MATPLOTLIB 基础

金　林
中南财经政法大学统计系
jinlin82@qq.com

2020 年 2 月 15 日

Facts

1. Initial release:2003; 13 years ago
2. Stable release: Stable release: 0.18.1 / 22 September 2016
3. Website: `http://matplotlib.org`

What is matplotlib?

1. matplotlib is a library for making 2D plots of arrays in Python.
2. Although it has its origins in emulating the MATLAB graphics commands, it is independent of MATLAB, and can be used in a Pythonic, object oriented way.
3. 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.

three parts

1. The matplotlib code is conceptually divided into three parts:
   1. the pylab interface is the set of functions provided by matplotlib.
      1. pylab which allow the user to create plots with code quite similar to MATLAB figure generating code.
      2. Typically pylab is imported to bring NumPy and matplotlib into a single global namespace for the most MATLAB like syntax, however a more explicit import style, which names both matplotlib and NumPy, is the preferred coding style.
   2. 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.
   3. The backends are device-dependent drawing devices, aka renderers, that transform the frontend representation to hardcopy or a display device.

Two interfaces

1. matplotlib has two interfaces.
2. The first is based on MATLAB and uses a state-based interface.
3. The second option is an an object-oriented(OO) interface.
4. knowing that there are two approaches is vitally important when plotting with matplotlib.

- Parts of a Figure
- Backends

Figure

1. The whole figure. The figure keeps track of all the child Axes, a smattering of 'special' artists (titles, figure legends, etc), and the canvas.
2. A figure can have any number of Axes, but to be useful should have at least one.
3. The easiest way to create a new figure is with pyplot:

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 fig = plt.figure()  # an empty figure with no axes
5 fig.suptitle('No axes on this figure')  # Add a title so we know
      which it is
6
7 fig, ax_lst = plt.subplots(2, 2)  # a figure with a 2x2 grid of
      Axes
```

Axes

1. This is what you think of as 'a plot', it is the region of the image with the data space.
2. A given figure can contain many Axes, but a given Axes object can only be in one Figure.
3. The Axes contains two (or three in the case of 3D) Axis objects (be aware of the difference between Axes and Axis) which take care of the data limits (the data limits can also be controlled via set via the set_xlim() and set_ylim() Axes methods).
4. Each Axes has a title (set via set_title()), an x-label (set via set_xlabel()), and a y-label set via set_ylabel()).
5. The Axes class and its member functions are the primary entry point to working with the OO interface.

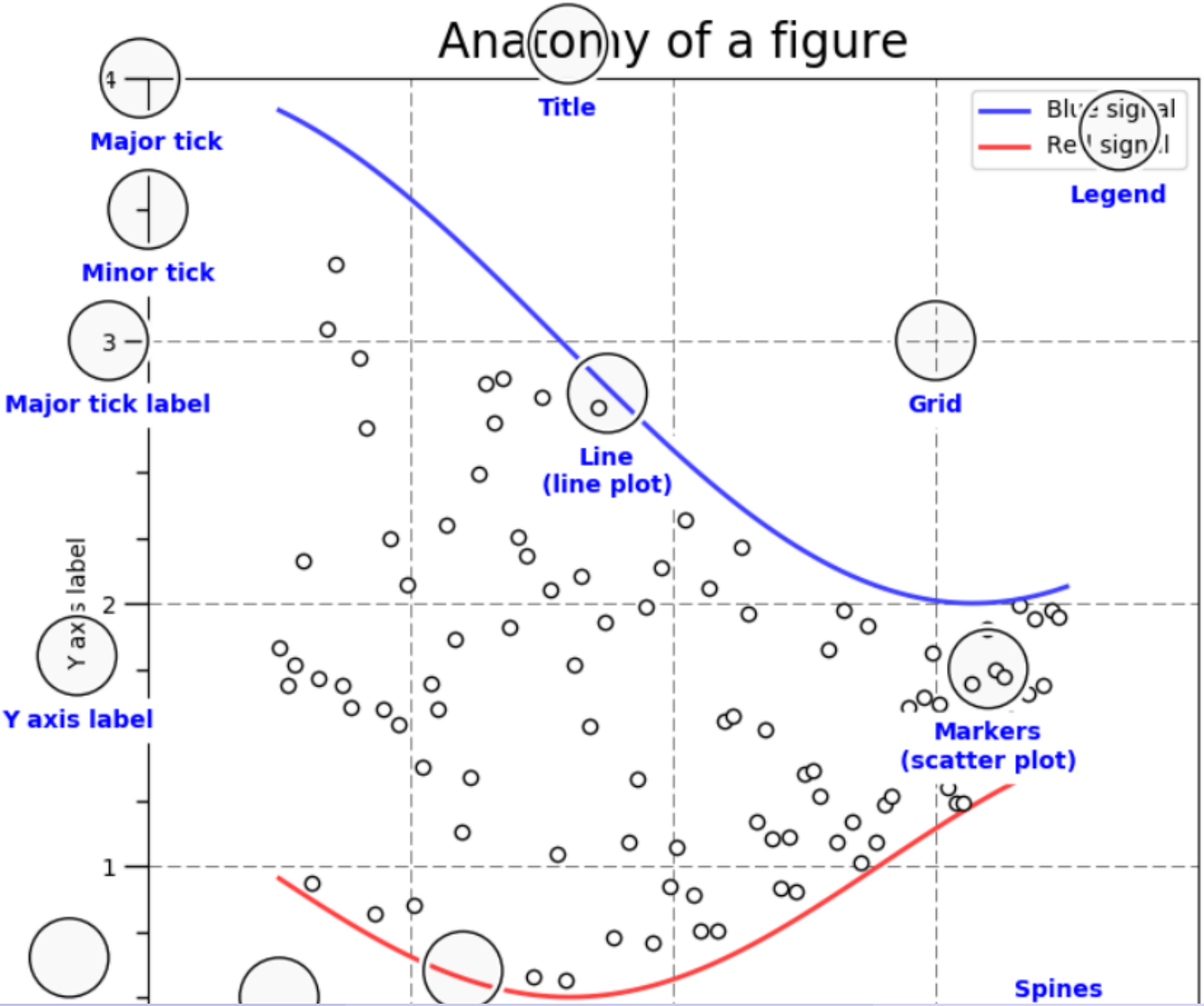
Axis

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

Artist

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

其他组成部分


Anatomy of a figure
Title
Major tick
Minor tick
Major tick label
Legend
Grid
Line
(line plot)
Y axis label
Markers
(scatter plot)
Spines

Types of inputs to plotting functions

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

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3 import pandas as pd
4
5 a = pd.DataFrame(np.random.rand(4,5), columns = list('abcde'))
6 a_asarray = a.values
7
8 b = np.matrix([[1,2],[3,4]])
9 b_asarray = np.asarray(b)
```

