axes method), 666
add_collection3d() (mpl_toolkits.mplot3d.Axes3D method), 485
add_collection3d() (mpl_toolkits.mplot3d.Axes3D method), 492
add_container() (matplotlib.axes.Axes method), 666
add_contour_set() (mpl_toolkits.mplot3d.Axes3D method), 492
add_contourf_set() (mpl_toolkits.mplot3d.Axes3D method), 492
add_line() (matplotlib.axes.Axes method), 666
add_lines() (matplotlib.colorbar.Colorbar method), 901
add_lines() (matplotlib.colorbar.ColorbarBase method), 902
add_patch() (matplotlib.axes.Axes method), 667
add_positions()
(matplotlib.collections.EventCollection method), 893
add_subplot() (matplotlib.figure.Figure method), 928
add_table() (matplotlib.axes.Axes method), 667
afmFontProperty() (in module matplotlib.font_manager), 951
AGG, 1239
alias (matplotlib.mathtext.BakomaFonts attribute), 969
aliased_name() (matplotlib.artist.ArtistInspector method), 605
aliased_name() (matplotlib.artist.ArtistInspector method), 605
align_iterators() (in module matplotlib.cbook), 873
allequal() (in module matplotlib.cbook), 874
allow_rasterization() (in module matplotlib.artist), 606
allowed_metadata
(matplotlib.animation.MencoderBase attribute), 595
allpairs() (in module matplotlib.cbook), 874
alltrue() (in module matplotlib.cbook), 874
alphaState() (matplotlib.backends.backend_pdf.PdfFile method), 862
amap() (in module matplotlib.mlab), 987
aname (matplotlib.artist.Artist attribute), 600
anchored() (matplotlib.transforms.BboxBase method), 383
Animation (class in matplotlib.animation), 591
annotate() (in module matplotlib.pyplot), 1014
annotate() (matplotlib.axes.Axes method), 667
Annotation (class in matplotlib.text), 653
append() (mpl_toolkits.axes_grid AxesDivider method), 461
append_axes() (mpl_toolkits.axes_grid AxesDivider method), 461
append_positions() (matplotlib.collections.EventCollection method), 893
append_size() (mpl_toolkits.axes_grid AxesDivider method), 458
apply_aspect() (matplotlib.axes.Axes method), 670
apply_tickdir() (matplotlib.axis.Tick method), 835
apply_tickdir() (matplotlib.axis.XTick method), 838
apply_tickdir() (matplotlib.axis.YTick method), 839
Arc (class in matplotlib.patches), 616
arc() (matplotlib.path.Path class method), 1006
args_key (matplotlib.animation.AVConvBase attribute), 591
args_key (matplotlib.animation.FFMpegBase attribute), 593
args_key (matplotlib.animation.ImageMagickBase attribute), 594
args_key (matplotlib.animation.MencoderBase attribute), 595
Arrow (class in matplotlib.patches), 618
arrow() (in module matplotlib.pyplot), 1018
arrow() (matplotlib.axes.Axes method), 670
ArrowStyle (class in matplotlib.patches), 619
ArrowStyle.BarAB (class in matplotlib.patches), 621
ArrowStyle.BracketA (class in matplotlib.patches), 621
ArrowStyle.BracketAB (class in matplotlib.patches), 621
ArrowStyle.BracketB (class in matplotlib.patches), 622
ArrowStyle.Curve (class in matplotlib.patches), 622
ArrowStyle.CurveA (class in matplotlib.patches), 622
ArrowStyle.CurveB (class in matplotlib.patches), 622
ArrowStyle.CurveFilledA (class in matplotlib.patches), 622
ArrowStyle.CurveFilledAB (class in matplotlib.patches), 622
ArrowStyle.CurveFilledB (class in matplotlib.patches), 623
ArrowStyle.Fancy (class in matplotlib.patches), 623
ArrowStyle.Simple (class in matplotlib.patches), 623
AxesDivider.Wedge (class in matplotlib.patches), 623
Artist (class in matplotlib.artist), 599
artist_picker() (matplotlib.legend.DraggableLegend method), 959
ArtistAnimation (class in matplotlib.animation), 592
AutoLocator (class in matplotlib.ticker), 1208
autoscale() (in module matplotlib.pyplot), 1021
autoscale() (matplotlib.cm.ScalarMappable method), 881
autoscale() (matplotlib.colors.LogNorm method), 910
autoscale() (matplotlib.colors.Normalize method), 911
autoscale() (matplotlib.colors.SymLogNorm method), 911
autoscale() (matplotlib.dates.AutoDateLocator method), 920
auto_adjust_subplotpars() (in module matplotlib.pyplot), 1211
auto_delim() (matplotlib.mathlib.FIFOBuffer method), 985
AsteriskPolygonCollection (class in matplotlib.collections), 885
auto_scale_xyz() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 492
AutoDateFormatter (class in matplotlib.dates), 918
AutoDateLocator (class in matplotlib.dates), 919
autofmt_xdate() (matplotlib.figure.Figure method), 930
AutoHeightChar (class in matplotlib.mathtext), 969
AutoLocator (class in matplotlib.tick), 1208
AutoMinorLocator (class in matplotlib.tick), 1208
 autoscale() (in module matplotlib.pyplot), 1021
 autoscale() (matplotlib.axes.Axes method), 673
 autoscale() (matplotlib.cm.ScalarMappable method), 881
 autoscale() (matplotlib.colors.LogNorm method), 910
 autoscale() (matplotlib.colors.Normalize method), 911
 autoscale() (matplotlib.colors.SymLogNorm method), 911
 autoscale() (matplotlib.dates.AutoDateLocator method), 920
autoscale() (matplotlib.dates.RRuleLocator method), 919
autoscale() (matplotlib.dates.YearLocator method), 920
autoscale() (matplotlib.ticker.Locator method), 1206
 autoscale() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 492
autoscale_None() (matplotlib.cm.ScalarMappable method), 881
autoscale_None() (matplotlib.colors.LogNorm method), 910
autoscale_None() (matplotlib.colors.Normalize method), 911
 autoscale_None() (matplotlib.colors.SymLogNorm method), 911
autoscale_view() (matplotlib.axes.Axes method), 673
 autoscale_view() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 492
AutoWidthChar (class in matplotlib.mathtext), 969
autumn() (in module matplotlib.pyplot), 1021
available() (matplotlib.widgets.LockDraw method), 1230
AVConvBase (class in matplotlib.animation), 591
 AVG ConvFileWriter (class in matplotlib.animation), 591
AVConvWriter (class in matplotlib.animation), 591
ax (matplotlib.colorbar.ColorbarBase attribute), 902
 Axes (class in matplotlib.axes), 665
 Axes (class in matplotlib.pyplot), 1021
Axes3D (class in mpl_toolkits.mplot3d.axes3d), 492
 AxesDivider (class in mpl_toolkits.axes_grid.axes_divider), 461
 AxesLocator (class in mpl_toolkits.axes_grid.axes_divider), 460
AxesStack (class in matplotlib.figure), 925
AxesWidget (class in matplotlib.widgets), 1227
 AxesX (class in mpl_toolkits.axes_grid.axes_size), 457
 AxesY (class in mpl_toolkits.axes_grid.axes_size), 457
axhline() (in module matplotlib.pyplot), 1022
axhline() (matplotlib.axes.Axes method), 673
 axhspan() (in module matplotlib.pyplot), 1023
axhspan() (matplotlib.axes.Axes method), 675
Axis (class in matplotlib.axis), 831
Axis (mpl_toolkits.mplot3d.axes3d), 512
axis (matplotlib.ticker.TickHelper attribute), 1203
axis() (in module matplotlib.pyplot), 1025
axis() (matplotlib.axes.Axes method), 677
axis_date() (matplotlib.axis.Axis method), 831
axis_name (matplotlib.axis.XAxis attribute), 837
axis_name (matplotlib.axis.YAxis attribute), 838
AxisArtist (class in mpl_toolkits.axes_grid.axis_artist), 463
AxisInfo (class in matplotlib.units), 1225
axisinfo() (matplotlib.units.ConversionInterface static method), 1226
AxisLabel (class in mpl_toolkits.axes_grid.axis_artist), 465
avxline() (in module matplotlib.pyplot), 1026
axvline() (matplotlib.axes.Axes method), 677
 axvspan() (in module matplotlib.pyplot), 1028
axvspan() (matplotlib.axes.Axes method), 679
B
back() (matplotlib.backend_bases.NavigationToolbar2 method), 854
back() (matplotlib.cbook.Stack method), 872
BakomaFonts (class in matplotlib.mathtext), 969
bar() (in module matplotlib.pyplot), 1029
bar() (matplotlib.axes.Axes method), 680
 bar() (mpl_toolkits.mplot3d.Axes3D method), 486
bar3d() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 493
bar3d() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 493
barbs() (in module matplotlib.pyplot), 1032
barbs() (matplotlib.axes.Axes method), 683
barh() (in module matplotlib.pyplot), 1036
barh() (matplotlib.axes.Axes method), 687
base() (matplotlib.ticker.LogFormatter method), 1205
base() (matplotlib.ticker.LogFormatter method), 1205
 base_repr() (in module matplotlib.mlab), 987
bbasepath (matplotlib.mathtext.StandardPsFonts attribute), 980
Bbox (class in matplotlib.transforms), 386
bbox_artist() (in module matplotlib.patches), 652
BBoxBase (class in matplotlib.transforms), 383
BBoxTransform (class in matplotlib.transforms), 399
BBoxTransformFrom (class in matplotlib.transforms), 400

Index 1249
BboxTransformTo (class in matplotlib.transforms), 399
bin_boundaries() (matplotlib.ticker.MaxNLocator method), 1208
bin_path() (matplotlib.animation.MovieWriter class method), 596
binary_repr() (in module matplotlib.mlab), 987
binom() (matplotlib.mathtext.Parser method), 978
bivariate_normal() (in module matplotlib.mlab), 987
blended_transform_factory() (in module matplotlib.transforms), 398
BlendedAffineTransform (class in matplotlib.transforms), 397
BlendedGenericTransform (class in matplotlib.transforms), 397
blit() (matplotlib.backend_bases.FigureCanvasBase method), 842
blit() (matplotlib.backends.backend_qt4agg.FigureCanvasQTAgg method), 860
blit() (matplotlib.backends.backend_wxagg.FigureCanvasWxAgg method), 861
bone() (in module matplotlib.pyplot), 1038
BoundaryNorm (class in matplotlib.colors), 906
bounds (matplotlib.transforms.BboxBase attribute), 383
Box (class in matplotlib.mathtext), 970
box() (in module matplotlib.pyplot), 1038
boxplot() (in module matplotlib.pyplot), 1038
boxplot() (matplotlib.axes.Axes method), 689
boxStyle (class in matplotlib.patches), 623
BoxStyle.LArrow (class in matplotlib.patches), 624
BoxStyle.RArrow (class in matplotlib.patches), 625
BoxStyle.Round (class in matplotlib.patches), 625
BoxStyle.Round4 (class in matplotlib.patches), 625
BoxStyle.Roundtooth (class in matplotlib.patches), 625
BoxStyle.Sawtooth (class in matplotlib.patches), 625
BoxStyle.Square (class in matplotlib.patches), 625
broken_barh() (in module matplotlib.pyplot), 1046
broken_barh() (matplotlib.axes.Axes method), 697
BrokenBarHCollection (class in matplotlib.collections), 886
bubble() (matplotlib.cbook.Stack method), 873
bubble() (matplotlib.figure.AxesStack method), 925
Bunch (class in matplotlib.cbook), 869
Button (class in matplotlib.widgets), 1227
button (matplotlib.backend_bases.MouseEvent attribute), 853
button_press_event() (matplotlib.backend_bases.FigureCanvasBase method), 842
button_release_event() (matplotlib.backend_bases.FigureCanvasBase method), 842
byAttribute() (matplotlib.cbook.Sorter method), 872
byItem() (matplotlib.cbook.Sorter method), 872
C
c_over_c() (matplotlib.mathtext.Parser method), 978
cache (matplotlib.colors.ColorConverter attribute), 906
cached (matplotlib.text.Text attribute), 657
Cairo, 1239
calculate_planar_coefficients() (matplotlib.tri.Triangulation method), 1213
callback_registry (class in matplotlib.cbook), 869
can_pan() (matplotlib.axes.Axes method), 699
can_pan() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 410
can_pan() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 493
can_zoom() (matplotlib.axes.Axes method), 699
can_zoom() (matplotlib.axes.Axes method), 699
can_zoom() (matplotlib.projections.polar.PolarAxes method), 410
can_zoom() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 493
center() (matplotlib.mlab.PCA method), 987
center_matrix() (in module matplotlib.mlab), 987
change_geometry() (matplotlib.axes.SubplotBase method), 829
change_geometry() (mpl_toolkits.axes_grid.axes_divider.SubplotDivider method), 460
changed() (matplotlib.cm.ScalarMappable method), 881
Char (class in matplotlib.mathtext), 970
check_update() (matplotlib.cm.ScalarMappable method), 881
CheckButtons (class in matplotlib.widgets), 1228
checksum (matplotlib.dviread.Tfm attribute), 866
Circle (class in matplotlib.patches), 625
circle_ratios() (matplotlib.tri.TriAnalyzer method), 1221
CircleCollection (class in matplotlib.collections), 887
CirclePolygon (class in matplotlib.patches), 627
copy() (matplotlib.font_manager.FontProperties method), 949

copy() (matplotlib.mathtext.GlueSpec method), 972

copy() (matplotlib.mathtext.Parser.State method), 978

copy() (matplotlib.path.Path method), 1007

copy_properties() (matplotlib.backend_bases.GraphicsContextBase method), 849

corners() (matplotlib.transforms.BboxBase method), 384

count_contains() (matplotlib.transforms.BboxBase method), 384

count_overlaps() (matplotlib.transforms.BboxBase method), 384

create_dummy_axis() (matplotlib.ticker.TickHelper method), 1203

creatFontList() (in module matplotlib.font_manager), 951

cross_from_above() (in module matplotlib.mlab), 990

cross_from_below() (in module matplotlib.mlab), 990

csd() (in module matplotlib.mlab), 991
csd() (in module matplotlib.pyplot), 1084
csd() (matplotlib.axes.Axes method), 731
csv2rec() (in module matplotlib.mlab), 992
csvformat_factory() (in module matplotlib.mlab), 992
CubicTriInterpolator (class in matplotlib.tri), 1215
current_key_axes() (matplotlib.figure.AxesStack method), 925

cursor (class in matplotlib.widgets), 1228

cursors (class in matplotlib.backend_bases), 841

CURRE3 (matplotlib.path.Path attribute), 1006
CURRE4 (matplotlib.path.Path attribute), 1006
customspace() (matplotlib.mathtext.Parser method), 978

d dict_delall() (in module matplotlib.cbook), 875

detrend() (in module matplotlib.mlab), 992

detrend_linear() (in module matplotlib.mlab), 992

detrend_mean() (in module matplotlib.mlab), 992
detrend_none() (in module matplotlib.mlab), 993
dict_delall() (in module matplotlib.cbook), 875
disable_mouse_rotation() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 495
disconnect() (in module matplotlib.pyplot), 1087
disconnect() (matplotlib.cbook.CallbackRegistry method), 870
disconnect() (matplotlib.widgets.Button method), 1228
disconnect() (matplotlib.widgets.CheckButtons method), 1228
disconnect() (matplotlib.widgets.RadioButton method), 1231
disconnect() (matplotlib.widgets.Slider method), 1234

daylight (matplotlib.dates.DayLocator method), 921
dblclick (matplotlib.backend_bases.MouseEvent attribute), 853
dedent() (in module matplotlib.cbook), 874
deprecated() (matplotlib.backend_bases.MouseEvent attribute), 853
delay (matplotlib.animation.ImageMagickBase attribute), 594
delete_masked_points() (in module matplotlib.mlab), 874
demean() (in module matplotlib.mlab), 992
deprecated() (in module matplotlib.cbook), 875
depth (matplotlib.dviread.Tfm attribute), 866
depth (matplotlib.transforms.Transform attribute), 389
design_size (matplotlib.dviread.Tfm attribute), 866
destroy() (matplotlib.backend_bases.FigureManagerBase method), 848
destroy() (matplotlib.mathtext.Fonts method), 971
destroy() (matplotlib.mathtext.TrueTypeFonts method), 981
detrend() (in module matplotlib.mlab), 992

detrend_linear() (in module matplotlib.mlab), 992
detrend_mean() (in module matplotlib.mlab), 992
detrend_none() (in module matplotlib.mlab), 993
dict_delall() (in module matplotlib.cbook), 875
disable_mouse_rotation() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 495
disconnect() (in module matplotlib.pyplot), 1087
disconnect() (matplotlib.cbook.CallbackRegistry method), 870
disconnect() (matplotlib.widgets.Button method), 1228
disconnect() (matplotlib.widgets.CheckButtons method), 1228
disconnect() (matplotlib.widgets.RadioButton method), 1231
disconnect() (matplotlib.widgets.Slider method), 1234

