Matplotlib tutorial
Nicolas Rougier

To cite this version:

HAL Id: cel-00907344
https://cel.archives-ouvertes.fr/cel-00907344
Submitted on 21 Nov 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers. L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Matplotlib tutorial
Nicolas P. Rougier - Euroscipy 2012 - Prace 2013 - Euroscipy 2013

Introduction
Simple plot
Figures, Subplots, Axes and Ticks
Other Types of Plots
Beyond this tutorial
Quick references

This tutorial is based on Mike Müller's tutorial available from the scipy lecture notes.

Sources are available here. Figures are in the figures directory and all scripts are located in the scripts directory. Github repository is here

All code and material is licensed under a Creative Commons Attribution 3.0 United States License (CC-by)
http://creativecommons.org/licenses/by/3.0/us

Many thanks to Bill Wing and Christoph Deil for review and corrections.

Introductory slides on scientific visualization are here

Introduction

matplotlib is probably the single most used Python package for 2D-graphics. It provides both a very quick way to visualize data from Python and publication-quality figures in many formats. We are going to explore matplotlib in interactive mode covering most common cases.
IPython and the pylab mode

IPython is an enhanced interactive Python shell that has lots of interesting features including named inputs and outputs, access to shell commands, improved debugging and many more. When we start it with the command line argument -pylab (--pylab since IPython version 0.12), it allows interactive matplotlib sessions that have Matlab/Mathematica-like functionality.

pylab

pylab provides a procedural interface to the matplotlib object-oriented plotting library. It is modeled closely after Matlab(TM). Therefore, the majority of plotting commands in pylab have Matlab(TM) analogs with similar arguments. Important commands are explained with interactive examples.

Simple plot

In this section, we want to draw the cosine and sine functions on the same plot. Starting from the default settings, we'll enrich the figure step by step to make it nicer.

First step is to get the data for the sine and cosine functions:

```python
from pylab import *
X = np.linspace(-np.pi, np.pi, 256, endpoint=True)
C,S = np.cos(X), np.sin(X)
```

X is now a numpy array with 256 values ranging from -\(\pi\) to +\(\pi\) (included). C is the cosine (256 values) and S is the sine (256 values).

To run the example, you can type them in an IPython interactive session

```bash
$ ipython --pylab
```

This brings us to the IPython prompt:
Welcome to pylab, a matplotlib-based Python environment.
For more information, type `help(pylab)`.

or you can download each of the examples and run it using regular python:

```
$ python exercice_1.py
```

You can get source for each step by clicking on the corresponding figure.

---

**Using defaults**

Matplotlib comes with a set of default settings that allow customizing all kinds of properties. 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. While matplotlib defaults are rather good in most cases, you may want to modify some properties for specific cases.

```
from pylab import *
X = np.linspace(-np.pi, np.pi, 256,endpoint=True)
C,S = np.cos(X), np.sin(X)
plot(X,C)
plot(X,S)
show()
```
Instantiating defaults

In the script below, we've instantiated (and commented) all the figure settings that influence the appearance of the plot. The settings have been explicitly set to their default values, but now you can interactively play with the values to explore their affect (see Line properties and Line styles below).

```python
# Import everything from matplotlib (numpy is accessible via 'np' alias)
from pylab import *

# Create a new figure of size 8x6 points, using 80 dots per inch
figure(figsize=(8,6), dpi=80)

# Create a new subplot from a grid of 1x1
subplot(1,1,1)

X = np.linspace(-np.pi, np.pi, 256,endpoint=True)
C,S = np.cos(X), np.sin(X)

# Plot cosine using blue color with a continuous line of width 1 (pixels)
plot(X, C, color="blue", linewidth=1.0, linestyle="-")

# Plot sine using green color with a continuous line of width 1 (pixels)
plot(X, S, color="green", linewidth=1.0, linestyle="-")

# Set x limits
xlim(-4.0,4.0)

# Set x ticks
xticks(np.linspace(-4,4,9,endpoint=True))

# Set y limits
ylim(-1.0,1.0)

# Set y ticks
```
Changing colors and line widths

First step, we want to have the cosine in blue and the sine in red and a slighty thicker line for both of them. We'll also slightly alter the figure size to make it more horizontal.

```python
... figure(figsize=(10,6), dpi=80)
plot(X, C, color="blue", linewidth=2.5, linestyle="-")
plot(X, S, color="red", linewidth=2.5, linestyle="-")
...```

Setting limits

Current limits of the figure are a bit too tight and we want to make some space in order to clearly see all data points.