Matplotlib, pyplot and pylab: how are they related?

1. Matplotlib is the whole package
2. and matplotlib.pyplot is a module in Matplotlib.
3. pylab is a convenience module that bulk imports matplotlib.pyplot (for plotting) and numpy (for mathematics and working with arrays) in a single namespace. pylab is deprecated and its use is strongly discouraged because of namespace pollution. Use pyplot instead.

pyplot

1. For functions in the pyplot module, there is always a "current" figure and axes (which is created automatically on request).
2. For example, in the following example, the first call to plt.plot creates the axes,
3. then subsequent calls to plt.plot add additional lines on the same axes,
4. and plt.xlabel, plt.ylabel, plt.title and plt.legend set the axes labels and title and add a legend.

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

plt.plot(x, x, label='linear')
plt.plot(x, x**2, label='quadratic')
plt.plot(x, x**3, label='cubic')

plt.xlabel('x label')
plt.ylabel('y label')
plt.title("Simple Plot")
plt.legend()

plt.show()
```

- Parts of a Figure
- Backends

What is a backend?

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

Configure your backend

1. The backend parameter in your matplotlibrc file
2. If your script depends on a specific backend you can use the use() function.
3. If you use the use() function, this must be done before importing matplotlib.pyplot. Calling use() after pyplot has been imported will have no effect.
4. Using use() will require changes in your code if users want to use a different backend.