Index 1253
disconnect_events() (matplotlib.widgets.AxesWidget method),

dist() (in module matplotlib.mlab),
dist_point_to_segment() (in module matplotlib.mlab),
distances_along_curve() (in module matplotlib.mlab),

Divider (class in mpl_toolkits.axes_grid.axes_divider),

do_3d_projection() (mpl_toolkits.mplot3d.art3d.Line3DCollection method),
do_3d_projection() (mpl_toolkits.mplot3d.art3d.Patch3D method),
do_3d_projection() (mpl_toolkits.mplot3d.art3d.Patch3DCollection method),
do_3d_projection() (mpl_toolkits.mplot3d.art3d.Poly3DCollection method),
do_3d_projection() (mpl_toolkits.mplot3d.art3d.Text3D method),
do_3d_projection() (mpl_toolkits.mplot3d.axes3d.Axes3D method),
draw() (matplotlib.collections.Collection method),
draw() (matplotlib.collections.CircleCollection method),
draw() (matplotlib.collections.EllipseCollection method),
draw() (matplotlib.collections.PathCollection method),
draw() (matplotlib.collections.PolyCollection method),
draw() (matplotlib.collections.QuadMesh method),
draw() (mpl_toolkits.mplot3d.art3d.Line3D method),
draw() (mpl_toolkits.mplot3d.art3d.Line3DCollection method),
draw() (mpl_toolkits.mplot3d.art3d.Patch3D method),
draw() (mpl_toolkits.mplot3d.art3d.Patch3DCollection method),
draw() (mpl_toolkits.mplot3d.art3d.Poly3DCollection method),
draw() (mpl_toolkits.mplot3d.text.Text3D method),
draw() (mpl_toolkits.mplot3d.text.TextWithDash method),

donthing_callback() (in module matplotlib.mlab),
dpi (matplotlib.figure.Figure attribute),
drag_pan() (matplotlib.axes.Axes method),
drag_pan() (matplotlib.backend_bases.NavigationToolbar2 method),
draggable() (matplotlib.legend.Legend method),

DraggableLegend (class in matplotlib.legend),
drange() (in module matplotlib.dates),
draw() (in module matplotlib.pyplot),
draw() (matplotlib.artist.Artist method),
draw() (matplotlib.artist.Artist method),
draw() (matplotlib.axis.Axis method),
draw() (matplotlib.axis.Tick method),
draw() (matplotlib.backend_bases.FigureCanvasBase method),
draw() (matplotlib.backend_bases.NavigationToolbar2 method),
draw() (matplotlib.legend.Legend method),
draw() (matplotlib.legend.Legend method),
draw() (matplotlib.legend.Legend method),

draw() (matplotlib.pyplot),
draw() (matplotlib.widgets.AxesWidget method),
draw() (matplotlib.collections.Collection method),

draw_all() (matplotlib.colorbar.ColorbarBase method),
draw_artist() (matplotlib.axes.Axes method),
draw_artist() (matplotlib.figure.Figure method), 933
draw_bbox() (in module matplotlib.patches), 652
draw_cursor() (matplotlib.backend_bases.FigureCanvasBase method), 842
draw_event() (matplotlib.backend_bases.FigureCanvasBase method), 842
draw_frame() (matplotlib.legend.Legend method), 961
draw_gouraud_triangle() (matplotlib.backend_bases.RendererBase method), 856
draw_gouraud_triangels() (matplotlib.backend_bases.RendererBase method), 856
draw_idle() (matplotlib.backend_bases.FigureCanvasBase method), 842
draw_image() (matplotlib.backend_bases.RendererBase method), 856
draw_markers() (matplotlib.backend_bases.RendererBase method), 856
draw_path() (mpl_toolkits.mplot3d.axis3d.Axis method), 512
draw_path_collection() (matplotlib.backend_bases.RendererBase method), 857
draw_quad_mesh() (matplotlib.backend_bases.RendererBase method), 857
draw_rubberband() (matplotlib.backend_bases.NavigationToolbar2 method), 854
draw_text() (matplotlib.figure.Figure method), 657
draw_text() (matplotlib.backend_bases.RendererBase method), 857
draw_text() (matplotlib.backend_bases.RendererBase method), 857
DrawEvent (class in matplotlib.backends), 841
drawn (matplotlib.widgets.Widget attribute), 1236
drawRectangle() (matplotlib.backends.backend_qt4agg.FigureCanvasBase method), 860
drawStyleKeys (matplotlib.lines.Line2D attribute), 610
drawStyles (matplotlib.lines.Line2D attribute), 610
Dvi (class in matplotlib.dviread), 864
DviFont (class in matplotlib.dviread), 865
dynamic_update() (matplotlib.backend_bases.NavigationToolbar2 method), 854

E
Ellipse (class in matplotlib.patches), 631
EllipseCollection (class in matplotlib.collections), 891
embedTTF() (matplotlib.backends.backend_pdf.PdfFile method), 862
empty() (matplotlib.cbook.Stack method), 873
Encoding (class in matplotlib.dviread), 865
Encodings (matplotlib.dviread.Encoding attribute), 865
end() (matplotlib.backends.backend_pdf.Stream method), 864
end_group() (matplotlib.mathtext.Parser method), 978
end_pan() (matplotlib.axes.Axes method), 734
enter_notify_event() (matplotlib.backend_bases.FigureCanvasBase method), 843
entropy() (in module matplotlib.mlab), 993
environment variable
HOME, 330, 333
MPLCONFIGDIR, 330, 333
PATH, 55, 59, 60, 64, 333
PYTHONPATH, 333, 340
epoch2num() (in module matplotlib.dates), 917
EPS, 1239
Error() (in module matplotlib.mathtext), 970
errorbar() (in module matplotlib.pyplot), 1088
toplot() (matplotlib.axes.Axes method), 734
Event (class in matplotlib.backend_bases), 842
EventCollection (class in matplotlib.collections), 842
EventCollection (class in matplotlib.pyplot), 1090
EventCollection (matplotlib.backend_bases.FigureCanvasBase attribute), 843
Events (matplotlib.backend_bases.FigureCanvasBase attribute), 843
exception_to_str() (in module matplotlib.cbook), 875
exec_key (matplotlib.animation.AVConvBase attribute), 591
exec_key (matplotlib.animation.FFMpegBase attribute), 593
exec_key (matplotlib.animation.ImageMagickBase attribute), 594
exec_key (matplotlib.animation.MencoderBase attribute), 595
exp_safe() (in module matplotlib.mlab), 993
expanded() (matplotlib.transforms.BboxBase method), 384
extend_positions() (matplotlib.collections.EventCollection method), 893
extents (matplotlib.transforms.BboxBase attribute), 384
F
factory() (matplotlib.mathtext.GlueSpec class method), 972
family_escape() (in module matplotlib.fontconfig_pattern), 953
family_unescape() (in module matplotlib.fontconfig_pattern), 953
FancyArrow (class in matplotlib.patches), 632
FancyArrowPatch (class in matplotlib.patches), 633
FancyBboxPatch (class in matplotlib.patches), 636
FFMpegBase (class in matplotlib.animation), 593
FFMpegFileWriter (class in matplotlib.animation), 593
FFMpegWriter (class in matplotlib.animation), 593
fftsurr() (in module matplotlib.mlab), 993
FIFOBu
fill() (matplotlib.patches.Patch attribute), 641
fill() (in module matplotlib.pyplot), 1097
fill() (in module matplotlib.backends.backend_pdf), 864
filled_markers (matplotlib.patches.Patch attribute), 641
filled_markers (matplotlib.markers.MarkerStyle attribute), 965
Fill () (in module matplotlib.mlab), 993
Fill () (in module matplotlib.pyplot), 1097
Fill () (in module matplotlib.backends.backend_pdf), 864
fill() (in module matplotlib.pyplot), 1097
fill() (matplotlib.axes.Axes method), 739
fill_between() (in module matplotlib.pyplot), 1099
fill_between() (matplotlib.axes.Axes method), 742
fill_betweenx() (in module matplotlib.pyplot), 1103
fill_betweenx() (matplotlib.axes.Axes method), 746
dotted_markers (matplotlib.lines.Line2D attribute), 610
dotted_markers (matplotlib.markers.MarkerStyle attribute), 965
Fill () (in module matplotlib.mlab), 993
Fill Styles (matplotlib.lines.Line2D attribute), 610
Fill Styles (matplotlib.markers.MarkerStyle attribute), 965
finalize_offset() (matplotlib.legend.DraggableLegend method), 959
find() (module matplotlib.mlab), 993
find() (module matplotlib.pyplot), 867
finddir() (module matplotlib.cbook), 875
findfont() (module matplotlib.font_manager), 951
findfont() (matplotlib.font_manager.FontManager method), 947
findobj() (module matplotlib.pyplot), 1106
findobj() (matplotlib.artist.Artist method), 600
findobj() (matplotlib.artist.ArtistInspector method), 605
findSystemFonts() (module matplotlib.font_manager), 951
finish() (matplotlib.animation.MovieWriter method), 593
<table>
<thead>
<tr>
<th>Method/Function</th>
<th>Class/Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>finish()</td>
<td>matplotlib.animation.MovieWriter method, 597</td>
</tr>
<tr>
<td>finish()</td>
<td>matplotlibsankey.Sankey method, 1194</td>
</tr>
<tr>
<td>fix_minus()</td>
<td>matplotlibticker.Formatter method, 1203</td>
</tr>
<tr>
<td>fix_minus()</td>
<td>matplotlibticker.ScalarFormatter method, 1204</td>
</tr>
<tr>
<td>Fixed</td>
<td>mpl_toolkits.axes_grid.axes_size, 457</td>
</tr>
<tr>
<td>FixedFormatter</td>
<td>matplotlibticker, 1203</td>
</tr>
<tr>
<td>FixedLocator</td>
<td>matplotlibticker, 1106</td>
</tr>
<tr>
<td>flatten()</td>
<td>matplotlibbackend_bases.RendererBase method, 875</td>
</tr>
<tr>
<td>flippy()</td>
<td>matplotlibbackend_bases.FigureCanvasBase method, 843</td>
</tr>
<tr>
<td>font</td>
<td>mathtex.Parser.State attribute, 978</td>
</tr>
<tr>
<td>font()</td>
<td>mathtex.Parser method, 978</td>
</tr>
<tr>
<td>FontconfigPatternParser</td>
<td>matplotlibfontconfig_pattern, 952</td>
</tr>
<tr>
<td>FontEntry</td>
<td>matplotlibfont_manager, 947</td>
</tr>
<tr>
<td>FontManager</td>
<td>matplotlibfont_manager, 947</td>
</tr>
<tr>
<td>fontmap</td>
<td>mathtex.StandardPsFonts attribute, 980</td>
</tr>
<tr>
<td>fontName()</td>
<td>backend_backends.backend_pdf.PdfFileFrameFormat method, 862</td>
</tr>
<tr>
<td>FontProperties</td>
<td>matplotlibfont_manager, 949</td>
</tr>
<tr>
<td>Fonts</td>
<td>mathtext, 970</td>
</tr>
<tr>
<td>format_coord()</td>
<td>matplotlibaxes.Axes method, 749</td>
</tr>
<tr>
<td>format_coord()</td>
<td>matplotlibprojections.polar.PolarAxes method, 410</td>
</tr>
<tr>
<td>format_coord()</td>
<td>mpl_toolkits.mplot3d.axes3d.Axes3D from_bounding_box method, 495</td>
</tr>
<tr>
<td>format_data()</td>
<td>matplotlibticker.Formatter method, 1203</td>
</tr>
<tr>
<td>format_data()</td>
<td>matplotlibticker.LogFormatter method, 1205</td>
</tr>
<tr>
<td>format_data()</td>
<td>matplotlibticker.ScalarFormatter method, 1204</td>
</tr>
<tr>
<td>format_data_short()</td>
<td>matplotlibticker.ScalarFormatter method, 1204</td>
</tr>
<tr>
<td>format_xdata()</td>
<td>matplotlibaxes.Axes method, 749</td>
</tr>
<tr>
<td>format_ydata()</td>
<td>matplotlibaxes.Axes method, 749</td>
</tr>
<tr>
<td>format_zdata()</td>
<td>mpl_toolkits.mplot3d.axes3d.Axes3D method, 495</td>
</tr>
<tr>
<td>FormatBool</td>
<td>matplotlibmlab, 985</td>
</tr>
<tr>
<td>FormatDate</td>
<td>matplotlibmlab, 985</td>
</tr>
<tr>
<td>FormatDatetimelike</td>
<td>matplotlibmlab, 986</td>
</tr>
<tr>
<td>FormatFloat</td>
<td>matplotlibmlab, 986</td>
</tr>
<tr>
<td>FormatStr</td>
<td>matplotlibmlab, 986</td>
</tr>
<tr>
<td>FormatInt</td>
<td>matplotlibmlab, 986</td>
</tr>
<tr>
<td>FormatThousands</td>
<td>matplotlibmlab, 987</td>
</tr>
<tr>
<td>forward()</td>
<td>matplotlibbackend_bases.NavigationToolbar2 method, 854</td>
</tr>
<tr>
<td>forward()</td>
<td>matplotlibbook.Stack method, 873</td>
</tr>
<tr>
<td>frac()</td>
<td>mathtext.Parser method, 978</td>
</tr>
<tr>
<td>Fraction</td>
<td>mpl_toolkits.axes_grid.axes_size, 457</td>
</tr>
<tr>
<td>from_any()</td>
<td>mpl_toolkits.axes_grid.axes_size, 457</td>
</tr>
<tr>
<td>from_bounds()</td>
<td>matplotlibtransforms.Bbox static method, 387</td>
</tr>
<tr>
<td>from_extents()</td>
<td>matplotlibtransforms.Bbox static method, 387</td>
</tr>
<tr>
<td>from_levels_and_colors()</td>
<td>matplotlibcolors, 911</td>
</tr>
<tr>
<td>from_list()</td>
<td>matplotlibcolors.LinearSegmentedColormap static method, 909</td>
</tr>
<tr>
<td>from_values()</td>
<td>matplotlibtransforms.Affine2D static method, 394</td>
</tr>
<tr>
<td>fromstr()</td>
<td>matplotlibmlab.FormatBool method, 985</td>
</tr>
<tr>
<td>fromstr()</td>
<td>matplotlibmlab.FormatDate method, 985</td>
</tr>
</tbody>
</table>
fromstr() (matplotlib.mlab.FormatDatetime method), 986
fromstr() (matplotlib.mlab.FormatFloat method), 986
fromstr() (matplotlib.mlab.FormatInt method), 986
fromstr() (matplotlib.mlab.FormatObj method), 986
frozen() (matplotlib.transforms.Affine2DBase method), 393
frozen() (matplotlib.transforms.BboxBase method), 384
frozen() (matplotlib.transforms.BlendedGenericTransform method), 397
frozen() (matplotlib.transforms.CompositeGenericTransform method), 398
frozen() (matplotlib.transforms.IdentityTransform method), 395
frozen() (matplotlib.transforms.TransformNode method), 383
frozen() (matplotlib.transforms.TransformWrapper method), 391
full_screen_toggle() (matplotlib.backend_bases.FigureManagerBase method), 848
fully_contains() (matplotlib.transforms.BboxBase method), 384
fully_containsx() (matplotlib.transforms.BboxBase method), 384
fully_containsy() (matplotlib.transforms.BboxBase method), 384
fully_overlaps() (matplotlib.transforms.BboxBase method), 384
FuncAnimation (class in matplotlib.animation), 594
funcbottom() (matplotlib.widgets/SubplotTool method), 1235
FuncFormatter (class in matplotlib.ticker), 1204
funchspace() (matplotlib.widgets/SubplotTool method), 1235
funcleft() (matplotlib.widgets/SubplotTool method), 1235
funcright() (matplotlib.widgets/SubplotTool method), 1235
function() (matplotlib.mathtext.Parser method), 978
funcwspace() (matplotlib.widgets/SubplotTool method), 1235