```python
... xlim(X.min()*1.1, X.max()*1.1)
ylim(C.min()*1.1, C.max()*1.1)
...```
Setting ticks

Current ticks are not ideal because they do not show the interesting values (+/-\(\pi\), +/-\(\pi/2\)) for sine and cosine. We'll change them such that they show only these values.

```python
...
xticks([-np.pi, -np.pi/2, 0, np.pi/2, np.pi])
yticks([-1, 0, +1])
...
```

Setting tick labels

Ticks are now properly placed but their label is not very explicit. We could guess that 3.142 is \(\pi\) but it would be better to make it explicit.
When we set tick values, we can also provide a corresponding label in the second argument list. Note that we'll use LaTeX to allow for nice rendering of the label.

```python
... xticks([-np.pi, -np.pi/2, 0, np.pi/2, np.pi],
          [r'$-\pi$', r'$-\pi/2$', r'$0$', r'$+\pi/2$', r'$+\pi$'])

yticks([-1, 0, +1],
          [r'$-1$', r'$0$', r'$+1$'])
...```

Moving spines

Spines are the lines connecting the axis tick marks and noting the boundaries of the data area. They can be placed at arbitrary positions and until now, they were on the border of the axis. We'll change that since we want to have them in the middle. Since there are four of them
(top/bottom/left/right), we'll discard the top and right by setting their color to none and we'll move the bottom and left ones to coordinate 0 in data space coordinates.

```python
...
ax = gca()
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.xaxis.set_ticks_position('bottom')
ax.spines['bottom'].set_position(('data',0))
ax.yaxis.set_ticks_position('left')
ax.spines['left'].set_position(('data',0))
...
```

### Adding a legend

![Legend example](image)

Let's add a legend in the upper left corner. This only requires adding the keyword argument label (that will be used in the legend box) to the plot commands.

```python
...
plot(X, C, color="blue", linewidth=2.5, linestyle="-", label="cosine")
plot(X, S, color="red", linewidth=2.5, linestyle="-", label="sine")
legend(loc='upper left')
...
```

### Annotate some points

Let's annotate some interesting points using the annotate command.
We chose the $2\pi/3$ value and we want to annotate both the sine and the cosine. We'll first draw a marker on the curve as well as a straight dotted line. Then, we’ll use the annotate command to display some text with an arrow.

```python
...  
t = 2*np.pi/3
plot([t,t],[0,np.cos(t)], color = 'blue', linewidth=2.5, linestyle="--")
scatter([t,],[np.cos(t),], 50, color = 'blue')
annotate(r'$\sin(\frac{2\pi}{3})=\frac{\sqrt{3}}{2}$',
        xy=(t, np.sin(t)), xycoords='data',
        xytext=(+10, +30), textcoords='offset points', fontsize=16,
        arrowprops=dict(arrowstyle="->", connectionstyle="arc3,rad=.2"))
plot([t,t],[0,np.sin(t)], color = 'red', linewidth=2.5, linestyle="--")
scatter([t,],[np.sin(t),], 50, color = 'red')
annotate(r'$\cos(\frac{2\pi}{3})=-\frac{1}{2}$',
        xy=(t, np.cos(t)), xycoords='data',
        xytext=(-90, -50), textcoords='offset points', fontsize=16,
        arrowprops=dict(arrowstyle="->", connectionstyle="arc3,rad=.2"))
...`