```
1 import matplotlib
2 matplotlib.use('pdf')   ### generate postscript output by default
```

What is interactive mode?

1. Use of an interactive backend permits plotting to the screen.
2. Interactive mode may also be turned on via `matplotlib.pyplot.ion()`,
3. and turned off via `matplotlib.pyplot.ioff()`.
4. Non-interactive example:

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

1. Nothing happened–or at least nothing has shown up on the screen, To make the plot appear, you need to do this:

```
1 plt.show()
```

1. 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.

Intro to pyplot

1. `matplotlib.pyplot` is a collection of command style functions that make matplotlib work like MATLAB.
2. Each pyplot function makes some change to a figure: e.g., creates a figure, creates a plotting area in a figure, plots some lines in a plotting area, decorates the plot with labels, etc.
3. In `matplotlib.pyplot` various states are preserved across function calls, so that it keeps track of things like the current figure and plotting area, and the plotting functions are directed to the current axes.
4. "axes" here and in most places in the documentation refers to the axes part of a figure and not the strict mathematical term for more than one axis.

Formatting the style of your plot

1. 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.
2. The letters and symbols of the format string are from MATLAB, and you concatenate a color string with a line style string.
3. The default format string is `'b-'`, which is a solid blue line. For example, to plot the above with red circles, use `'ro'`.
4. The `axis()` command in the example above takes a list of `[xmin, xmax, ymin, ymax]` and specifies the viewport of the axes.

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

use numpy arrays

1. matplotlib uses numpy arrays.
2. In fact, all sequences are converted to numpy arrays internally.
3. The example below illustrates a plotting several lines with different format styles in one command using arrays.

```
import numpy as np

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

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

Plotting with keyword strings

1. There are some instances where you have data in a format that lets you access particular variables with strings.
2. For example, `pandas.DataFrame`.
3. Matplotlib allows you provide such an object with the `data` keyword argument.
4. If provided, then you may generate plots with the strings corresponding to these variables.

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

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

Plotting with categorical variables

1. Matplotlib allows you to pass categorical variables directly to many plotting functions.

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

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

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

- line properties
- multiple figures and axes
- Working with text

Controlling line properties

1. Lines have many attributes that you can set: linewidth, dash style, antialiased, etc; see matplotlib.lines.Line2D.
2. There are several ways to set line properties:
   1. Use keyword args: `plt.plot(x, y, linewidth=2.0)`
   2. Use the setter methods of a Line2D instance. plot returns a list of Line2D objects; e.g., line1, line2 = plot(x1, y1, x2, y2). In the code below we will suppose that we have only one line so that the list returned is of length 1. We use tuple unpacking with line, to get the first element of that list:

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

Controlling line properties

3. 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.

```
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)
```

4. To get a list of settable line properties, call the `setp()` function with a line or lines as argument.

```
lines = plt.plot([1, 2, 3])
plt.setp(lines)
```

- line properties
- multiple figures and axes
- Working with text

Working with multiple figures and axes

1. pyplot has the concept of the current figure and the current axes.
2. All plotting commands apply to the current axes.
3. The function `gca()` returns the current axes (a matplotlib.axes.Axes instance),
4. and `gcf()` returns the current figure (matplotlib.figure.Figure instance).
5. Normally, you don't have to worry about this, because it is all taken care of behind the scenes.

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

t1 = np.arange(0.0, 5.0, 0.1)
t2 = np.arange(0.0, 5.0, 0.02)
plt.figure()
plt.subplot(211)
plt.plot(t1, f(t1), 'bo', t2, f(t2), 'k')
plt.subplot(212)
plt.plot(t2, np.cos(2*np.pi*t2), 'r--')
plt.show()
```

Working with multiple figures and axes

1. 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 any axes.
2. The subplot() command specifies numrows, numcols, plot_number where plot_number ranges from 1 to numrows*numcols.
3. The commas in the subplot command are optional if numrows*numcols<10. So subplot(211) is identical to subplot(2, 1, 1).
4. create multiple figures by using multiple figure() calls with an increasing figure number.
5. clear the current figure with clf() and the current axes with cla().
6. the memory required for a figure is not completely released until the figure is explicitly closed with close().

例子