G

gca() (in module matplotlib.pyplot), 1106
gca() (matplotlib.figure.Figure method), 935
gcf() (in module matplotlib.pyplot), 1107
gci() (in module matplotlib.pyplot), 1107

GDK, 1239
generate_fontconfig_pattern() (in module matplotlib.fontconfig_pattern), 953
genfrac() (matplotlib.mathtext.Parser method), 978
get() (in module matplotlib.artist), 606
get() (matplotlib.cbook.RingBuffer method), 872
get() (matplotlib.figure.AxesStack method), 925
get(aa) (matplotlib.lines.Line2D method), 610
get_aa() (matplotlib.transforms.Patch method), 641
get_active() (matplotlib.widgets.RectangleSelector method), 1233
get_adjustable() (matplotlib.axes.Axes method), 749
get_affine() (matplotlib.transforms.AffineTransform method), 392
get_affine() (matplotlib.transforms.BlendedGenericTransform method), 397
get_affine() (matplotlib.transforms.CompositeGenericTransform method), 398
get_affine() (matplotlib.transforms.IdentityTransform method), 395
get_affine() (matplotlib.transforms.Transform method), 389
get_agg_filter() (matplotlib.artist.Artist method), 600
get_aliases() (matplotlib.artist.ArtInspector method), 606
get_alpha() (matplotlib.artist.Artist method), 600
get_alpha() (matplotlib.backend_bases.GraphicsContextBase method), 849
get_alt_path() (matplotlib.markers.MarkerStyle method), 965
get_alt_transform() (matplotlib.markers.MarkerStyle method), 965
get_anchord() (matplotlib.axes.Axes method), 749
get_anchord() (mpl_toolkits.axes_grid.axes_divider.Divider method), 458
get_angle() (matplotlib.afm.AFM method), 587
get_animated() (matplotlib.artist.Artist method), 600
get_annotated() (matplotlib.patches.ConnectionPatch method), 629
get_linestyle() (matplotlib.lines.Line2D method), 610
get_linestyle() (matplotlib.patches.Patch method), 641
get_linestyles() (matplotlib.collections.Collection method), 888
get_linewidth() (matplotlib.backend_bases.GraphicsContextBase method), 850
get_linewidth() (matplotlib.collections.Collection method), 888
get_linewidth() (matplotlib.lines.Line2D method), 611
get_linewidth() (matplotlib.patches.Patch method), 641
get_linewidths() (matplotlib.collections.Collection method), 888
get_loc() (matplotlib.axis.Tick method), 836
get_locator() (matplotlib.dates.AutoDateLocator method), 920
get_locator() (mpl_toolkits.axes_grid.axes_divider.Divider method), 458
get_ls() (matplotlib.lines.Line2D method), 611
get_ls() (matplotlib.patches.Patch method), 641
get_lw() (matplotlib.lines.Line2D method), 611
get_lw() (matplotlib.patches.Patch method), 641
get_major_formatter() (matplotlib.axis.Axis method), 832
get_major_locator() (matplotlib.axis.Axis method), 832
get_major_ticks() (matplotlib.axis.Axis method), 832
get_major_ticks() (mpl_toolkits.mplot3d.axis3d.Axes3D method), 512
get_major ticklabels() (matplotlib.axis.Axis method), 832
get_major ticklines() (matplotlib.axis.Axis method), 832
get_major ticklocs() (matplotlib.axis.Axis method), 832
get_marker() (matplotlib.lines.Line2D method), 611
get_marker() (matplotlib.markers.MarkerStyle method), 965
get_markeredgecolor() (matplotlib.lines.Line2D method), 611
get_markeredgewidth() (matplotlib.lines.Line2D method), 611
get_markerfacecolor() (matplotlib.lines.Line2D method), 611
get_markerfacecoloralt() (matplotlib.lines.Line2D method), 611
get_markersize() (matplotlib.lines.Line2D method), 611
get маркеровку() (matplotlib.lines.Line2D method), 611
get_masked_triangles() (matplotlib.tri.Triangulation method), 1213
get_matrix() (matplotlib.projections.polar.PolarAffine method), 408
get_matrix() (matplotlib.projections.polar.PolarAxes.PolarAffine method), 409
get_matrix() (matplotlib.transforms.Affine2D method), 394
get_matrix() (matplotlib.transforms.BboxTransform method), 399
get_matrix() (matplotlib.transforms.BboxTransformFrom method), 400
get_matrix() (matplotlib.transforms.BboxTransformTo method), 400
get_matrix() (matplotlib.transforms.BlendedAffine2D method), 398
get_matrix() (matplotlib.transforms.CompositeAffine2D method), 399
get_matrix() (matplotlib.transforms.IdentityTransform method), 395
get_matrix() (matplotlib.transforms.ScaledTranslation method), 400
get_matrix() (matplotlib.transforms.Transform method), 390
get_mec() (matplotlib.lines.Line2D method), 611
get_metrics() (matplotlib.mathtext.Fonts method), 971
get_mew() (matplotlib.lines.Line2D method), 611
get_mfc() (matplotlib.lines.Line2D method), 611
get_mfcalt() (matplotlib.lines.Line2D method), 611
Index
get_transform() (matplotlib.markers.MarkerStyle method), 965
get_transform() (matplotlib.patches.Patch method), 642
get_transform() (matplotlib.scale.LinearScale method), 405
get_transform() (matplotlib.scale.LogScale method), 405
get_transform() (matplotlib.scale.ScaleBase method), 406
get_transform() (matplotlib.scale.SymmetricalLogScale method), 406
get_transform() (mpl_toolkits.axes_grid_artist.AxisArtist method), 463
get_transformed_clip_path_and_affine() (matplotlib.artist.Artist method), 601
get_transformed_path_and_affine() (matplotlib.transforms.TransformedPath method), 400
get_transformed_points_and_affine() (matplotlib.transforms.TransformedPath method), 400
get_transforms() (matplotlib.collections.Collection method), 889
get_trifinder() (matplotlib.tri.Triangulation method), 1214
get_underline_thickness() (matplotlib.afm.AFM method), 588
get_underline_thickness() (matplotlib.mathtext.Fonts method), 972
get_underline_thickness() (matplotlib.mathtext.StandardPsFonts method), 980
get_underline_thickness() (matplotlib.mathtext.TruetypeFonts method), 981
get_unicode_index() (in module matplotlib.mathtext), 982
get_unit() (matplotlib.text.OffsetFrom method), 656
get_unit_generic() (matplotlib.dates.RRuleLocator static method), 919
get_units() (matplotlib.axis.Axis method), 833
get_url() (matplotlib.artist.Artist method), 601
get_url() (matplotlib.backend_bases.GraphicsContextBase method), 850
get_urls() (matplotlib.collections.Collection method), 889
get_used_characters() (matplotlib.mathtext.Fonts method), 972
get_useLocale() (matplotlib.ticker.ScalarFormatter method), 1204
get_useOffset() (matplotlib.ticker.ScalarFormatter method), 1204
get_va() (matplotlib.text.Text method), 658
get_valid_values() (matplotlib.artist.ArtistInspector method), 606
get_variant() (matplotlib.font_manager.FontProperties method), 950
get_variant() (matplotlib.text.Text method), 658
get_vector() (mpl_toolkits.mplot3d.art3d.Poly3DCollection method), 515
get_vertical() (mpl_toolkits.axes_grid_artist.Axis method), 601
get_vertical() (mpl_toolkits.mplot3d.axes3d.Divider method), 512
get_vertical() (matplotlib.axis.Axis method), 833
get_view_interval() (matplotlib.axis.Tick method), 836
get_view_interval() (matplotlib.axis.XAxis method), 837
get_view_interval() (matplotlib.axis.XTick method), 838
get_view_interval() (matplotlib.axis.YAxis method), 838
get_view_interval() (matplotlib.axis.YTick method), 839
get_view_interval() (mpl_toolkits.mplot3d.axes3d.Axis method), 512
get_visible() (matplotlib.artist.Artist method), 601
get_vsize_hsize() (mpl_toolkits.axes_grid_artist.Divider method), 458
get_w_lims() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 496
get_weight() (matplotlib.afm.AFM method), 588
get_weight() (matplotlib.font_manager.FontProperties method), 950
get_weight() (matplotlib.text.Text method), 659
get ylim3d() (mpl_toolkits.mplot3d.axes3d.Axes3D
method), 496
get ymajorticklabels() (mpl_toolkits.mplot3d.axes3d.Axes3D
method), 496
get yminorticklabels() (mpl_toolkits.mplot3d.axes3d.Axes3D
method), 496
get yscale() (matplotlib.axes.Axes method), 754
get yticklabels() (matplotlib.axes.Axes method), 754
get yticklines() (matplotlib.axes.Axes method), 754
get zbound() (mpl_toolkits.mplot3d.axes3d.Axes3D
method), 496
get zlabel() (mpl_toolkits.mplot3d.axes3d.Axes3D
method), 497
get zlim() (mpl_toolkits.mplot3d.axes3d.Axes3D
method), 497
get zlim3d() (mpl_toolkits.mplot3d.axes3d.Axes3D
method), 497
get zmajorticklabels()
  (mpl_toolkits.mplot3d.axes3d.Axes3D
method), 497
get zminorticklabels()
  (mpl_toolkits.mplot3d.axes3d.Axes3D
method), 497
get_zorder() (matplotlib.artist.Artist method), 602
get_zscale() (mpl_toolkits.mplot3d.axes3d.Axes3D
method), 497
get_zticklabels() (mpl_toolkits.mplot3d.axes3d.Axes3D
method), 497
get_zticklines() (mpl_toolkits.mplot3d.axes3d.Axes3D
method), 497
get_zticks() (mpl_toolkits.mplot3d.axes3d.Axes3D
method), 497
getp() (in module matplotlib.artist), 607
getpoints() (matplotlib.patches.YAArrow method), 652
getRealpathAndStat (class in matplotlib.cbook), 870
ginput() (in module matplotlib.pyplot), 1107
ginput() (matplotlib.figure.Figure method), 937
Glue (class in matplotlib.mathtext), 972
GlueSpec (class in matplotlib.gridspec), 955
GlueSpecBase (class in matplotlib.gridspec), 956
GridSpec (class in matplotlib.gridspec), 955
GridSpecBase (class in matplotlib.gridspec), 956
Group() (in module matplotlib.mathtext.parser), 978
Group() (in module matplotlib.cbook), 870
grow() (matplotlib.mathtext.Accent method), 969
grow() (matplotlib.mathtext.Box method), 970
grow() (matplotlib.mathtext.Char method), 970
grow() (matplotlib.mathtext.Glue method), 972
grow() (matplotlib.mathtext.List method), 974
grow() (matplotlib.mathtext.Node method), 977
H
  HAND (matplotlib.backend_bases.Cursors attribute), 841
has_data() (matplotlib.axes.Axes method), 755
has_inverse (matplotlib.transforms.Transform method), 390
has_nonfinite (matplotlib.path.Path attribute), 1007
hatch() (matplotlib.path.Path method), 1008
have_units() (matplotlib.artist.Artist method), 602
have_units() (matplotlib.axes.Axes method), 833
have_units() (mpl_toolkits.mplot3d.axes3d.Axes3D
method), 498
HBox (class in matplotlib.mathtext), 973
HCentered (class in matplotlib.mathtext), 973
height (matplotlib.dviread.Tfm attribute), 866
height (matplotlib.mathtext.Kern attribute), 973
height (matplotlib.transforms.BboxBase attribute), 384
hex2color() (in module matplotlib.colors), 912
hexbin() (in module matplotlib.pyplot), 1109
hexbin() (matplotlib.axes.Axes method), 761
hist() (in module matplotlib.pyplot), 1112
hist() (matplotlib.axes.Axes method), 758
hist2d() (in module matplotlib.pyplot), 1115
hist2d() (matplotlib.axes.Axes method), 761
hitlist() (matplotlib.artist.Artist method), 602
hlines() (in module matplotlib.pyplot), 1117
hlines() (matplotlib.axes.Axes method), 762
Hlist (class in matplotlib.mathtext), 973
hlist_out() (matplotlib.mathtext.Hlist method), 980
hms0d (matplotlib.dates.DateFormatter attribute), 918
hold() (in module matplotlib.pyplot), 1118
hold() (matplotlib.axes.Axes method), 763
hold() (matplotlib.figure.Figure method), 938
HOME, 330, 333
home() (matplotlib.backend_bases.NavigationToolBar2 method), 854
home() (matplotlib.cbook.Stack method), 873
hot() (in module matplotlib.pyplot), 1118
HourLocator (class in matplotlib.dates), 921
hours() (in module matplotlib.dates), 923
hpack() (matplotlib.mathtext.Hlist method), 973
Hrule (class in matplotlib.mathtext), 973
hs() (in module matplotlib.pyplot), 1118
hsv() (in module matplotlib.pyplot), 1118
hsv_to_rgb() (in module matplotlib.colors), 912
id (matplotlib.cbook.Scheduler attribute), 872
identity() (in module matplotlib.mlab), 995
identity() (matplotlib.transforms.Affine2D static method), 394
IdentityTransform (class in matplotlib.transforms), 395
Idle (class in matplotlib.cbook), 871
idle_event() (matplotlib.backend_bases.FigureCanvas method), 843
IdleEvent (class in matplotlib.backend_bases), 851
idlelock (matplotlib.cbook.Scheduler attribute), 872
ignore() (matplotlib.transforms.Bbox method), 387
ignore() (matplotlib.transforms.Bbox method), 387
ignore() (matplotlib.widgets.AxesWidget method), 1227
ignore() (matplotlib.widgets.LassoSelector method), 1230
ignore() (matplotlib.widgets.RectangleSelector method), 1233
ignore() (matplotlib.widgets.SpanSelector method), 1235
illegal_s (matplotlib.dates DateFormatter attribute), 918
ImageGrid (class in mpl_toolkits.axes_grid.axes_grid), 462
ImageMagickBase (class in matplotlib.animation), 594
ImageMagickFileWriter (class in matplotlib.animation), 594
ImageMagickWriter (class in matplotlib.animation), 594
imageObject() (matplotlib.backends.backend_pdf.PdfPages method), 862
imread() (in module matplotlib.pyplot), 1118
imsave() (in module matplotlib.pyplot), 1119
imshow() (matplotlib.axes.Axes method), 764
in_axes() (matplotlib.axes.Axes method), 766
inaxes (matplotlib.backend_bases.LocationEvent attribute), 852
inaxes (matplotlib.backend_bases.MouseEvent attribute), 853
IndexDateFormatter (class in matplotlib.dates), 918
IndexLocator (class in matplotlib ticking), 1206
infodict() (matplotlib.backends.backend_pdf.PdfPages method), 863
init3d() (mpl_toolkits.mplot3d.axis3d.Axis method), 512
input_dims (matplotlib.transforms.Transform attribute), 390
inside_poly() (in module matplotlib.mlab), 995
interpolated() (matplotlib.path.Path method), 1008
intersection() (matplotlib.transforms.BboxBase static method), 384
BboxBase.intersects_bbox() (matplotlib.path.Path method), 1008
intersects_path() (matplotlib.path.Path method), 1008
interval (matplotlib.backend_bases.TimerBase attribute), 859
intervalx (matplotlib.transforms.BboxBase attribute), 385
intervaly (matplotlib.transforms.BboxBase attribute), 385
inv_transform() (in module mpl_toolkits.mplot3d.proj3d), 517
index
is_unit() (matplotlib.transforms.BboxBase method), 385
is_writable_file_like() (in module matplotlib.cbook), 876
isAvailable() (matplotlib.animation.MovieWriter class method), 597
iscolor() (in module mpl_toolkits.mplot3d.art3d), 516
ishold() (in module matplotlib.pyplot), 1121
ishold() (matplotlib.axes.Axes method), 766
isinteractive() (in module matplotlib.pyplot), 1121
ishold() (matplotlib.axes.Axes method), 766
isowner() (matplotlib.widgets.LockDraw method), 1230
ispower2() (in module matplotlib.mlab), 995
issubclass_safe() (in module matplotlib.cbook), 876
isvector() (in module matplotlib.mlab), 995
iter_segments() (matplotlib.path.Path method), 1008
iter_ticks() (matplotlib.axis.Axis method), 833
iterable() (in module matplotlib.cbook), 876