**Devil is in the details**

The tick labels are now hardly visible because of the blue and red lines. We can make them bigger and we can also adjust their properties such that they’ll be rendered on a semi-transparent white background. This will allow us to see both the data and the labels.

```
Figures, Subplots, Axes and Ticks

So far we have used implicit figure and axes creation. This is handy for fast plots. We can have more control over the display using figure, subplot, and axes explicitly. A figure in matplotlib means the whole window in the user interface. Within this figure there can be subplots. While subplot positions the plots in a regular grid, axes allows free placement within the figure. Both can be useful depending on your intention. We've already worked with figures and subplots without explicitly calling them. When we call plot, matplotlib calls gca() to get the current axes and gca in turn calls gcf() to get the current figure. If there is none it calls figure() to make one, strictly speaking, to make a subplot(111). Let's look at the details.

Figures

A figure is the windows in the GUI that has "Figure #" as title. Figures are numbered starting from 1 as opposed to the normal Python way starting from 0. This is clearly MATLAB-style. There are several parameters that determine what the figure looks like:
<table>
<thead>
<tr>
<th>Argument</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>num</td>
<td>1</td>
<td>number of figure</td>
</tr>
<tr>
<td>figsize</td>
<td>figure.figsize</td>
<td>figure size in in inches (width, height)</td>
</tr>
<tr>
<td>dpi</td>
<td>figure.dpi</td>
<td>resolution in dots per inch</td>
</tr>
<tr>
<td>facecolor</td>
<td>figure.facecolor</td>
<td>color of the drawing background</td>
</tr>
<tr>
<td>edgecolor</td>
<td>figure.edgecolor</td>
<td>color of edge around the drawing background</td>
</tr>
<tr>
<td>frameon</td>
<td>True</td>
<td>draw figure frame or not</td>
</tr>
</tbody>
</table>

The defaults can be specified in the resource file and will be used most of the time. Only the number of the figure is frequently changed.

When you work with the GUI you can close a figure by clicking on the x in the upper right corner. But you can close a figure programmatically by calling close. Depending on the argument it closes (1) the current figure (no argument), (2) a specific figure (figure number or figure instance as argument), or (3) all figures (all as argument).

As with other objects, you can set figure properties also setp or with the set_something methods.

---

**Subplots**

With subplot you can arrange plots in a regular grid. You need to specify the number of rows and columns and the number of the plot. Note that the gridspec command is a more powerful alternative.
Axes

Axes are very similar to subplots but allow placement of plots at any location in the figure. So if we want to put a smaller plot inside a bigger one we do so with axes.

Ticks

Well formatted ticks are an important part of publishing-ready figures. Matplotlib provides a totally configurable system for ticks. There are tick locators to specify where ticks should appear and tick formatters to give ticks the appearance you want. Major and minor ticks can be
located and formatted independently from each other. Per default minor ticks are not shown, i.e. there is only an empty list for them because it is as NullLocator (see below).

**Tick Locators**

There are several locators for different kind of requirements:

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NullLocator</td>
<td>No ticks.</td>
</tr>
<tr>
<td>IndexLocator</td>
<td>Place a tick on every multiple of some base number of points plotted.</td>
</tr>
<tr>
<td>FixedLocator</td>
<td>Tick locations are fixed.</td>
</tr>
<tr>
<td>LinearLocator</td>
<td>Determine the tick locations.</td>
</tr>
<tr>
<td>MultipleLocator</td>
<td>Set a tick on every integer that is multiple of some base.</td>
</tr>
<tr>
<td>AutoLocator</td>
<td>Select no more than n intervals at nice locations.</td>
</tr>
<tr>
<td>LogLocator</td>
<td>Determine the tick locations for log axes.</td>
</tr>
</tbody>
</table>

All of these locators derive from the base class matplotlib.ticker.Locator. You can make your own locator deriving from it. Handling dates as ticks can be especially tricky. Therefore, matplotlib provides special locators in matplotlib.dates.

**Other Types of Plots**
REGULAR PLOT
PLOT LINES AND/OR MARKERS

SCATTER PLOT
MAKE A SCATTER PLOT OF X VERSUS Y

BAR PLOT
MAKE A BAR PLOT WITH RECTANGLES
CONTOUR PLOT
DRAW CONTOUR LINES AND FILLED CONTOURS

IMSHOW
DISPLAY AN IMAGE TO CURRENT AXES

QUIVER PLOT
PLOT A 2-D FIELD OF ARROWS
PIE CHART
MAKE A PIE CHART OF AN ARRAY

GRID
DRAW TICKS AND GRID

MULTI PLOT
PLOT SEVERAL PLOTS AT ONCE
Polar Axis
Plot anything using Polar Axis

3D Plots
Plot 2D or 3D data

Text
Draw any kind of text

\[ E = mc^2 = \sqrt{m_0^2 c^4 + \left(\frac{p^2}{c^2}\right)^2} = m_0 c^2 \]
**Regular Plots**

**Hints**
You need to use the `fill_between` command.

Starting from the code below, try to reproduce the graphic on the right taking care of filled areas:

```python
from pylab import *

n = 256
X = np.linspace(-n * np.pi, np.pi, n, endpoint=True)
Y = np.sin(2*X)

plot(X, Y+1, color='blue', alpha=1.00)
plot(X, Y-1, color='blue', alpha=1.00)
show()
```

Click on figure for solution.

---

**Scatter Plots**

**Hints**
Color is given by angle of (X,Y).

Starting from the code below, try to reproduce the graphic on the right taking care of marker size, color and transparency.

```python
from pylab import *

n = 1024
X = np.random.randn(n)
Y = np.random.randn(n)

scatter(X, Y)
show()
```

Click on figure for solution.
Bar Plots

Starting from the code below, try to reproduce the graphic on the right by adding labels for red bars.

```python
from pylab import *

n = 12
X = np.arange(n)
Y1 = (1-X/float(n)) * np.random.uniform(0.5,1.0,n)
Y2 = (1-X/float(n)) * np.random.uniform(0.5,1.0,n)

bar(X, +Y1, facecolor='#9999ff', edgecolor='white')
bar(X, -Y2, facecolor='#ff9999', edgecolor='white')

for x,y in zip(X,Y1):
    text(x+0.4, y+0.05, '%.2f' % y, ha='center', va='bottom')

ylim(-1.25,+1.25)
show()
```

Click on figure for solution.

Contour Plots

Starting from the code below, try to reproduce the graphic on the right taking care of the colormap (see Colormaps below).

```python
from pylab import *

def f(x,y): return (1-x/2+x**5+y**3) *np.exp(-x**2-y**2)

n = 256
x = np.linspace(-3,3,n)
y = np.linspace(-3,3,n)
X,Y = np.meshgrid(x,y)

contourf(X, Y, f(X,Y), 8, alpha=.75, cmap='jet')
```
C = contour(X, Y, f(X,Y), 8, colors='black', linewidth=.5) show()

Click on figure for solution.

**Imshow**

**Hints**
You need to take care of the origin of the image in the imshow command and use a colorbar

Starting from the code below, try to reproduce the graphic on the right taking care of colormap, image interpolation and origin.

```python
from pylab import *

def f(x,y): return (1-x/2+x**5+y**3)*np.exp(-x**2-y**2)

n = 10
x = np.linspace(-3,3,4*n)
y = np.linspace(-3,3,3*n)
X,Y = np.meshgrid(x,y)
imshow(f(X,Y)), show()
```

Click on figure for solution.

**Pie Charts**

**Hints**
You need to modify Z.

Starting from the code below, try to reproduce the graphic on the right taking care of colors and slices size.

```python
from pylab import *

n = 20
Z = np.random.uniform(0,1,n)
pie(Z), show()
```

Click on figure for solution.
### Quiver Plots

**Hints**
You need to draw arrows twice.

Starting from the code above, try to reproduce the graphic on the right taking care of colors and orientations.