```
1  import matplotlib.pyplot as plt
2  plt.figure(1)                # the first figure
3  plt.subplot(211)             # the first subplot in the first
       figure
4  plt.plot([1, 2, 3])
5  plt.subplot(212)             # the second subplot in the first
       figure
6  plt.plot([4, 5, 6])
7
8
9  plt.figure(2)                # a second figure
10 plt.plot([4, 5, 6])          # creates a subplot(111) by default
11
12 plt.figure(1)                # figure 1 current; subplot(212) still
       current
13 plt.subplot(211)             # make subplot(211) in figure1 current
14 plt.title('Easy as 1, 2, 3') # subplot 211 title
```

- line properties
- multiple figures and axes
- Working with text

Working with text

1. The text() command can be used to add text in an arbitrary location,
2. and the xlabel(), ylabel() and title() are used to add text in the indicated locations.
3. All of the text() commands return an matplotlib.text.Text instance.
4. Just as with with lines above, you can customize the properties by passing keyword arguments into the text functions or using setp():

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

例子

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

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


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

Using mathematical expressions in text

1. matplotlib accepts TEX equation expressions in any text expression.

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

2. The r preceding the title string is important – it signifies that the string is a raw string and not to treat backslashes as python escapes.
3. matplotlib has a built-in TEX expression parser and layout engine, and ships its own math fonts. Thus you can use mathematical text across platforms without requiring a TEX installation.
4. 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

Annotating text

1. A common use for text is to annotate some feature of the plot, and the annotate() method provides helper functionality to make annotations easy.
2. In an annotation, there are two points to consider:
   1. the location being annotated represented by the argument xy
   2. and the location of the text xytext.
   3. Both of these arguments are (x,y) tuples.

例子

```
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),
^^I     arrowprops=dict(facecolor='black', shrink=0.05),
^^I     )

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

the Object-Oriented API vs Pyplot

1. Matplotlib has two interfaces. The first is an object-oriented (OO) interface. In this case, we utilize an instance of axes.Axes in order to render visualizations on an instance of figure.Figure.
2. The second is based on MATLAB and uses a state-based interface. This is encapsulated in the pyplot module.
3. Most of the terms are straightforward but the main thing to remember is that:
   1. The Figure is the final image that may contain 1 or more Axes.
   2. The Axes represent an individual plot (don't confuse this with the word "axis", which refers to the x/y axis of a plot).
   3. We call methods that do the plotting directly from the Axes, which gives us much more flexibility and power in customizing our plot.
4. In general, try to use the object-oriented interface over the pyplot interface.

data

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.ticker import FuncFormatter

data = {'Barton LLC': 109438.50,
^^I'Frami, Hills and Schmidt': 103569.59,
^^I'Fritsch, Russel and Anderson': 112214.71,
^^I'Jerde-Hilpert': 112591.43,
^^I'Keeling LLC': 100934.30,
^^I'Koepp Ltd': 103660.54,
^^I'Kulas Inc': 137351.96,
^^I'Trantow-Barrows': 123381.38,
^^I'White-Trantow': 135841.99,
^^I'Will LLC': 104437.60}

group_data = list(data.values())
group_names = list(data.keys())
group_mean = np.mean(group_data)
```

Figure and axes

1. To do this with the object-oriented approach, we'll first generate an instance of figure.Figure and axes.Axes.
2. The Figure is like a canvas, and the Axes is a part of that canvas on which we will make a particular visualization.
3. Figures can have multiple axes on them.

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

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

```
1 ax.barh(group_names, group_data)
```

Controlling the style

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

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

1. You can activate a style with the following:

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

Customizing the plot

1. rotate the labels on the x-axis so that they show up more clearly.
2. We can gain access to these labels with the axes.Axes.get_xticklabels() method:

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

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

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

Customizing the plot

1. tell Matplotlib to automatically make room for elements in the figures that we create.
2. To do this we'll set the autolayout value of our rcParams.

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

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

1. add labels to the plot. To do this with the OO interface, we can use the axes.Axes.set() method to set properties of this Axes object.
2. adjust the size of this plot using the pyplot.subplots() function. We can do this with the figsize kwarg.
3. For labels, we can specify custom formatting guidelines in the form of functions by using the ticker.FuncFormatter class.

Customizing the plot

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

formatter = FuncFormatter(currency)

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

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

Combining multiple visualizations

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

Combining multiple visualizations