J
Jet() (in module matplotlib.pyplot), 1121
join() (matplotlib.cbook.Grouper method), 871
joined() (matplotlib.cbook.Grouper method), 871
JPG, 1239
juggle_axes() (in module mpl_toolkits.mplot3d.art3d), 516

K
Kern (class in matplotlib.mathtext), 973
kern() (matplotlib.mathtext.Hlist method), 973
key_press() (matplotlib.backend_bases.FigureManagerBase method), 848
key_press_event() (matplotlib.backend_bases.FigureCanvasBase method), 843
key_press_handler() (in module matplotlib.backend_bases), 860
key_press_handler_id (matplotlib.backend_bases.FigureManagerBase attribute), 848
key_release_event() (matplotlib.backend_bases.FigureCanvasBase method), 843
KeyEvent (class in matplotlib.backend_bases), 852
kwdoc() (in module matplotlib.artist), 607

L
l1norm() (in module matplotlib.mlab), 995
l2norm() (in module matplotlib.mlab), 995
label_minor() (matplotlib.ticker.LogFormatter method), 1205
label_outer() (matplotlib.axes.SubplotBase method), 830
LABE.LAP() (mpl_toolkits.axes_grid.axis_artist.AxisArtist attribute), 463
Lasso (class in matplotlib.widgets), 1229
LassoSelector (class in matplotlib.widgets), 1230
last() (matplotlib.mlab.FIFOBu method), 985
leave_notify_event() (matplotlib.backend_bases.FigureCanvasBase method), 843
Legend (class in matplotlib.legend), 959
legend() (in module matplotlib.pyplot), 1121
legend() (matplotlib.axes.Axes method), 766
legend() (matplotlib.figure.Figure method), 938
less_simple_linear_interpolation() (in module matplotlib.mlab), 995
levypdf() (in module matplotlib.mlab), 996
LightSource (class in matplotlib.colors), 907
limit_range_for_scale() (matplotlib.axis.Axis method), 833
limit_range_for_scale() (matplotlib.scale.LogScale method), 405
limit_range_for_scale() (matplotlib.scale.ScaleBase method), 406
Line2D (class in matplotlib.lines), 608
line2d() (in module mpl_toolkits.mplot3d.proj3d), 517
line2d_dist() (in module mpl_toolkits.mplot3d.proj3d), 517
line2d_seg_dist() (in module mpl_toolkits.mplot3d.proj3d), 517
Line3D (class in mpl_toolkits.mplot3d.art3d), 513
Line3DCollection (class in mpl_toolkits.mplot3d.art3d), 513
line_2d_to_3d() (in module mpl_toolkits.mplot3d.art3d), 516
line_collection_2d_to_3d() (in module mpl_toolkits.mplot3d.art3d), 516
linear_spine() (matplotlib.spines.Spine class method), 1199
LinearLocator (class in matplotlib.ticker), 1207
LinearScale (class in matplotlib.scale), 405
LinearSegmentedColormap (class in matplotlib.colors), 908
LinearTriInterpolator (class in matplotlib.tri), 1214
LineCollection (class in matplotlib.collections), 894
lineStyles (matplotlib.lines.Line2D attribute), 612
LINETO (matplotlib.path.Path attribute), 1006
List (class in matplotlib.mathtext), 974
list() (matplotlib.animation.MovieWriterRegistry method), 597
list Fonts() (in module matplotlib.font_manager), 952
ListedColormap (class in matplotlib.colors), 909
listFiles() (in module matplotlib.cbook), 876
locally_modified_subplot_params() (matplotlib.gridspec.GridSpec method), 955
locate() (mpl_toolkits.axes_grid1.axes_divider.Divider method), 459
LocationEvent (class in matplotlib.backend_bases), 852
Locator (class in matplotlib.ticker), 1206
locator (matplotlib.axis.Ticker attribute), 836
locator_params() (in module matplotlib.pyplot), 1124
locator_params() (matplotlib.axes.Axes method), 769
locator_params() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 498
LockDraw (class in matplotlib.widgets), 1230
locked() (matplotlib.widgets.LockDraw method), 1230
locs (matplotlib.ticker.Formatter attribute), 1203
log2() (in module matplotlib.mlab), 996
LogFormatter (class in matplotlib.ticker), 1205
LogFormatterExponent (class in matplotlib.ticker), 1205
LogFormatterMathtext (class in matplotlib.ticker), 1205
LogLocator (class in matplotlib.ticker), 1207
loglog() (in module matplotlib.pyplot), 1125
loglog() (matplotlib.axes.Axes method), 769
LogNorm (class in matplotlib.colors), 909
LogScale (class in matplotlib.scale), 405
logspace() (in module matplotlib.mlab), 996
longest_contiguous_ones() (in module matplotlib.mlab), 996
longest_ones() (in module matplotlib.mlab), 996
M
main() (matplotlib.mathtext.Parser method), 978
mainloop() (matplotlib.backend_bases.ShowBase method), 859
make_axes() (in module matplotlib.colorbar), 903
make_axes_gridspec() (in module matplotlib.colorbar), 904
make_compound_path() (matplotlib.path.Path class method), 1009
make_compound_path_from_polys() (matplotlib.path.Path class method), 1009
makeMappingArray() (in module matplotlib.colors), 912
margins() (in module matplotlib.pyplot), 1127
margins() (matplotlib.axes.Axes method), 771
margins() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 498
markerObject() (matplotlib.backends.backend_pdf.PdfFile method), 862
markers (matplotlib.lines.Line2D attribute), 612
markers (matplotlib.markers.MarkerStyle attribute), 965
MarkerStyle (class in matplotlib.markers), 964
math() (matplotlib.mathtext.Parser method), 979
math_string() (matplotlib.mathtext.Parser method), 979
math_to_image() (in module matplotlib.mathtext), 982
MathtextBackend (class in matplotlib.mathtext), 975
MathtextBackendAgg (class in matplotlib.mathtext), 975
MathtextBackendBitmap (class in matplotlib.mathtext), 976
MathtextBackendCairo (class in matplotlib.mathtext), 976
MathtextBackendPath (class in matplotlib.mathtext), 976
MathtextBackendPdf (class in matplotlib.mathtext), 976
MathtextBackendPs (class in matplotlib.mathtext), 976
MathtextBackendSvg (class in matplotlib.mathtext), 977
MathTextParser (class in matplotlib.mathtext), 974
MathTextWarning, 975
matplotlib.afm (module), 587
Index 1275

matplotlib.animation (module), 591
matplotlib.artist (module), 599
matplotlib.axes (module), 665
matplotlib.axis (module), 831
matplotlib.backends.backend_bases (module), 841
matplotlib.backends.backend_pdf (module), 862
matplotlib.backends.backend_qt4agg (module), 860
matplotlib.backends.backend_wxagg (module), 861
matplotlib.cbook (module), 869
matplotlib.cm (module), 881
matplotlib.collections (module), 885
matplotlib.colors (module), 905
matplotlib.dates (module), 915
matplotlib.dviread (module), 925
matplotlib.legend (module), 959
matplotlib.lines (module), 608
matplotlib.markers (module), 963
matplotlib.mathtext (module), 968
matplotlib.mlab (module), 983
matplotlib.margins (module), 1213
matplotlib.matplotlib_fname() (in module matplotlib), 584
MatplotlibDeprecationWarning, 871
matshow() (matplotlib.axes.Axes method), 772
max (matplotlib.transforms.BboxBase attribute), 385
maxdict (class in matplotlib.cbook), 876
MaxHeight (class in mpl_toolkits.axes_grid.axes_size), 457
MaxNLocator (class in matplotlib.ticker), 1208
MAXTICKS (matplotlib.ticker.Locator attribute), 1206
MaxWidth (class in mpl_toolkits.axes_grid.axes_size), 457
MemoryMonitor (class in matplotlib.cbook), 871
MencoderBase (class in matplotlib.animation), 595
MencoderFileWriter (class in matplotlib.animation), 595
MencoderWriter (class in matplotlib.animation), 595
MicrosecondLocator (class in matplotlib.dates), 921
min (matplotlib.transforms.BboxBase attribute), 385
minorticks_off() (in module matplotlib.pyplot), 1128
minorticks_off() (matplotlib.axes.Axes method), 774
minorticks_on() (in module matplotlib.pyplot), 1128
minorticks_on() (matplotlib.axes.Axes method), 774
MinuteLocator (class in matplotlib.dates), 920
motion_notify_event() (matplotlib.backend_bases.FigureCanvasBase method), 844
mouse_init() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 498
mouse_move() (matplotlib.backend_bases.NavigationToolbar2 method), 854
MouseEvent (class in matplotlib.backend_bases), 853
movavg() (in module matplotlib.mlab), 996
MOVE (matplotlib.backend_bases.Cursors attribute), 841
move_from_center() (in module mpl_toolkits.mplot3d.axis3d), 513
MOVETO (matplotlib.path.Path attribute), 1006
MovieWriter (class in matplotlib.animation), 596
MovieWriterRegistry (class in matplotlib.animation), 597
mpl_connect() (matplotlib.backend_bases.FigureCanvasBase method), 844
null" (module matplotlib.transforms.Bbox method), 387
mx2num() (module matplotlib.dates), 917

N
Name (module matplotlib.backends.backend_pdf), 862
name (module matplotlib.axes.Axes attribute), 774
name (module matplotlib_toolkits.mplot3d.axes3d.Axes3D attribute), 499
NavigationToolbar2 (module matplotlib.backend_bases), 853
NavigationToolbar2QTAgg (module matplotlib.backends.backend_qt4agg), 860
NavigationToolbar2WxAgg (module matplotlib.backends.backend_wxagg), 861
NegFil (module matplotlib.mathtext), 977
NegFill (module matplotlib.mathtext), 977
NegFilll (module matplotlib.mathtext), 977
new_axes() (module matplotlib.widgets.SpanSelector method), 1235
new_figure_manager() (module matplotlib.backends.backend_pdf), 864
new_figure_manager() (module matplotlib.backends.backend_qt4agg), 861
new_figure_manager() (module matplotlib.backends.backend_wxagg), 861
new_figure_manager_given_figure() (module matplotlib.backends.backend_pdf), 864
new_figure_manager_given_figure() (module matplotlib.backends.backend_qt4agg), 861
new_figure_manager_given_figure() (module matplotlib.backends.backend_wxagg), 862
new_frame_seq() (module matplotlib.animation.Animation method), 592
new_frame_seq() (module matplotlib.animation.FuncAnimation method), 594
new_gc() (module matplotlib.backend_bases.RendererBase method), 858
new_horizontal() (module matplotlib_toolkits.axes_grid.axes_divider.AxesDivider method), 461
new_locator() (module matplotlib_toolkits.axes_grid.axes_divider.Divider method), 459
new_saved_frame_seq() (module matplotlib.animation.Animation method), 592
new_saved_frame_seq() (module matplotlib.animation.FuncAnimation method), 594
new_subplotspec() (module matplotlib.gridspec.GridSpecBase method), 956
new_timer() (module matplotlib.backends.backend_bases.FigureCanvasBase method), 845
new_vertical() (module matplotlib_toolkits.axes_grid.axes_divider.AxesDivider method), 461
no_norm() (module matplotlib.colors), 912
Node (module matplotlib.mathtext), 977
non_math() (module matplotlib.mathtext.Parser method), 979
NonGuiException, 855
NoNorm (module matplotlib.colors), 910
nonsingular() (module matplotlib.transforms), 400
nonsingular() (module matplotlib.dates.AutoDateLocator method), 920
nonsingular() (module matplotlib.dates.DateLocator method), 918
norm (module matplotlib.cm.ScalarMappable attribute), 882
norm_angle() (module mpl_toolkits.mplot3d.art3d), 516
norm_flat() (module matplotlib.mlab), 996
norm_text_angle() (module mpl_toolkits.mplot3d.art3d), 516
Normalize (module matplotlib.colors), 912
normalize() (module matplotlib.colors), 912
normpdf() (module matplotlib.mlab), 996
Null (module matplotlib.cbook), 871
null() (module matplotlib.transforms.Bbox static method), 387
NullFormatter (module matplotlib.tick), 1204
NullLocator (class in matplotlib.ticker), 1207
num2date() (in module matplotlib.dates), 917
num2epoch() (in module matplotlib.dates), 917
NUM_VERTICES_FOR_CODE (matplotlib.path.Path attribute), 1006
numpy, 1239
numvertices (matplotlib.patches.RegularPolygon attribute), 648
offset_line() (in module matplotlib.mlab), 996
OffsetFrom (class in matplotlib.text), 656
OFFSETTEXTPAD (matplotlib.axis.Axis attribute), 831
on_changed() (matplotlib.widgets.Slider method), 1234
on_clicked() (matplotlib.widgets.Button method), 1228
on_clicked() (matplotlib.widgets.CheckButtons method), 1228
on_clicked() (matplotlib.widgets.RadioButtons method), 1231
on_mappable_changed() (matplotlib.colorbar.Colorbar method), 901
ontrue() (in module matplotlib.cbook), 877
onHilite() (matplotlib.backend_bases.FigureCanvasBase method), 845
onmove() (matplotlib.widgets.Cursor method), 1229
onmove() (matplotlib.widgets.Lasso method), 1229
onmove() (matplotlib.widgets.LassoSelector method), 1230
onmove() (matplotlib.widgets.MultiCursor method), 1231
onmove() (matplotlib.widgets.RectangleSelector method), 1233
onmove() (matplotlib.widgets.SpanSelector method), 1235
onpick() (matplotlib.lines.VertexSelector method), 616
onpress() (matplotlib.widgets.LassoSelector method), 1230
onrelease() (matplotlib.widgets.Lasso method), 1229
onrelease() (matplotlib.widgets.LassoSelector method), 1230
onRemove() (matplotlib.backend_bases.FigureCanvasBase method), 845
open_group() (matplotlib.backend_bases.RendererBase method), 858
Operator (class in matplotlib.backends.backend_pdf), 862
operatorname() (matplotlib.mathtext.Parser method), 979
option_image_nocomposite() (matplotlib.backend_bases.RendererBase method), 858
option_scale_image() (matplotlib.backend_bases.RendererBase method), 858
orientation (matplotlib.patches.RegularPolygon attribute), 649
OSXInstalledFonts() (in module matplotlib.font_manager), 951
output_args (matplotlib.animation.FFMpegBase attribute), 593
output_args (matplotlib.animation.ImageMagickBase attribute), 594
output_args (matplotlib.animation.MencoderBase attribute), 595
output_dims (matplotlib.transforms.Transform attribute), 390
over() (in module matplotlib.pyplot), 1128
overlaps() (matplotlib.transforms.BboxBase method), 385
overline() (matplotlib.mathtext.Parser method), 979
P
p0 (matplotlib.transforms.BboxBase attribute), 385
p1 (matplotlib.transforms.BboxBase attribute), 385
Padded (class in mpl_toolkits.axes_grid.axes_size), 457
padded() (matplotlib.transforms.BboxBase method), 385
paintEvent() (matplotlib.backends.backend_qt4agg.FigureCanvasQTAgg method), 860
pan() (matplotlib.axis.Axis method), 833
pan() (matplotlib.backend_bases.NavigationToolbar2 method), 854
pan() (matplotlib.ticker.Locator method), 1206
parse() (matplotlib.fontconfig_pattern.FontconfigPatternParser method), 952
parse() (matplotlib.mathtext.MathTextParser method), 974