```python
from pylab import *
n = 8
X,Y = np.mgrid[0:n,0:n]
quiver(X,Y), show()
```

Click on figure for solution.

### Grids

Starting from the code below, try to reproduce the graphic on the right taking care of line styles.

```python
from pylab import *
axes = gca()
axes.set_xlim(0,4)
axes.set_ylim(0,3)
axes.set_xticklabels([])
axes.set_yticklabels([])
show()
```

Click on figure for solution.

### Multi Plots
Hints
You can use several subplots with different partition.

Starting from the code below, try to reproduce the graphic on the right.

```python
from pylab import *

subplot(2,2,1)
subplot(2,2,3)
subplot(2,2,4)
show()
```

Click on figure for solution.

---

### Polar Axis

Hints
You only need to modify the axes line

Starting from the code below, try to reproduce the graphic on the right.

```python
from pylab import *

axes([0,0,1,1])

N = 20
theta = np.arange(0.0, 2*np.pi, 2*np.pi/N)
radii = 10*np.random.rand(N)
width = np.pi/4*np.random.rand(N)
bars = bar(theta, radii, width=width, bottom=0.0)

for r,bar in zip(radii, bars):
    bar.set_facecolor( cm.jet(r/10.))
    bar.set_alpha(0.5)

show()
```

Click on figure for solution.

---

### 3D Plots

Starting from the code below, try to reproduce the graphic on the right.

```python
from pylab import *
```
from mpl_toolkits.mplot3d import Axes3D

fig = figure()
ax = Axes3D(fig)
X = np.arange(-4, 4, 0.25)
Y = np.arange(-4, 4, 0.25)
X, Y = np.meshgrid(X, Y)
R = np.sqrt(X**2 + Y**2)
Z = np.sin(R)

ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap='hot')

show()

Click on figure for solution.

### Text

Hints

You need to use `contourf`.

Have a look at the `matplotlib` logo.

Try to do the same from scratch!

Click on figure for solution.

### Beyond this tutorial

Matplotlib benefits from extensive documentation as well as a large community of users and developers. Here are some links of interest:

### Tutorials

Pyplot tutorial
Introduction
Controlling line properties
Working with multiple figures and axes
Working with text

Image tutorial
Startup commands
Importing image data into Numpy arrays
Plotting numpy arrays as images

Text tutorial
Text introduction
Basic text commands
Text properties and layout
Writing mathematical expressions
Text rendering With LaTeX
Annotating text

Artist tutorial
Introduction
Customizing your objects
Object containers
Figure container
Axes container
Axis containers
Tick containers

Path tutorial
Introduction
Bézier example
Compound paths

Transforms tutorial
Introduction
Data coordinates
Axes coordinates
Blended transformations
Using offset transforms to create a shadow effect
The transformation pipeline

Matplotlib documentation

User guide
FAQ
Installation
Usage
How-To
Troubleshooting
Environment Variables
Code documentation

The code is fairly well documented and you can quickly access a specific command from within a python session:

```python
>>> from pylab import *
>>> help(plot)
Help on function plot in module matplotlib.pyplot:

plot(*args, **kwargs)
   Plot lines and/or markers to the
   :class:`matplotlib.axes.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:

   >>> 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.
   ...
```

Galleries

The matplotlib gallery is also incredibly useful when you search how to render a given graphic. Each example comes with its source.

A smaller gallery is also available [here](#).

Mailing lists

Finally, there is a user mailing list where you can ask for help and a developers mailing list that is more technical.
Quick references

Here is a set of tables that show main properties and styles.

## Line properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Appearance</th>
</tr>
</thead>
<tbody>
<tr>
<td>alpha (or a)</td>
<td>alpha transparency on 0-1 scale</td>
<td><img src="image" alt="alpha" /></td>
</tr>
<tr>
<td>antialiasd</td>
<td>True or False - use antialiased rendering</td>
<td>Aliased</td>
</tr>
<tr>
<td>color (or c)</td>
<td>matplotlib color arg</td>
<td><img src="image" alt="color" /></td>
</tr>
<tr>
<td>linestyle (or ls)</td>
<td>see Line properties</td>
<td><img src="image" alt="linestyle" /></td>
</tr>
<tr>
<td>linewidth (or lw)</td>
<td>float, the line width in points</td>
<td><img src="image" alt="linewidth" /></td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>Cap style for solid lines</td>
<td><img src="image" alt="solid_capstyle" /></td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>Join style for solid lines</td>
<td><img src="image" alt="solid_joinstyle" /></td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>Cap style for dashes</td>
<td><img src="image" alt="dash_capstyle" /></td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>Join style for dashes</td>
<td><img src="image" alt="dash_joinstyle" /></td>
</tr>
<tr>
<td>marker</td>
<td>see Markers</td>
<td><img src="image" alt="marker" /></td>
</tr>
<tr>
<td>markeredgewidth</td>
<td>line width around the marker symbol</td>
<td><img src="image" alt="markeredgewidth" /></td>
</tr>
<tr>
<td>markeredgecolor</td>
<td>edge color if a marker is used</td>
<td><img src="image" alt="markeredgecolor" /></td>
</tr>
<tr>
<td>markerfacecolor</td>
<td>face color if a marker is used</td>
<td><img src="image" alt="markerfacecolor" /></td>
</tr>
<tr>
<td>markersize (ms)</td>
<td>size of the marker in points</td>
<td><img src="image" alt="markersize" /></td>
</tr>
</tbody>
</table>
### Line styles

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Appearance</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>solid line</td>
<td></td>
</tr>
<tr>
<td>--</td>
<td>dashed line</td>
<td>:</td>
</tr>
<tr>
<td>-.</td>
<td>dash-dot line</td>
<td></td>
</tr>
<tr>
<td>:</td>
<td>dotted line</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td>points</td>
<td></td>
</tr>
<tr>
<td>,</td>
<td>pixels</td>
<td></td>
</tr>
<tr>
<td>o</td>
<td>circle</td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>triangle up</td>
<td>△</td>
</tr>
<tr>
<td>v</td>
<td>triangle down</td>
<td>▽</td>
</tr>
<tr>
<td>&lt;</td>
<td>triangle left</td>
<td>▼</td>
</tr>
<tr>
<td>&gt;</td>
<td>triangle right</td>
<td>▲</td>
</tr>
<tr>
<td>s</td>
<td>square</td>
<td>□</td>
</tr>
<tr>
<td>+</td>
<td>plus</td>
<td>+</td>
</tr>
<tr>
<td>x</td>
<td>cross</td>
<td>×</td>
</tr>
<tr>
<td>D</td>
<td>diamond</td>
<td>◆</td>
</tr>
<tr>
<td>d</td>
<td>thin diamond</td>
<td>◆</td>
</tr>
<tr>
<td>1</td>
<td>tripod down</td>