```
fig, ax = plt.subplots(figsize=(8, 8))
ax.barh(group_names, group_data)
labels = ax.get_xticklabels()
plt.setp(labels, rotation=45, horizontalalignment='right')
# Add a vertical line, here we set the style in the function call
ax.axvline(group_mean, ls='--', color='r')
# Annotate new companies
for group in [3, 5, 8]:
    ax.text(145000, group, "New Company", fontsize=10,
^^I    verticalalignment="center")
# Now we'll move our title up since it's getting a little cramped
ax.title.set(y=1.05)
ax.set(xlim=[-10000, 140000], xlabel='Total Revenue', ylabel='
    Company',
       title='Company Revenue')
ax.xaxis.set_major_formatter(formatter)
ax.set_xticks([0, 25e3, 50e3, 75e3, 100e3, 125e3])
fig.subplots_adjust(right=.1)
plt.show()
```

Saving plots

1. There are many file formats we can save to in Matplotlib. To see a list of available options, use:

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

1. We can then use the figure.Figure.savefig() in order to save the figure to disk.
2. Note that there are several useful flags:
   1. transparent=True makes the background of the saved figure transparent if the format supports it.
   2. dpi=80 controls the resolution (dots per square inch) of the output.
   3. bbox_inches="tight" fits the bounds of the figure to our plot.

```
fig.savefig('sales.png', transparent=False, dpi=80, bbox_inches
    ="tight")
```

- style sheets
- matplotlib rcParams

Using style sheets

1. The style package adds support for easy-to-switch plotting "styles" with the same parameters as a `matplotlib rc` file (which is read at startup to configure matplotlib).
2. There are a number of pre-defined styles provided by Matplotlib. For example, there's a pre-defined style called "ggplot", which emulates the aesthetics of ggplot (a popular plotting package for R).

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

1. To list all available styles, use: `print(plt.style.available)`

Defining your own style

1. You can create custom styles and use them by calling style.use with the path or URL to the style sheet.
2. Additionally, if you add your <style-name>.mplstyle file to mpl_configdir/stylelib, you can reuse your custom style sheet with a call to style.use(<style-name>).

Composing styles

1. Style sheets are designed to be composed together.
2. So you can have a style sheet that customizes colors and a separate style sheet that alters element sizes for presentations.
3. These styles can easily be combined by passing a list of styles:
4. Note that styles further to the right will overwrite values that are already defined by styles on the left.

```
1 # plt.style.use(['dark_background', 'presentation'])
```

Temporary styling

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

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

plt.show()
```

- style sheets
- matplotlib rcParams

Dynamic rc settings

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

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

Dynamic rc settings

1. 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.
2. The matplotlib.rcdefaults() command will restore the standard matplotlib default settings.
3. There is some degree of validation when setting the values of rcParams, see matplotlib.rcsetup for details.

```
mpl.rc('lines', linewidth=4, color='g')
plt.plot(data)
```

The matplotlibrc file

1. matplotlib uses matplotlibrc configuration files to customize all kinds of properties, which we call rc settings or rc parameters.
2. 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.
3. Once a matplotlibrc file has been found, it will not search any of the other paths.
4. To display where the currently active matplotlibrc file was loaded from, one can do the following:

```
1 import matplotlib
2 matplotlib.matplotlib_fname()
```

- 基本概念
- Customizing your objects
- Object containers

three layers to the matplotlib API

1. the matplotlib.backend_bases.FigureCanvas is the area onto which the figure is drawn
2. the matplotlib.backend_bases.Renderer is the object which knows how to draw on the FigureCanvas
3. and the matplotlib.artist.Artist is the object that knows how to use a renderer to paint onto the canvas.
4. Typically, all visible elements in a figure are subclasses of Artist.

three layers to the matplotlib API

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

two types of Artists: primitives and containers

1. The primitives represent the standard graphical objects we want to paint onto our canvas: `Line2D, Rectangle, Text, AxesImage`, etc.,
2. and the containers are places to put them (`Axis, Axes and Figure`).
3. The standard use is to create a Figure instance, use the Figure to create one or more Axes or Subplot instances,
4. and use the Axes instance helper methods to create the primitives.

how to create Figure instance

1. we can 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. However, this is not necessary.
2. you can work directly with PostScript, PDF Gtk+, or wxPython FigureCanvas instances, instantiate your Figures directly and connect them yourselves.