Index 1277
Matplotlib, Release 1.3.0

parse() (matplotlib.mathtext.Parser method), 979
parse_afm() (in module matplotlib.mathtext.afm), 588
Parser (class in matplotlib.mathtext), 977
Parser.State (class in matplotlib.mathtext), 977
parts (matplotlib.type1font.Type1Font attribute), 867, 868
pass_through (matplotlib.transforms.TransformNode attribute), 383
Patch (class in matplotlib.patches), 639
Patch3D (class in mpl_toolkits.mplot3d.art3d), 514
Patch3DCollection (class in mpl_toolkits.mplot3d.art3d), 514
patch_2d_to_3d() (in module mpl_toolkits.mplot3d.art3d), 516
patch_collection_2d_to_3d() (in module mpl_toolkits.mplot3d.art3d), 516
PatchCollection (class in matplotlib.collections), 895
PATH, 55, 59, 60, 64
Path (class in matplotlib.path), 1005
path_in_path() (in module matplotlib.path), 1011
path_intersects_path() (in module matplotlib.path), 1011
path_length() (in module matplotlib.mlab), 996
path_to_3d_segment() (in module mpl_toolkits.mplot3d.art3d), 516
PathCollection (class in matplotlib.collections), 896
PathPatch (class in matplotlib.patches), 643
PathPatch3D (class in mpl_toolkits.mplot3d.art3d), 514
pathpatch_2d_to_3d() (in module mpl_toolkits.mplot3d.art3d), 516
paths_to_3d_segments() (in module mpl_toolkits.mplot3d.art3d), 517
pause() (in module matplotlib.pyplot), 1128
PCA (class in matplotlib.mlab), 987
pchanged() (matplotlib.artist.Artist method), 602
pcolor() (in module matplotlib.pyplot), 1128
pcolormesh() (matplotlib.axes.Axes method), 777
pcolorfast() (matplotlib.axes.Axes method), 777
pcolormesh() (in module matplotlib.pyplot), 1131
PDF, 1239
PdfFile (class in matplotlib.backends.backend_pdf), 862
PdfPages (class in matplotlib.backends.backend_pdf), 863
pdfRepr() (in module matplotlib.backends.backend_pdf), 864
persp_transformation() (in module mpl_toolkits.mplot3d.proj3d), 517
pick() (matplotlib.artist.Artist method), 602
pick() (matplotlib.axes.Axes method), 780
pick() (matplotlib.axes.Axes method), 780
pick_event() (matplotlib.backend_bases.FigureCanvasBase method), 845
pickable() (matplotlib.artist.Artist method), 602
PickEvent (class in matplotlib.backend_bases), 855
pickle_dump() (in module matplotlib.font_manager), 952
pickle_load() (in module matplotlib.font_manager), 952
pie() (in module matplotlib.pyplot), 1133
pie() (matplotlib.axes.Axes method), 780
pieces() (in module matplotlib.cbook), 877
pink() (in module matplotlib.pyplot), 1134
plot() (in module matplotlib.pyplot), 1134
plot() (matplotlib.axes.Axes method), 781
plot() (matplotlib.axes.Axes method), 781
plot() (mpl_toolkits.mplot3d.Axes3D method), 470
plot3D() (mpl_toolkits.mplot3d.Axes3D method), 499
plot3D() (mpl_toolkits.mplot3d.Axes3D method), 499
plot_date() (in module matplotlib.pyplot), 1137
plot_date() (matplotlib.axes.Axes method), 784
plot_surface() (mpl_toolkits.mplot3d.Axes3D method), 473
plot_surface() (mpl_toolkits.mplot3d.Axes3D method), 473
plot_trisurf() (mpl_toolkits.mplot3d.Axes3D method), 499
plot_trisurf() (mpl_toolkits.mplot3d.Axes3D method), 499
plot_wireframe() (mpl_toolkits.mplot3d.Axes3D method), 472
plot_wireframe() (mpl_toolkits.mplot3d.Axes3D method), 472
plotfile() (in module matplotlib.pyplot), 1139
plotting() (in module matplotlib.pyplot), 541
PNG, 1239
point_in_path() (in module matplotlib.path), 1011
point_in_path_collection() (in module matplotlib.path), 1012
POINTER (matplotlib.backend_bases.Cursors attribute), 841
points_in_path() (in module matplotlib.path), 1012
points_to_pixels() (matplotlib.backend_bases.RendererBase method), 858
polar() (in module matplotlib.pyplot), 1140
PolarAffine (class in matplotlib.projections.polar), 408
PolarAxes (class in matplotlib.projections.polar), 408
PolarAxes.InvertedPolarTransform (class in matplotlib.projections.polar), 408
PolarAxes.PolarAffine (class in matplotlib.projections.polar), 409
PolarAxes.PolarTransform (class in matplotlib.projections.polar), 409
PolarAxes.RadialLocator (class in matplotlib.projections.polar), 410
PolarAxes ThetaFormatter (class in matplotlib.projections.polar), 410
PolarTransform (class in matplotlib.projections.polar), 413
Poly3DCollection (class in mpl_toolkits.mplot3d.art3d), 515
poly_below() (in module matplotlib.mlab), 997
poly_between() (in module matplotlib.mlab), 997
poly_collection_2d_to_3d() (in module mpl_toolkits.mplot3d.art3d), 517
PolyCollection (class in matplotlib.collections), 896
Polygon (class in matplotlib.patches), 644
pop_state() (matplotlib.mathtext.Parser method), 979
popall() (in module matplotlib.cbook), 877
pprint_getters() (matplotlib.artist.ArtistInspector method), 606
pprint_setters() (matplotlib.artist.ArtistInspector method), 606
pprint_setters_rest() (matplotlib.artist.ArtistInspector method), 606
pprint_val() (matplotlib.ticker.LogFormatter method), 1205
pprint_val() (matplotlib.ticker.ScalarFormatter method), 1204
prctile() (in module matplotlib.mlab), 997
prctile_rank() (in module matplotlib.mlab), 997
prepca() (in module matplotlib.mlab), 997
press() (matplotlib.backend_bases.NavigationToolbar2 method), 854
press() (matplotlib.widgets.RectangleSelector method), 1233
press() (matplotlib.widgets.SpanSelector method), 1235
press_pan() (matplotlib.backend_bases.NavigationToolbar2 method), 854
press_zoom() (matplotlib.backend_bases.NavigationToolbar2 method), 854
print_bmp() (matplotlib.backend_bases.FigureCanvasBase method), 845
print_cycles() (in module matplotlib.cbook), 877
print_eps() (matplotlib.backend_bases.FigureCanvasBase method), 845
print_figure() (matplotlib.backend_bases.FigureCanvasBase method), 845
print_figure() (matplotlib.backends.backend_qt4agg.FigureCanvasQTAgg method), 860
print_figure() (matplotlib.backends.backend_wxagg.FigureCanvasWxAgg method), 861
print_jpeg() (matplotlib.backend_bases.FigureCanvasBase method), 846
print_jpg() (matplotlib.backend_bases.FigureCanvasBase method), 846
print_pdf() (matplotlib.backend_bases.FigureCanvasBase method), 846
print_pgf() (matplotlib.backend_bases.FigureCanvasBase method), 846
print_png() (matplotlib.backend_bases.FigureCanvasBase method), 846
print_ps() (matplotlib.backend_bases.FigureCanvasBase method), 846
print_raw() (matplotlib.backend_bases.FigureCanvasBase method), 846
print_rgba() (matplotlib.backend_bases.FigureCanvasBase method), 846
print_svg() (matplotlib.backend_bases.FigureCanvasBase method), 846
print_svgz() (matplotlib.backend_bases.FigureCanvasBase method), 846
print_tif() (matplotlib.backend_bases.FigureCanvasBase.
mETHOD), 846
print_tiff() (matplotlib.backend_bases.FigureCanvasBase.
mETHOD), 846
prism() (in module matplotlib.pyplot), 1140
process() (matplotlib.cbook.CallbackRegistry.
mETHOD), 870
process_projection_requirements() (in module mat-
plotlib.projections), 407
process_selected() (matplotlib.lines.VertexSelector.
mETHOD), 616
process_value() (matplotlib.colors.Normalize.static.
mETHOD), 911
proj_points() (in module mpl_toolkits.mplot3d proj3d), 517
proj_trans_clip_points() (in module mpl_toolkits.
mplot3d proj3d), 517
proj_trans_points() (in module mpl_toolkits.mplot3d.
mplot3d proj3d), 517
proj_transform() (in module mpl_toolkits.mplot3d.
mplot3d proj3d), 517
proj_transform_clip() (in module mpl_toolkits.mplot3d.
mplot3d proj3d), 518
proj_transform_vec() (in module mpl_toolkits.
mplot3d proj3d), 518
proj_transform_vec_clip() (in module mpl_toolkits.
mplot3d proj3d), 518
quad2cubic() (in module matplotlib.mlab), 998
QuadMesh (class in matplotlib.collections), 897
quiver() (matplotlib.axes.Axes method), 789
quiver() (in module matplotlib.pyplot), 1143
quiverkey() (in module matplotlib.pyplot), 1145
quiverkey() (matplotlib.axes.Axes method), 791
RadialLocator (class in matplotlib.projections.polar), 414
RadioButtons (class in matplotlib.widgets), 1231
radius (matplotlib.patches.Circle attribute), 626
radius (matplotlib.patches.RegularPolygon attribute), 649
raise_if_exceeds() (matplotlib.ticker.Locator.
mETHOD), 1206
rc() (in module matplotlib.pyplot), 1146
rc() (in module matplotlib.pyplot), 1147
rc_params() (in module matplotlib.pyplot), 584
rc_params_from_file() (in module matplotlib.pyplot), 1147
RcParams (class in matplotlib.pyplot), 585
readonly (matplotlib.path.Path attribute), 1009
rec2csv() (in module matplotlib.mlab), 998
rec2txt() (in module matplotlib.mlab), 999
rec_append_fields() (in module matplotlib.mlab), 999
rec_context() (in module matplotlib.pyplot), 1147
rec_context() (in module matplotlib.pyplot), 1147
rec_drop_fields() (in module matplotlib.mlab), 999
rec_groupby() (in module matplotlib.pyplot), 999
rec_join() (in module matplotlib.pyplot), 1000
rec_keep_fields() (in module matplotlib.pyplot), 1000
rec_summarize() (in module matplotlib.pyplot), 1000
recache() (matplotlib.pyplot.Line2D method), 612
recache_always() (matplotlib.pyplot.Line2D method), 612
recs_join() (in module matplotlib.mlab), 1000
Index
render_rect_filled() (matplotlib.mathtext.MathtextBackendCairo method), 976
render_rect_filled() (matplotlib.mathtext.MathtextBackendPath method), 976
render_rect_filled() (matplotlib.mathtext.MathtextBackendPdf method), 976
render_rect_filled() (matplotlib.mathtext.MathtextBackendPs method), 977
render_rect_filled() (matplotlib.mathtext.MathtextBackendSvg method), 977
RendererBase (class in matplotlib.backend_bases), 855
report() (matplotlib.cbook.MemoryMonitor method), 871
report_memory() (in module matplotlib.cbook), 877
required_group() (matplotlib.mathtext.Parser method), 979
reserveObject() (matplotlib.backends.backend_pdf.PdfFile method), 862
reset() (matplotlib.widgets.Slider method), 1234
reset_position() (matplotlib.axes.Axes method), 792
resize() (matplotlib.backend_bases.FigureCanvasBase method), 847
resize() (matplotlib.backend_bases.FigureManagerBase method), 848
resize_event() (matplotlib.backend_bases.FigureCanvasBase method), 847
ResizeEvent (class in matplotlib.backend_bases), 858
restore() (matplotlib.backend_bases.GraphicsContextBase method), 850
restrict_dict() (in module matplotlib.cbook), 877
revcmap() (in module matplotlib.cm), 883
reverse_dict() (in module matplotlib.cbook), 877
rgb2hex() (in module matplotlib.colors), 912
rgb_to_hsv() (in module matplotlib.colors), 913
rgrids() (in module matplotlib.pyplot), 1147
RingBuffer (class in matplotlib.cbook), 872
rk4() (in module matplotlib.mlab), 1000
rms_flat() (in module matplotlib.mlab), 1001
rot_x() (in module mpl_toolkits.mplot3d.proj3d), 518
rotate() (matplotlib.transforms.Affine2D method), 394
rotate_around() (matplotlib.transforms.Affine2D method), 394
rotate_axes() (in module mpl_toolkits.mplot3d.axis3d), 517
rotate_deg() (matplotlib.transforms.Affine2D method), 394
rotate_deg_around() (matplotlib.transforms.Affine2D method), 395
rotated() (matplotlib.transforms.BboxBase method), 385
rrule (class in matplotlib.dates), 922
RRuleLocator (class in matplotlib.dates), 919
Rule (class in matplotlib.mathtext), 979
run() (matplotlib.cbook.Idle method), 871
run() (matplotlib.cbook.Timeout method), 873
safe_isinf() (in module matplotlib.mlab), 1001
safe_isnan() (in module matplotlib.mlab), 1001
safe_masked_invalid() (in module matplotlib.cbook), 877
safezip() (in module matplotlib.cbook), 877
Sankey (class in matplotlib.sankey), 1189
save() (matplotlib.animation.Animation method), 592
savefigure() (matplotlib.backend_bases.NavigationToolbar2 method), 855
savefig() (in module matplotlib.pylab), 1148
savefig() (matplotlib.backends.backend_pdf.PdfPages method), 863
savefig() (matplotlib.figure.Figure method), 940
scaling() (matplotlib.animation.MovieWriter method), 597
scale() (matplotlib.mlab), 1149
scale() (matplotlib.figure.Figure method), 941
ScalarFormatter (class in matplotlib.ticker), 1204
ScalarMappable (class in matplotlib.cm), 881
scale() (matplotlib.transforms.Affine2D method), 395
scale_factors (matplotlib.tri.TriAnalyzer attribute), 1222
scale_factory() (in module matplotlib.scale), 407
ScaleBase (class in matplotlib.scale), 405
Scaled (class in mpl_toolkits.axes_grid.axes_size), 457
scaled() (matplotlib.colors.Normalize method), 911
ScaledTranslation (class in matplotlib.transforms), 400
scatter() (in module matplotlib.pyplot), 1149
scatter() (matplotlib.axes.Axes method), 792
scatter() (mpl_toolkits.mplot3d.Axes3D method), 471
scatter() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 503
scatter3D() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 504
Scheduler (class in matplotlib.cbook), 872
sci() (in module matplotlib.pyplot), 1150
score_family() (matplotlib.font_manager.FontManager method), 948
score_size() (matplotlib.font_manager.FontManager method), 948
score_stretch() (matplotlib.font_manager.FontManager method), 948
score_style() (matplotlib.font_manager.FontManager method), 948
score_variant() (matplotlib.font_manager.FontManager method), 948
score_weight() (matplotlib.font_manager.FontManager method), 948
scroll_event() (matplotlib.backend_bases.FigureCanvasBase method), 847
SecondLocator (class in matplotlib.dates), 921
seconds() (in module matplotlib.dates), 923
segment_hits() (in module matplotlib.lines), 616
segments_intersect() (in module matplotlib.mlab), 1001
SELECT_REGION (matplotlib.backend_bases.Cursors attribute), 841
semilogx() (in module matplotlib.pyplot), 1151
semilogy() (matplotlib.axes.Axes method), 794
semilogy() (in module matplotlib.pyplot), 1152
set() (matplotlib.artist.Artist method), 602
set() (matplotlib.transforms.Affine2D method), 395
set() (matplotlib.transforms.Bbox method), 387
set() (matplotlib.transforms.TransformWrapper method), 392
set_3d_properties() (mpl_toolkits.mplot3d.art3d.Line3D method), 513
set_3d_properties() (mpl_toolkits.mplot3d.art3d.Patch3D method), 514
set_3d_properties() (mpl_toolkits.mplot3d.art3d.Patch3DCollection method), 514
set_3d_properties() (mpl_toolkits.mplot3d.art3d.PathPatch3D method), 515
set_3d_properties() (mpl_toolkits.mplot3d.art3d.Poly3DCollection method), 515
set_3d_properties() (mpl_toolkits.mplot3d.art3d.Text3D method), 516
set_aa() (matplotlib.lines.Line2D method), 612
set_aa() (matplotlib.patches.Patch method), 642
set_active() (matplotlib.widgets.RectangleSelector method), 1233
set_adjustable() (matplotlib.axes.Axes method), 797
set_agg_filter() (matplotlib.artist.Artist method), 602
set_alpha() (matplotlib.artist.Artist method), 602
set_alpha() (matplotlib.backend_bases.GraphicsContextBase method), 850
set_alpha() (matplotlib.collections.Collection method), 889
set_alpha() (matplotlib.colorbar.ColorbarBase method), 902
set_alpha() (matplotlib.patches.Patch method), 642
set_anchor() (matplotlib.axes.Axes method), 797
set_anchor() (mpl_toolkits.axes_grid.axes_divider.Divider method), 459
set_animated() (matplotlib.artist.Artist method), 603
set_antialiased() (matplotlib.backend_bases.GraphicsContextBase method), 629
set_antialiased() (matplotlib.collections.Collection method), 850
set_antialiased() (matplotlib.backend_bases.GraphicsContextBase method), 850
set_antialiased() (matplotlib.collections.Collection method), 889
set_antialiased() (matplotlib.patches.ConnectionPatch method), 612
set_antialiased() (matplotlib.patches.Patch method), 642
Index 1283
set_clip_rectangle() (matplotlib.backend_bases.GraphicsContextBase method), 850
set_closed() (matplotlib.patches.Polygon method), 645
set_cmap() (in module matplotlib.pyplot), 1154
set_cmap() (matplotlib.cm.ScalarMappable method), 882
set_color() (matplotlib.collections.Collection method), 889
set_color() (matplotlib.collections.LineCollection method), 895
set_color() (matplotlib.collections.LineCollection method), 895
set_color() (matplotlib.patches.Polygon method), 645
set_color() (matplotlib.patches.Patch method), 642
set_color() (matplotlib.patches.Patch method), 642
set_color() (matplotlib.patches.FancyArrowPatch method), 636
set_color_cycle() (matplotlib.axes.Axes method), 798
set_data() (matplotlib.lines.Line2D method), 613
set_data_interval() (matplotlib.axis.Axis method), 833
set_data_interval() (matplotlib.axis.XAxis method), 837
set_data_interval() (matplotlib.axis.YAxis method), 838
set_data_interval() (matplotlib.dates.MonthLocator method), 921
set_data_interval() (matplotlib.dates.MonthLocator method), 921
set_default_handler_map() (matplotlib.legend.Legend class method), 962
set_default_intervals() (matplotlib.axis.Axis method), 833
set_default_intervals() (matplotlib.axis.XAxis method), 837
set_default_intervals() (matplotlib.axis.YAxis method), 839
set_default_locators_and_formatters() (matplotlib.scale.LinearScale method), 405
set_default_locators_and_formatters() (matplotlib.scale.LogScale method), 405