<td>▼</td>
</tr>
<tr>
<td>2</td>
<td>tripod up</td>
<td>▲</td>
</tr>
<tr>
<td>3</td>
<td>tripod left</td>
<td>▼</td>
</tr>
<tr>
<td>4</td>
<td>tripod right</td>
<td>▲</td>
</tr>
<tr>
<td>h</td>
<td>hexagon</td>
<td>□</td>
</tr>
<tr>
<td>H</td>
<td>rotated hexagon</td>
<td>□</td>
</tr>
<tr>
<td>p</td>
<td>pentagon</td>
<td>□</td>
</tr>
<tr>
<td></td>
<td></td>
<td>vertical line</td>
</tr>
<tr>
<td>_</td>
<td>horizontal line</td>
<td>—</td>
</tr>
</tbody>
</table>

### Markers

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Appearance</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>tick left</td>
<td>—</td>
</tr>
<tr>
<td>1</td>
<td>tick right</td>
<td>—</td>
</tr>
<tr>
<td>2</td>
<td>tick up</td>
<td>—</td>
</tr>
<tr>
<td>3</td>
<td>tick down</td>
<td>—</td>
</tr>
<tr>
<td>4</td>
<td>caret left</td>
<td>◄</td>
</tr>
<tr>
<td>5</td>
<td>caret right</td>
<td>►</td>
</tr>
</tbody>
</table>
### Colormaps

All colormaps can be reversed by appending \_r. For instance, gray_r is the reverse of gray.

If you want to know more about colormaps, checks Documenting the matplotlib colormaps.

### Base

<table>
<thead>
<tr>
<th>Name</th>
<th>Appearance</th>
</tr>
</thead>
</table>
### GIST

<table>
<thead>
<tr>
<th>Name</th>
<th>Appearance</th>
</tr>
</thead>
<tbody>
<tr>
<td>gist_earth</td>
<td></td>
</tr>
<tr>
<td>gist_gray</td>
<td></td>
</tr>
<tr>
<td>gist_heat</td>
<td></td>
</tr>
<tr>
<td>gist_ncar</td>
<td></td>
</tr>
<tr>
<td>gist_rainbow</td>
<td></td>
</tr>
<tr>
<td>gist_stern</td>
<td></td>
</tr>
<tr>
<td>gist_yarg</td>
<td></td>
</tr>
</tbody>
</table>

### Sequential

<table>
<thead>
<tr>
<th>Name</th>
<th>Appearance</th>
</tr>
</thead>
<tbody>
<tr>
<td>BrBG</td>
<td></td>
</tr>
<tr>
<td>PiYG</td>
<td></td>
</tr>
<tr>
<td>PRGn</td>
<td></td>
</tr>
<tr>
<td>PuOr</td>
<td></td>
</tr>
<tr>
<td>RdBu</td>
<td></td>
</tr>
<tr>
<td>RdGy</td>
<td></td>
</tr>
<tr>
<td>RdYlBu</td>
<td></td>
</tr>
<tr>
<td>RdYlGn</td>
<td></td>
</tr>
<tr>
<td>Spectral</td>
<td></td>
</tr>
</tbody>
</table>

### Diverging

<table>
<thead>
<tr>
<th>Name</th>
<th>Appearance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blues</td>
<td></td>
</tr>
</tbody>
</table>
### Qualitative

<table>
<thead>
<tr>
<th>Name</th>
<th>Appearance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accent</td>
<td></td>
</tr>
<tr>
<td>Dark2</td>
<td></td>
</tr>
<tr>
<td>Paired</td>
<td></td>
</tr>
<tr>
<td>Pastel1</td>
<td></td>
</tr>
<tr>
<td>Pastel2</td>
<td></td>
</tr>
<tr>
<td>Set1</td>
<td></td>
</tr>
<tr>
<td>Set2</td>
<td></td>
</tr>
<tr>
<td>Set3</td>
<td></td>
</tr>
</tbody>
</table>

### Miscellaneous

<table>
<thead>
<tr>
<th>Name</th>
<th>Appearance</th>
</tr>
</thead>
<tbody>
<tr>
<td>afmhot</td>
<td></td>
</tr>
<tr>
<td>binary</td>
<td></td>
</tr>
<tr>
<td>brg</td>
<td></td>
</tr>
<tr>
<td>bwr</td>
<td></td>
</tr>
<tr>
<td>coolwarm</td>
<td></td>
</tr>
<tr>
<td>CMRmap</td>
<td></td>
</tr>
<tr>
<td>cubehelix</td>
<td></td>
</tr>
<tr>
<td>gnuplot</td>
<td></td>
</tr>
<tr>
<td>gnuplot2</td>
<td></td>
</tr>
<tr>
<td>Term</td>
<td>Color</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
</tr>
<tr>
<td>ocean</td>
<td></td>
</tr>
<tr>
<td>rainbow</td>
<td></td>
</tr>
<tr>
<td>seismic</td>
<td></td>
</tr>
<tr>
<td>terrain</td>
<td></td>
</tr>
</tbody>
</table>