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

Axes

1. The Axes is probably the most important class in the matplotlib API, and the one you will be working with most of the time.
2. This is because the Axes is the plotting area into which most of the objects go,
3. and the Axes has many special helper methods (plot(), text(), hist(), imshow()) to create the most common graphics primitives (Line2D, Text, Rectangle, Image, respectively).
4. 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.
5. 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.
6. 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:

例子

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

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)

type(ax.lines)
len(ax.lines)
type(line)
```

1. ax is the Axes instance created by the fig.add_subplot call above (remember Subplot is just a subclass of Axes)
2. and when you call ax.plot, it creates a Line2D instance and adds it to the Axes.lines list.
3. remove lines by calling the list methods

- 基本概念
- Customizing your objects
- Object containers

简介

1. Every element in the figure is represented by a matplotlib Artist,
2. and each has an extensive list of properties to configure its appearance.
3. 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.
4. 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.
5. These instances are stored as member variables `Figure.patch` and `Axes.patch`.

get Properties list

1. Every matplotlib Artist has many properties.
2. inspect the Artist properties is to use the `matplotlib.artist.getp()` Function (simply `getp()` in pyplot), which lists the properties and their values. This works for classes derived from Artist as well, e.g., Figure and Rectangle.

```
1 plt.getp(fig)
2 plt.getp(ax)
```

get and set properties

1. Each of the properties is accessed with an old-fashioned setter or getter.
2. If you want to set a number of properties at once, you can also use the set method with keyword arguments.

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

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

- 基本概念
- Customizing your objects
- Object containers

Figure container

1. The top level container Artist is the matplotlib.figure.Figure, and it contains everything in the figure.
2. The background of the figure is a Rectangle which is stored in Figure.patch.
3. As you add subplots (add_subplot()) and axes (add_axes()) to the figure these will be appended to the Figure.axes.
4. Because the figure maintains the concept of the "current axes" (see Figure.gca and Figure.sca) to support the 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.
5. iterate over the list of axes or index into it to get access to Axes instances you want to customize.

例子

```
fig = plt.figure()

ax1 = fig.add_subplot(211)
ax2 = fig.add_axes([0.1, 0.1, 0.7, 0.3])
print(fig.axes)

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

Axes container

1. The matplotlib.axes.Axes is the center of the matplotlib universe.
2. 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.
3. Like the Figure, it contains a Patch patch which is a Rectangle for Cartesian coordinates and a Circle for polar coordinates;
4. this patch determines the shape, background and border of the plotting region.

Axis containers

1. The matplotlib.axis.Axis instances handle the drawing of the tick lines, the grid lines, the tick labels and the axis label.
2. You can configure the left and right ticks separately for the y-axis, and the upper and lower ticks separately for the x-axis.
3. 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.
4. Each Axis object contains a label attribute (this is what pyplot modifies in calls to xlabel() and ylabel()) as well as a list of major and minor ticks.
5. The ticks are XTick and YTick instances, which contain the actual line and text primitives that render the ticks and ticklabels.
6. access the lists of major and minor ticks through their accessor methods get_major_ticks() and get_minor_ticks().

例子

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

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

1. 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.

Tick containers

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

例子

```
np.random.seed(19680801)

fig, ax = plt.subplots()
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.label1.set_visible(False)
    tick.label2.set_visible(True)
    tick.label2.set_color('green')

plt.show()
```

三种用法

1. `legend()`
2. `legend(labels)`
3. `legend(handles, labels)`

Automatic detection of elements to be shown in the legend

1. The elements to be added to the legend are automatically determined, when you do not pass in any extra arguments.
2. In this case, the labels are taken from the artist.
3. You can specify them either at artist creation or by calling the set_label() method on the artist.
4. Specific lines can be excluded from the automatic legend element selection by defining a label starting with an underscore.
5. This is default for all artists, so calling Axes.legend without any arguments and without setting the labels manually will result in no legend being drawn.

例子

```
line, = ax.plot([1, 2, 3], label='Inline label')
ax.legend()

line, = ax.plot([1, 2, 3])
line.set_label('Label via method')
ax.legend()
```

Labeling existing plot elements

1. To make a legend for lines which already exist on the axes (via plot for instance), simply call this function with an iterable of strings, one for each legend item.
2. Note: This way of using is discouraged, because the relation between plot elements and labels is only implicit by their order and can easily be mixed up.

```
1 ax.plot([1, 2, 3])
2 ax.legend(['A simple line'])
```

Explicitly defining the elements in the legend

1. For full control of which artists have a legend entry, it is possible to pass an iterable of legend artists followed by an iterable of legend labels respectively.

```
legend((line1, line2, line3), ('label1', 'label2', 'lab
```

1. Parameters:
   1. loc : The location of the legend. 'best'
   2. bbox_to_anchor: Box that is used to position the legend in conjunction with loc.
   3. fontsize