set_default_locators_and_formatters() (matplotlib.scale.ScaleBase method), 406
set_default_locators_and_formatters() (matplotlib.scale.SymmetricalLogScale method), 406
set_default_weight() (matplotlib.font_manager.FontManager method), 949
set_dpi() (matplotlib.figure.Figure method), 941
set_dpi() (matplotlib.figure.Figure method), 941
set_dpi_cor() (matplotlib.patches.FancyArrowPatch method), 636
set_drawstyle() (matplotlib.lines.Line2D method), 613
set_edgecolor() (matplotlib.collections.Collection method), 889
set_edgecolor() (matplotlib.collections.Collection method), 889
set_edgecolor() (matplotlib.collections.Collection method), 889
set_edgecolor() (matplotlib.collections.Collection method), 889
set_edgecolor() (mpl_toolkits.mplot3d.art3d.Poly3DCollection method), 515
set_dashes() (matplotlib.backend_bases.GraphicsContextBase method), 850
set_dashes() (matplotlib.collections.Collection method), 899
set_dashes() (matplotlib.collections.Collection method), 899
set_dashes() (matplotlib.collections.Collection method), 899
set_dashes() (matplotlib.collections.Collection method), 899
set_dashes() (matplotlib.figure.Figure method), 941
set_dpi() (matplotlib.figure.Figure method), 941
set_dpi() (matplotlib.figure.Figure method), 941
set_dpi_cor() (matplotlib.patches.FancyArrowPatch method), 636
set_drawstyle() (matplotlib.lines.Line2D method), 613
set_ecc() (matplotlib.patches.Patch method), 642
set_edgecolor() (matplotlib.collections.Collection method), 889
set_edgecolor() (matplotlib.collections.Collection method), 889
set_edgecolor() (matplotlib.patches.Patch method), 642
set_edgecolor() (mpl_toolkits.mplot3d.art3d.Poly3DCollection method), 515
set_edgecolors() (matplotlib.collections.Collection method), 890
set_edgecolors() (mpl_toolkits.mplot3d.art3d.Poly3DCollection method), 515
set_facecolor() (matplotlib.collections.Collection method), 890
set_facecolor() (matplotlib.figure.Figure method), 941
set_facecolor() (matplotlib.patches.Patch method), 642
set_facecolor() (mpl_toolkits.mplot3d.art3d.Poly3DCollection method), 515
set_facecolors() (matplotlib.collections.Collection method), 890
set_facecolors() (mpl_toolkits.mplot3d.art3d.Poly3DCollection method), 515
set_facecolors() (mpl_toolkits.mplot3d.art3d.Poly3DCollection method), 515
set_family() (matplotlib.font_manager.FontProperties method), 950
set_family() (matplotlib.text.Text method), 659
set_fc() (matplotlib.patches.Patch method), 642
set_figheight() (matplotlib.figure.Figure method), 941
set_figure() (matplotlib.artist.Artist method), 603
set_figure() (matplotlib.axes.Axes method), 799
set_figure() (matplotlib.text.Annotation method), 655
set_figure() (matplotlib.text.TextWithDash method), 664
set_figwidth() (matplotlib.figure.Figure method), 941
set_file() (matplotlib.font_manager.FontProperties method), 950
set_fill() (matplotlib.patches.Patch method), 642
set_fillstyle() (matplotlib.lines.Line2D method), 613
set_fillstyle() (matplotlib.markers.MarkerStyle method), 965
set_font_properties() (matplotlib.text.Text method), 660
set_fontconfig_pattern() (matplotlib.font_manager.FontProperties method), 950
set_fontname() (matplotlib.text.Text method), 660
set_fontproperties() (matplotlib.text.Text method), 660
set_fontsize() (matplotlib.text.Text method), 660
set_fontstretch() (matplotlib.text.Text method), 660
set_fontstyle() (matplotlib.text.Text method), 660
set_fontvariant() (matplotlib.text.Text method), 660
set_fontweight() (matplotlib.text.Text method), 660
set_foreground() (matplotlib.text.Text method), 660
set_frame_on() (matplotlib.axes.Axes method), 799
set_frame_on() (matplotlib.legend.Legend method), 962
set_frame_on() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 505
set_frameon() (matplotlib.figure.Figure method), 941
set_gamma() (matplotlib.colors.LinearSegmentedColormap method), 909
set_grid() (matplotlib.artist.Artist method), 603
set_grid() (matplotlib.backend_bases.GraphicsContextBase method), 851
set_graylevel() (matplotlib.backend_bases.GraphicsContextBase method), 851
set_ha() (matplotlib.text.Text method), 660
set_hatch() (matplotlib.backend_bases.GraphicsContextBase method), 851
set_hatch() (matplotlib.collections.Collection method), 890
set_hatch() (matplotlib.collections.Collection method), 890
set_height() (matplotlib.patches.FancyBboxPatch method), 639
set_height() (matplotlib.patches.Rectangle method), 647
set_height_ratios() (matplotlib.gridspec.GridSpecBase method), 956
set_history_buttons() (matplotlib.backend_bases.NavigationToolbar2 method), 855
set_horizontal() (mpl_toolkits.axes_grid.axes_divider.Divider method), 459
set_horizontalalignment() (matplotlib.text.Text method), 660
set_joinstyle() (matplotlib.backend_bases.GraphicsContextBase method), 851
set_label() (matplotlib.artist.Artist method), 603
set_label() (matplotlib.axis.Tick method), 836
set_label() (matplotlib.colorbar.ColorbarBase method), 902
set_label() (mpl_toolkits.axes_grid.axis_artist.AxisArtist method), 464
set_label1() (matplotlib.axis.Tick method), 836
set_label2() (matplotlib.axis.Tick method), 836
set_label_coords() (matplotlib.axis.Axis method), 833
set_label_position() (matplotlib.axis.XAxis method), 837
set_label_position() (matplotlib.axis.YAxis method), 839
set_label_text() (matplotlib.axis.Axis method), 834
set_linelength() (matplotlib.collections.EventCollection method), 894
set_lineoffset() (matplotlib.collections.EventCollection method), 894
set_linespacing() (matplotlib.text.Text method), 660
set_linestyle() (matplotlib.collections.Collection method), 891
set_lw() (matplotlib.collections.Collection method), 891
set_lw() (matplotlib.lines.Line2D method), 614
set_lw() (matplotlib.patches.Patch method), 643
set_major_formatter() (matplotlib.axis.Axis method), 834
set_major_locator() (matplotlib.axis.Axis method), 834
set_major_locator() (matplotlib.axis.XAxis method), 834
set_major_locator() (matplotlib.axis.YAxis method), 834
set_marker() (matplotlib.lines.Line2D method), 614
set_marker() (matplotlib.patches.Patch method), 643
set_markeredgecolor() (matplotlib.lines.Line2D method), 614
set_markeredgewidth() (matplotlib.lines.Line2D method), 614
set_markerfacecolor() (matplotlib.lines.Line2D method), 614
set_markerfacecoloralt() (matplotlib.lines.Line2D method), 614
set_markersize() (matplotlib.lines.Line2D method), 614
set_markeredgecolor() (matplotlib.patches.Patch method), 643
set_markerfacecolor() (matplotlib.patches.Patch method), 643
set_markerfacecoloralt() (matplotlib.patches.Patch method), 643
set_minor_formatter() (matplotlib.axis.Axis method), 834
set_minor_locator() (matplotlib.axis.Axis method), 834
set_minor_locator() (matplotlib.axis.XAxis method), 834
set_minor_locator() (matplotlib.axis.YAxis method), 834
set_mask() (matplotlib.tri.Triangulation method), 1214
set_matrix() (matplotlib.transforms.Affine2D method), 395
set_mec() (matplotlib.lines.Line2D method), 614
set_message() (matplotlib.backend_bases.NavigationToolbar2 method), 855
set_mew() (matplotlib.lines.Line2D method), 614
set_mfc() (matplotlib.lines.Line2D method), 614
set_mfcalt() (matplotlib.lines.Line2D method), 614
set_multialignment() (matplotlib.text.Text method), 660
set_multialignment() (matplotlib.text.Text method), 660
set_matrix() (matplotlib.text.Text method), 660
set_mutation_aspect() (matplotlib.patches.FancyArrowPatch method), 636
set_mutation_aspect() (matplotlib.patches.FancyBboxPatch method), 636
set_multialignment() (matplotlib.text.Text method), 660
set_multialignment() (matplotlib.text.Text method), 660
set_multialignment() (matplotlib.patches.FancyArrowPatch method), 636
set_multialignment() (matplotlib.patches.FancyBboxPatch method), 636
set_mutation_scale() (matplotlib.patches.FancyArrowPatch method), 639
set_mutation_scale() (matplotlib.patches.FancyBboxPatch method), 636
set_name() (matplotlib.font_manager.FontProperties method), 951
set_name() (matplotlib.text.Text method), 660
set_navigate() (matplotlib.axes.Axes method), 799
set_navigate_mode() (matplotlib.axes.Axes method), 799
set_norm() (matplotlib.cm.ScalarMappable method), 882
set_offset_position() (matplotlib.axis.YAxis method), 839
set_offset_position() (matplotlib.collections.Collection method), 891
set_offset_string() (matplotlib.ticker.FixedFormatter method), 1204
set_offsets() (matplotlib.collections.Collection method), 891
set_orientation() (matplotlib.collections.EventCollection method), 894
set_over() (matplotlib.colors.Colormap method), 907
set_pad() (matplotlib.axis.Tick method), 836
set_pad() (mpl_toolkits.axes_grid.axis_artist.AxisLabel method), 465
set_pane_color() (mpl_toolkits.mplot3d.axis3d.Axis method), 512
set_pane_pos() (mpl_toolkits.mplot3d.axis3d.Axis method), 512
set_params() (matplotlib.ticker.MaxNLocator method), 1208
set_patch_circle() (matplotlib.spines.Spine method), 1199
set_patch_line() (matplotlib.spines.Spine method), 1199
set_patchA() (matplotlib.patches.FancyArrowPatch method), 636
set_patchB() (matplotlib.patches.FancyArrowPatch method), 636
set_path_effects() (matplotlib.artist.Artist method), 604
set_path() (matplotlib.collections.Collection method), 891
set_path() (matplotlib.collections.LineCollection method), 895
set_path() (matplotlib.collections.PatchCollection method), 896
set_path() (matplotlib.collections.PathCollection method), 896
set_path() (matplotlib.collections.PolyCollection method), 897
set_path() (matplotlib.collections.QuadMesh method), 898
set_path() (matplotlib.collections.TriMesh method), 900
set_picker() (matplotlib.artist.Artist method), 604
set_picker() (matplotlib.lines.Line2D method), 615
set_pickradius() (matplotlib.axis.Axis method), 834
set_pickradius() (matplotlib.collections.Collection method), 891
set_pickradius() (matplotlib.collections.LineCollection method), 615
set_pickradius() (matplotlib.collections.PatchCollection method), 615
set_pickradius() (matplotlib.collections.PolyCollection method), 615
set_powerlimits() (matplotlib.ticker.ScalarFormatter method), 1205
set_radius() (matplotlib.patches.Circle method), 626
set_radius() (matplotlib.patches.Wedge method), 651
set_rasterization_zorder() (matplotlib.axis.Axes method), 799
set_rasterized() (matplotlib.artist.Artist method), 604
set_rgrids() (matplotlib.projections.polar.PolarAxes method), 410
set_rotate_label() (mpl_toolkits.mplot3d.axis3d.Axis method), 512
set_rotation() (matplotlib.text.Text method), 661
set_rotation_mode() (matplotlib.text.Text method), 661
set_scale() (matplotlib.axis.Axis method), 834
set_scientific() (matplotlib.ticker.ScalarFormatter method), 1205
set_segments() (matplotlib.collections.LineCollection method), 895
set_segments() (mpl_toolkits.mplot3d.art3d.Line3DCollection method), 514
set_size() (matplotlib.font_manager.FontProperties method), 951
set_size() (matplotlib.text.Text method), 661
set_size_inches() (matplotlib.figure.Figure method), 942
set_sketch_params() (matplotlib.artist.Artist method), 604
set_sketch_params() (matplotlib.backend_bases.GraphicsContextBase method), 851
set_slant() (matplotlib.font_manager.FontProperties method), 951
set_smart_bounds() (matplotlib.axis.Axis method), 834
set_smart_bounds() (matplotlib.spines.Spine method), 1199
set_snap() (matplotlib.artist.Artist method), 604
set_snap() (matplotlib.backend_bases.GraphicsContextBase method), 851
set_solid_capstyle() (matplotlib.lines.Line2D method), 615
set_solid_joinstyle() (matplotlib.lines.Line2D method), 615
set_sort_zpos() (mpl_toolkits.mplot3d.axis3d.Line3DCollection method), 514
set_sort_zpos() (mpl_toolkits.mplot3d.art3d.Patch3DCollection method), 514
set_sort_zpos() (mpl_toolkits.mplot3d.art3d.Poly3DCollection method), 515
set_stretch() (matplotlib.font_manager.FontProperties method), 951
set_stretch() (matplotlib.text.Text method), 661
set_style() (matplotlib.font_manager.FontProperties method), 951
set_style() (matplotlib.text.Text method), 661
set_subplotspec() (matplotlib.axes.SubplotBase method), 830
set_subplotspec() (mpl_toolkits.axes_grid.axes_divider.SubplotDivider method), 460
set_text() (matplotlib.text.Text method), 661
set_theta1() (matplotlib.patches.Wedge method), 651
set_theta2() (matplotlib.patches.Wedge method), 651
set_theta_direction() (matplotlib.projections.polar.PolarAxes method), 411
set_theta_offset() (matplotlib.projections.polar.PolarAxes method), 412
set_theta_zero_location() (matplotlib.projections.polar.PolarAxes method), 412
set_thetagrids() (matplotlib.projections.polar.PolarAxes method), 412
set_tick_out() (mpl_toolkits.axes_grid.axis_artist.Ticks method), 465
set_tick_params() (matplotlib.axis.Axis method), 834
set_ticklabel_direction() (mpl_toolkits.axes_grid.axis_artist.AxisArtist method), 464
set_ticklabels() (matplotlib.axis.Axis method), 834
set_ticklabels() (matplotlib.colorbar.ColorbarBase method), 902
set_ticks() (matplotlib.axis.Axis method), 835
set_ticks() (matplotlib.colorbar.ColorbarBase method), 903
set_ticks_position() (matplotlib.axis.XAxis method), 837
set_ticks_position() (matplotlib.axis.YAxis method), 839
set_title() (matplotlib.axes.Axes method), 799
set_title() (matplotlib.legend.Legend method), 962
set_title() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 505
set_top_view() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 505
set_y() (matplotlib.text.TextWithDash method), 664
set_ybound() (matplotlib.axes.Axes method), 803
set_ydata() (matplotlib.lines.Line2D method), 614
set_ylabel() (matplotlib.axes.Axes method), 803
set_ylim() (matplotlib.axes.Axes method), 803
set_ylim3d() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 506
set_ymargin() (matplotlib.axes.Axes method), 804
set_yscale() (matplotlib.axes.Axes method), 804
set_yscale() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 507
set_yticklabels() (matplotlib.axes.Axes method), 804
set_yticks() (matplotlib.axes.Axes method), 806
set_zbound() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 507
set_zlabel() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 507
set_zlim() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 507
set_zlim3d() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 507
set_zmargin() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 508
set_zorder() (matplotlib.artist.Artist method), 605
set_zscale() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 508
set_zsort() (mpl_toolkits.mplot3d.art3d.Poly3DCollection method), 515
set_zticklabels() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 509
set_zticks() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 509
setp() (in module matplotlib.matplotlib.artist), 607
setp() (in module matplotlib.pyplot), 1154
setup() (matplotlib.animation.FileMovieWriter method), 594
setup() (matplotlib.animation.MovieWriter method), 594
show() (matplotlib.backend_bases.FigureManagerBase method), 848
show() (matplotlib.figure.Figure method), 942
show_popup() (matplotlib.backend_bases.FigureManagerBase method), 849
ShowBase (class in matplotlib.backend_bases), 859
shrink() (matplotlib.mathtext.Accent method), 969
shrink() (matplotlib.mathtext.Box method), 970
shrink() (matplotlib.mathtext.Char method), 970
shrink() (matplotlib.mathtext.Glue method), 972
shrink() (matplotlib.mathtext.Kern method), 973
shrink() (matplotlib.mathtext.List method), 974
shrink() (matplotlib.mathtext.Node method), 977
shrunk() (matplotlib.transforms.BboxBase method), 385
shrunk_to_aspect() (matplotlib.transforms.BboxBase method), 385
silent_list (class in matplotlib.cbook), 877
simple_group() (matplotlib.mathtext.Parser method), 979
simple_linear_interpolation() (in module matplotlib.pyplot), 1155
simplify_threshold (matplotlib.path.Path attribute), 1009
single_shot (matplotlib.backend_bases.TimerBase attribute), 859
size (matplotlib.dviread.DviFont attribute), 865
size (matplotlib.transforms.BboxBase attribute), 385
Slider (class in matplotlib.widgets), 1233
slopes() (in module matplotlib.mlab), 1001
sort() (matplotlib.cbook.SortedSeries method), 872
sort() (in module matplotlib.cbook), 872
soundex() (in module matplotlib.cbook), 878
space() (matplotlib.mathtext.Parser method), 979
span_where() (matplotlib.collections.BrokenBarHCollection static method), 886
SpanSelector (class in matplotlib.widgets), 1234
specgram() (in module matplotlib.mlab), 1002
specgram() (in module matplotlib.pyplot), 1155
specgram() (matplotlib.axes.Axes method), 806
spectral() (in module matplotlib.pyplot), 1155
spine() (in module matplotlib.collections), 1197
splitx() (matplotlib.transforms.BboxBase method), 386

Index 1291
splity() (matplotlib.transforms.BboxBase method), 386
spring() (in module matplotlib.pyplot), 1157
spy() (in module matplotlib.pyplot), 1157
spy() (matplotlib.axes.Axes method), 808
sqrt() (matplotlib.mathtext.Parser method), 979
SsGlue (class in matplotlib.mathtext), 980
Stack (class in matplotlib.cbook), 872
stackplot() (in module matplotlib.pyplot), 1158
stackplot() (matplotlib.axes.Axes method), 809
stackrel() (matplotlib.mathtext.Parser method), 979
StandardPsFonts (class in matplotlib.mathtext), 980
StarPolygonCollection (class in matplotlib.collections), 899
start() (matplotlib.backend_bases.TimerBase method), 859
start_event_loop() (matplotlib.backend_bases.FigureCanvasBase method), 847
start_event_loop_default() (matplotlib.backend_bases.FigureCanvasBase method), 847
start_filter() (matplotlib.backend_bases.RendererBase method), 858
start_group() (matplotlib.mathtext.Parser method), 979
start_pan() (matplotlib.axes.Axes method), 809
start_rasterizing() (matplotlib.backend_bases.RendererBase method), 858
stop() (matplotlib.backend_bases.TimerBase method), 860
stop_filter() (matplotlib.backend_bases.RendererBase method), 858
stop_rasterizing() (matplotlib.backend_bases.RendererBase method), 858
Stream (class in matplotlib.backends.backend_pdf), 863
streamplot() (in module matplotlib.pyplot), 1160
streamplot() (matplotlib.axes.Axes method), 811
strftime() (matplotlib.dates.DateFormatter method), 918
string_width_height() (matplotlib.afm.AFM method), 588
strip_math() (in module matplotlib.cbook), 878
strip_math() (matplotlib.backend_bases.RendererBase method), 858
Subplot (in module matplotlib.axes), 829
subplot() (in module matplotlib.pyplot), 1161
subplot2grid() (in module matplotlib.pyplot), 1163
subplot_class_factory() (in module matplotlib.axes), 830
subplot_tool() (in module matplotlib.pyplot), 829
SubplotDivider (class in mpl_toolkits.axes_grid.axes_divider), 460
SubplotParams (class in matplotlib.figure), 944
subplots() (in module matplotlib.pyplot), 1163
subplots_adjust() (in module matplotlib.pyplot), 1165
subplots_adjust() (matplotlib.figure.Figure method), 942
SubplotSpec (class in matplotlib.gridspec), 956
SubplotTool (class in matplotlib.widgets), 1235
subs() (matplotlib.ticker.LogLocator method), 1207
subsuper() (matplotlib.mathtext.Parser method), 979
SubSuperCluster (class in matplotlib.mathtext), 981
summer() (in module matplotlib.pyplot), 1165
supported_formats (matplotlib.animation.FFMpegFileWriter attribute), 593
supported_formats (matplotlib.animation.ImageMagickFileWriter attribute), 594
supported_formats (matplotlib.animation.MencoderFileWriter attribute), 595
supports_blit (matplotlib.backend_bases.FigureCanvasBase attribute), 848
supuptitle() (in module matplotlib.pyplot), 1165
supuptitle() (matplotlib.figure.Figure method), 942
SVG, 1239
switch_backend() (in module matplotlib.pyplot), 1165
switch_backends() (matplotlib.backend_bases.FigureCanvasBase method), 848
switch_orientation() (matplotlib.collections.EventCollection method), 894
symbol() (matplotlib.mathtext.Parser method), 979
SymLogNorm (class in matplotlib.colors), 911
SymmetricalLogScale (class in matplotlib.scale), 406
T

table() (in module matplotlib.pyplot), 1166
table() (matplotlib.axes.Axes method), 811
target (matplotlib.mathtext.BakomaFonts attribute), 970
test_lines_dists() (in module mpl_toolkits.mplot3d.proj3d), 518
test_proj() (in module mpl_toolkits.mplot3d.proj3d), 518
test_proj_draw_axes() (in module mpl_toolkits.mplot3d.proj3d), 518
test_proj_make_M() (in module mpl_toolkits.mplot3d.proj3d), 518
test_rot() (in module mpl_toolkits.mplot3d.proj3d), 518
test_world() (in module mpl_toolkits.mplot3d.proj3d), 518
texname (matplotlib.dviread.DviFont attribute), 865
Text (class in matplotlib.text), 656
text() (in module matplotlib.pyplot), 1166
text() (matplotlib.axes.Axes method), 812
text() (matplotlib.figure.Figure method), 943
text() (mpl_toolkits.mplot3d.Axes3D method), 488
text() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 509
text2D() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 509
Text3D (class in mpl_toolkits.mplot3d.art3d), 516
text3D() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 510
text_2d_to_3d() (in module mpl_toolkits.mplot3d.art3d), 517
TextWithDash (class in matplotlib.text), 662
Tfm (in module matplotlib.dviread), 866
ThetaFormatter (class in matplotlib.projections.polar), 414
thetagrids() (in module matplotlib.pyplot), 1167
Tick (class in matplotlib.axis), 835
tick_bottom() (matplotlib.axis.xaxis method), 837
tick_left() (matplotlib.axis.yaxis method), 839
tick_params() (in module matplotlib.pyplot), 1168
tick_params() (matplotlib.axes.Axes method), 813
tick_params() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 510
tick_right() (matplotlib.axis.xaxis method), 839
tick_top() (matplotlib.axis.xaxis method), 838
tick_update_position() (in module mpl_toolkits.mplot3d.axes3d), 513
tick_values() (matplotlib.ticker.AutoMinorLocator method), 1209
tick_values() (matplotlib.ticker.FixedLocator method), 1207
tick_values() (matplotlib.ticker.IndexLocator method), 1207
tick_values() (matplotlib.ticker.LinearLocator method), 1207
tick_values() (matplotlib.ticker.Locator method), 1206
tick_values() (matplotlib.ticker.LogLocator method), 1208
tick_values() (matplotlib.ticker.MaxNLocator method), 1208
tick_values() (matplotlib.ticker.MultipleLocator method), 1208
tick_values() (matplotlib.ticker.NullLocator method), 1207
Ticker (class in matplotlib.axis), 836
TickHelper (class in matplotlib.ticker), 1202
ticklabel_format() (in module matplotlib.pyplot), 1168
ticklabel_format() (matplotlib.axes.Axes method), 813
ticklabel_format() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 510
TickLabels (class in mpl_toolkits.mplot3d.axes_grid.axis_artist), 465
<table>
<thead>
<tr>
<th>Function</th>
<th>Module/Class/Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ticks</td>
<td>mpl_toolkits.axes_grid.axis_artist</td>
</tr>
<tr>
<td>TIFF</td>
<td></td>
</tr>
<tr>
<td>tight_layout()</td>
<td>matplotlib.pyplot.figure.Figure</td>
</tr>
<tr>
<td>tight_layout()</td>
<td>matplotlib.gridspec.GridSpec</td>
</tr>
<tr>
<td>TimedAnimation</td>
<td>matplotlib.animation</td>
</tr>
<tr>
<td>Timeout</td>
<td>matplotlib.cbook</td>
</tr>
<tr>
<td>Title</td>
<td>matplotlib.pyplot</td>
</tr>
<tr>
<td>Tk</td>
<td>matplotlib.cbook</td>
</tr>
<tr>
<td>to_filehandle()</td>
<td>matplotlib.cbook</td>
</tr>
<tr>
<td>to_mask()</td>
<td>matplotlib.mathtext.MathTextParser</td>
</tr>
<tr>
<td>to_png()</td>
<td>matplotlib.mathtext.MathTextParser</td>
</tr>
<tr>
<td>to_polygons()</td>
<td>matplotlib.path.Path</td>
</tr>
<tr>
<td>to_rgb()</td>
<td>matplotlib.colors.ColorConverter</td>
</tr>
<tr>
<td>to_rgba()</td>
<td>matplotlib.cm.ScalarMappable</td>
</tr>
<tr>
<td>to_rgba()</td>
<td>matplotlib.colors.ColorConverter</td>
</tr>
<tr>
<td>to_rgba()</td>
<td>matplotlib.mathtext.MathTextParser</td>
</tr>
<tr>
<td>to_values()</td>
<td>matplotlib.transforms.Affine2DBase</td>
</tr>
<tr>
<td>todate</td>
<td>matplotlib.cbook</td>
</tr>
<tr>
<td>todatet ime()</td>
<td>matplotlib.cbook</td>
</tr>
<tr>
<td>tofloat</td>
<td>matplotlib.cbook</td>
</tr>
<tr>
<td>toggle()</td>
<td>mpl_toolkits.axes_grid.axis_artist.AxesArtist</td>
</tr>
<tr>
<td>toint</td>
<td>mpl_toolkits.axes_grid.axis_artist</td>
</tr>
<tr>
<td>toolitems</td>
<td>matplotlib.backend_bases.NavigationToolbar2</td>
</tr>
<tr>
<td>tostr()</td>
<td>matplotlib.mlab.FormatFormatStr</td>
</tr>
<tr>
<td>tostr()</td>
<td>matplotlib.mlab.FormatInt</td>
</tr>
<tr>
<td>tostr()</td>
<td>matplotlib.mlab.FormatObj</td>
</tr>
<tr>
<td>tostr()</td>
<td>matplotlib.mlab.FormatString</td>
</tr>
<tr>
<td>toval()</td>
<td>matplotlib.mlab.FormatDate</td>
</tr>
<tr>
<td>toval()</td>
<td>matplotlib.mlab.FormatFloat</td>
</tr>
<tr>
<td>toval()</td>
<td>matplotlib.mlab.FormatInt</td>
</tr>
<tr>
<td>transform()</td>
<td>matplotlib.transforms.AffineBase</td>
</tr>
<tr>
<td>transform()</td>
<td>matplotlib.transforms.IdentityTransform</td>
</tr>
<tr>
<td>transform()</td>
<td>matplotlib.transforms.Transform</td>
</tr>
<tr>
<td>transform()</td>
<td>matplotlib.transforms.CompositeGenericTransform</td>
</tr>
<tr>
<td>transform_non_affine()</td>
<td>matplotlib.projections.polar.InvertedPolarTransform</td>
</tr>
<tr>
<td>transform_non_affine()</td>
<td>matplotlib.projections.polar.PolarAxes.InvertedPolarTransform</td>
</tr>
<tr>
<td>transform_non_affine()</td>
<td>matplotlib.projections.polar.PolarAxes.PolarTransform</td>
</tr>
<tr>
<td>transform_non_affine()</td>
<td>matplotlib.transformations.AffineBase</td>
</tr>
<tr>
<td>transform_non_affine()</td>
<td>matplotlib.transformations.IdentityTransform</td>
</tr>
<tr>
<td>transform_non_affine()</td>
<td>matplotlib.transformations.Transform</td>
</tr>
<tr>
<td>transform_non_affine()</td>
<td>matplotlib.transformations.CompositeGenericTransform</td>
</tr>
<tr>
<td>transform_non_affine()</td>
<td>matplotlib.projections.polar.PolarAxes.InvertedPolarTransform</td>
</tr>
<tr>
<td>transform_non_affine()</td>
<td>matplotlib.projections.polar.PolarAxes.PolarTransform</td>
</tr>
<tr>
<td>transform_non_affine()</td>
<td>matplotlib.transformations.AffineBase</td>
</tr>
<tr>
<td>transform_non_affine()</td>
<td>matplotlib.transformations.BlendedGenericTransform</td>
</tr>
</tbody>
</table>
transform_non_affine() (matplotlib.transforms.CompositeGenericTransform method), 398
transform_non_affine() (matplotlib.transforms.IdentityTransform method), 396
transform_non_affine() (matplotlib.transforms.Transform method), 391
transform_path() (matplotlib.transforms.AffineBase method), 392
transform_path() (matplotlib.transforms.Transform method), 396
transform_path_affine() (matplotlib.transforms.AffineBase method), 392
transform_path_affine() (matplotlib.transforms.IdentityTransform method), 396
transform_path_non_affine() (matplotlib.projections.polar.PolarAxes.PolarTransform method), 409
transform_path_non_affine() (matplotlib.projections.polar.PolarsTransform method), 414
transform_path_non_affine() (matplotlib.transforms.AffineBase method), 393
transform_path_non_affine() (matplotlib.transforms.CompositeGenericTransform method), 399
transform_path_non_affine() (matplotlib.transforms.IdentityTransform method), 396
transform_path_non_affine() (matplotlib.transforms.Transform method), 391
transform_point() (matplotlib.transforms.Affine2DBase method), 395
transform_point() (matplotlib.transforms.Transform method), 391
transformed() (matplotlib.path.Path method), 1010
transformed() (matplotlib.transforms.BboxBase method), 386
TransformedBbox (class in matplotlib.transforms), 388
TransformedPath (class in matplotlib.transforms), 400
TransformNode (class in matplotlib.transforms), 382
TransformWrapper (class in matplotlib.transforms), 391
translate() (matplotlib.transforms.Affine2D method), 395
translated() (matplotlib.transforms.BboxBase method), 386
transmute() (matplotlib.patches.ArrowStyle.Fancy method), 623
transmute() (matplotlib.patches.ArrowStyle.Simple method), 623
transmute() (matplotlib.patches.ArrowStyle.Wedge method), 623
transmute() (matplotlib.patches.BoxStyle.LArrow method), 625
transmute() (matplotlib.patches.BoxStyle.RArrow method), 625
transmute() (matplotlib.patches.BoxStyle.Round method), 625
transmute() (matplotlib.patches.BoxStyle.Round4 method), 625
transmute() (matplotlib.patches.BoxStyle.Roundtooth method), 625
transmute() (matplotlib.patches.BoxStyle.Sawtooth method), 625
transmute() (matplotlib.patches.BoxStyle.Square method), 626
TrapezoidMapTriFinder (class in matplotlib.tri), 1214
TriAnalyzer (class in matplotlib.tri), 1220
Triangulation (class in matplotlib.tri), 1213
tricontour() (in module matplotlib.pyplot), 1169
tricontour() (matplotlib.axes.Axes method), 814
tricontour() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 510
tricontourf() (in module matplotlib.pyplot), 1173
tricontourf() (matplotlib.axes.Axes method), 818
tricontourf() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 511
Index

TriFinder (class in matplotlib.tri), 1214
TriInterpolat or (class in matplotlib.tri), 1214
TriMesh (class in matplotlib.collections), 899
tripcolor() (in module matplotlib.pyplot), 1177
tripcolor() (matplotlib.axes.Axes method), 822
triplot() (in module matplotlib.pyplot), 1180
triplot() (matplotlib.axes.Axes method), 824
TriRefiner (class in matplotlib.tri), 1218
TruetypeFonts (class in matplotlib.mathtext), 981
TruetypeFonts.CachedFont (class in matplotlib.mathtext), 981
tfdict_to_fnames() (in module matplotlib.font_manager), 952
ttfFontProperty() (in module matplotlib.font_manager), 952
tunit_cube() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 511
tunit_edges() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 511
twinx() (in module matplotlib.pyplot), 1181
twinx() (matplotlib.axes.Axes method), 826
twiny() (in module matplotlib.pyplot), 1181
twiny() (matplotlib.axes.Axes method), 826
Type1Font (class in matplotlib.type1font), 867

U
unchr_safe() (in module matplotlib.mathtext), 982
unicode_safe() (in module matplotlib.cbook), 878
UnicodeFonts (class in matplotlib.mathtext), 981
UniformTriRefiner (class in matplotlib.tri), 1218
union() (matplotlib.transforms.BboxBase static method), 386
unique() (in module matplotlib.cbook), 878
unit() (matplotlib.transforms.Bbox static method), 387
unit_bbox() (in module mpl_toolkits.mplot3d.axes3d), 512
unit_circle() (matplotlib.path.Path class method), 1010
unit_circle_righthalf() (matplotlib.path.Path class method), 1010
unit_cube() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 511
unit_rectangle() (matplotlib.path.Path class method), 1010
unit_regular_polygon() (matplotlib.path.Path class method), 1010
unit_regular_star() (matplotlib.path.Path class method), 1010
unknown_symbol() (matplotlib.mathtext.Parser method), 979
unmasked_index_ranges() (in module matplotlib.cbook), 878
update() (matplotlib.artist.Artist method), 605
update() (matplotlib.backend_bases.NavigationToolbar2 method), 855
update() (matplotlib.figure.SubplotParams method), 945
update() (matplotlib.gridspec.GridSpec method), 955
update() (matplotlib.widgets.RectangleSelector method), 1233
update() (matplotlib.widgets.SpanSelector method), 1235
update_background() (matplotlib.widgets.LassoSelector method), 1230
update_background() (matplotlib.widgets.RectangleSelector method), 1233
update_background() (matplotlib.widgets.SpanSelector method), 1235
update_bbox_position_size() (matplotlib.colorbar.Colorbar method), 901
update_coords() (matplotlib.text.TextWithDash method), 664
update_datalim() (matplotlib.axes.Axes method), 826
update_datalim() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 511
update_datalim_bounds() (matplotlib.axes.Axes method), 827
update_datalim_numerix() (matplotlib.axes.Axes method), 827
update_datalim_to_current() (matplotlib.mlab.FIFOBu

1296
update_fonts()  (matplotlib.font_manager.FontManager method), 949
update_from()  (matplotlib.artist.Artist method), 605
update_from()  (matplotlib.collections.Collection method), 891
update_from()  (matplotlib.lines.Line2D method), 615
update_from()  (matplotlib.patches.Patch method), 643
update_from()  (matplotlib.text.Text method), 662
update_from_data()  (matplotlib.transforms.Bbox method), 387
update_from_data_xy()  (matplotlib.transforms.Bbox method), 388
update_from_path()  (matplotlib.transforms.Bbox method), 388
update_normal()  (matplotlib.colorbar.Colorbar method), 901
update_params()  (matplotlib.axes.SubplotBase method), 830
update_params()  (mpl_toolkits.axes_grid.axes_divider.SubplotDivider method), 460
update_position()  (matplotlib.axis.XTick method), 838
update_position()  (matplotlib.axis.YTick method), 839
update_positions()  (matplotlib.text.Annotation method), 655
update_scalarmappable()  (matplotlib.collections.Collection method), 891
update_ticks()  (matplotlib.colorbar.ColorbarBase method), 903
update_units()  (matplotlib.axis.Axis method), 835
use()  (in module matplotlib), 583
use_cmex (matplotlib.mathtext.StixFonts attribute), 981
use_cmex (matplotlib.mathtext.UnicodeFonts attribute), 982
useLocale (matplotlib.ticker.ScalarFormatter attribute), 1205
useOffset (matplotlib.ticker.ScalarFormatter attribute), 1205
validCap (matplotlib.lines.Line2D attribute), 615
validJoin (matplotlib.lines.Line2D attribute), 615
value_escape()  (in module matplotlib.fontconfig_pattern), 953
value_unescape()  (in module matplotlib.fontconfig_pattern), 953
Vbox (class in matplotlib.mathtext), 982
VCentered (class in matplotlib.mathtext), 982
vec_pad_ones()  (in module mpl_toolkits.mplot3d.proj3d), 518
vector graphics, 1240
vector_lengths()  (in module matplotlib.mlab), 1003
VertexSelector (class in matplotlib.lines), 615
vertices (matplotlib.path.Path attribute), 1010
Vf (class in matplotlib.dviread), 866
view_init()  (mpl_toolkits.mplot3d.axes3d.Axes3D method), 511
view_limits()  (matplotlib.ticker.LinearLocator method), 1207
view_limits()  (matplotlib.ticker.LogLocator method), 1208
view_limits()  (matplotlib.ticker.MaxNLocator method), 1208
view_limits()  (matplotlib.ticker.MultipleLocator method), 1208
view_transformation()  (in module mpl_toolkits.mplot3d.proj3d), 518
viewlim_to_dt()  (matplotlib.dates.DateLocator method), 919
vlines()  (in module matplotlib.pyplot), 1181
vlines()  (matplotlib.axes.Axes method), 827
Vlist (class in matplotlib.mathtext), 982
Vrule (class in matplotlib.mathtext), 982
W
waitforbuttonpress()  (in module matplotlib.pyplot), 1182
waitforbuttonpress()  (matplotlib.figure.Figure method), 944
waittime (matplotlib.cbook.Idle attribute), 871
warn_deprecated()  (in module matplotlib.cbook), 879
Wedge (class in matplotlib.patches), 650
wedge()  (matplotlib.path.Path class method), 1010
WeekdayLocator (class in matplotlib.dates), 920

Index 1297
weeks() (in module matplotlib.dates), 923
weight_as_number() (in module matplotlib.font_manager), 952
Widget (class in matplotlib.widgets), 1236
width (matplotlib.dviread.Tfm attribute), 866
width (matplotlib.transforms.BboxBase attribute), 386
widths (matplotlib.dviread.DviFont attribute), 865
win32FontDirectory() (in module matplotlib.font_manager), 952
win32InstalledFonts() (in module matplotlib.font_manager), 952
window_hanning() (in module matplotlib.mlab), 1003
window_none() (in module matplotlib.mlab), 1004
winter() (in module matplotlib.pyplot), 1182
world_transformation() (in module mpl_toolkits.mplot3d.proj3d), 518
wrap() (in module matplotlib.cbook), 879
write() (matplotlib.backends.backend_pdf.Stream method), 864
writeInfoDict() (matplotlib.backends.backend_pdf.PdfFile method), 862
writeTrailer() (matplotlib.backends.backend_pdf.PdfFile method), 862
writeXref() (matplotlib.backends.backend_pdf.PdfFile method), 863
wxpython, 1240
wxWidgets, 1240

X
x (matplotlib.backend_bases.LocationEvent attribute), 852
x (matplotlib.backend_bases.MouseEvent attribute), 852
x0 (matplotlib.transforms.BboxBase attribute), 386
x1 (matplotlib.transforms.BboxBase attribute), 386
XAxis (class in matplotlib.axis), 837
XAxis (class in mpl_toolkits.mplot3d.axes3d), 513
xaxis_date() (matplotlib.axes.Axes method), 827
xaxis_inverted() (matplotlib.axes.Axes method), 829
xcorr() (in module matplotlib.pyplot), 1182
xcorr() (matplotlib.axes.Axes method), 828
xdata (matplotlib.backend_bases.LocationEvent attribute), 852
xdata (matplotlib.backend_bases.MouseEvent attribute), 853
xkcd() (in module matplotlib.pyplot), 1184
xlabel() (in module matplotlib.pyplot), 1184
xlat() (matplotlib.cbook.Xlator method), 873
Xlator (class in matplotlib.cbook), 873
xlim() (in module matplotlib.pyplot), 1185
xmax (matplotlib.transforms.BboxBase attribute), 386
xmin (matplotlib.transforms.BboxBase attribute), 386
yscale() (in module matplotlib.pyplot), 1187
xticks() (in module matplotlib.pyplot), 1186
xscale() (in module matplotlib.pyplot), 1185
XTick (class in matplotlib.axis), 838
xticks() (in module matplotlib.pyplot), 1186
xy (matplotlib.patches.Polygon attribute), 645
xy (matplotlib.patches.Rectangle attribute), 647
xy (matplotlib.patches.RegularPolygon attribute), 649
xy() (matplotlib.cbook.MemoryMonitor method), 871

Y
y (matplotlib.backend_bases.LocationEvent attribute), 852
y (matplotlib.backend_bases.MouseEvent attribute), 853
y0 (matplotlib.transforms.BboxBase attribute), 386
y1 (matplotlib.transforms.BboxBase attribute), 386
YAArrow (class in matplotlib.patches), 651
YAxis (class in matplotlib.axis), 838
YAxis (class in mpl_toolkits.mplot3d.axes3d), 513
yaxis_date() (matplotlib.axes.Axes method), 829
yaxis_inverted() (matplotlib.axes.Axes method), 829
ydata (matplotlib.backend_bases.LocationEvent attribute), 852
ydata (matplotlib.backend_bases.MouseEvent attribute), 853
YearLocator (class in matplotlib.dates), 920
ylabel() (in module matplotlib.pyplot), 1186
ylim() (in module matplotlib.pyplot), 1186
ymin (matplotlib.transforms.BboxBase attribute), 386
yscale() (in module matplotlib.pyplot), 1187
YTick (class in matplotlib.axis), 839
yticks() (in module matplotlib.pyplot), 1187
Z
zalpha() (in module mpl_toolkits.mplot3d.art3d), 517
ZAxis (class in mpl_toolkits.mplot3d.axis3d), 513
zaxis_date() (mpl_toolkits.mplot3d.axes3d.Axes3D
method), 511
zaxis_inverted() (mpl_toolkits.mplot3d.axes3d.Axes3D
method), 511
zoom() (matplotlib.axis.Axis method), 835
zoom() (matplotlib.backend_bases.NavigationToolbar2
method), 855
zoom() (matplotlib.ticker.Locator method), 1206
zorder (matplotlib.artist.Artist attribute), 605
zorder (matplotlib.legend.Legend attribute), 962
zorder (matplotlib.lines.Line2D attribute), 615
zorder (matplotlib.patches.Patch attribute), 643
zorder (matplotlib.text.Text attribute), 662
ZORDER (mpl_toolkits.axes_grid.axis_artist.AxisArtist
attribute), 463