例子

1. Examples using matplotlib.pyplot.legend，更多例子见：https://matplotlib.org/api/_as_gen/matplotlib.pyplot.legend.html#matplotlib.pyplot.legend

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  # Make some fake data.
4  a = b = np.arange(0, 3, .02)
5  c = np.exp(a)
6  d = c[::-1]
7  # Create plots with pre-defined labels.
8  fig, ax = plt.subplots()
9  ax.plot(a, c, 'k--', label='Model length')
10 ax.plot(a, d, 'k:', label='Data length')
11 ax.plot(a, c + d, 'k', label='Total message length')
12 legend = ax.legend(loc='upper center', shadow=True, fontsize='x
       -large')
13 # Put a nicer background color on the legend.
14 legend.get_frame().set_facecolor('C0')
15
16 plt.show()
```

Tight Layout

1. tight_layout automatically adjusts subplot params so that the subplot(s) fits in to the figure area.
2. tight_layout() only considers ticklabels, axis labels, and titles. Thus, other artists may be clipped and also may overlap.
3. An alternative to tight_layout is constrained_layout.
4. 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).
5. `tight_layout()` that does this automatically for you.
6. Note that `matplotlib.pyplot.tight_layout()` will only adjust the subplot params when it is called.
7. In order to perform this adjustment each time the figure is redrawn, you can call `fig.set_tight_layout(True)`, or, equivalently, set the figure.autolayout rcParam to True.

例子

```
1  import matplotlib.pyplot as plt
2  import numpy as np
3  plt.rcParams['savefig.facecolor'] = "0.8"
4
5  def example_plot(ax, fontsize=12):
6      ax.plot([1, 2])
7
8      ax.locator_params(nbins=3)
9      ax.set_xlabel('x-label', fontsize=fontsize)
10     ax.set_ylabel('y-label', fontsize=fontsize)
11     ax.set_title('Title', fontsize=fontsize)
12
13 plt.close('all')
14 fig, ax = plt.subplots()
15 example_plot(ax, fontsize=24)
16
17 fig, ax = plt.subplots()
18 example_plot(ax, fontsize=24)
19 plt.tight_layout()
```

multiple subplots

1. When you have multiple subplots, often you see labels of different axes overlapping each other.
2. tight_layout() will also adjust spacing between subplots to minimize the overlaps.
3. tight_layout() will work even if the sizes of subplots are different as far as their grid specification is compatible.

例子

```
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()
```

Constrained Layout

1. constrained_layout automatically adjusts subplots and decorations like legends and colorbars so that they fit in the figure window while still preserving, as best they can, the logical layout requested by the user.
2. constrained_layout is similar to tight_layout, but uses a constraint solver to determine the size of axes that allows them to fit.
3. constrained_layout needs to be activated before any axes are added to a figure. Two ways of doing so are
   1. using the respective argument to subplots() or figure(), e.g.:
      `plt.subplots(constrained_layout=True)`
   2. activate it via rcParams, like:
      `plt.rcParams['figure.constrained_layout.use'] = True`

例子

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

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

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

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

简介

1. Matplotlib has extensive text support, including support for mathematical expressions, truetype support for raster and vector outputs, newline separated text with arbitrary rotations, and unicode support.
2. Matplotlib includes its own matplotlib.font_manager, which implements a cross platform, W3C compliant font finding algorithm.
3. The user has a great deal of control over text properties (font size, font weight, text location and color, etc.) with sensible defaults set in the rc file. And significantly, for those interested in mathematical or scientific figures, Matplotlib implements a large number of TEX math symbols and commands, supporting mathematical expressions anywhere in your figure.

Basic text commands

pyplot API	OO API	description
text	text	Add text at an arbitrary location of the Axes.
annotate	annotate	Add an annotation, with an optional arrow, at an
xlabel	set_xlabel	Add a label to the Axes's x-axis.
ylabel	set_ylabel	Add a label to the Axes's y-axis.
title	set_title	Add a title to the Axes.
figtext	text	Add text at an arbitrary location of the Figure.
suptitle	suptitle	Add a title to the Figure.

- All of these functions create and return a Text instance, which can be configured with a variety of font and other properties.

例子

```
1  import matplotlib
2  import matplotlib.pyplot as plt
3
4  fig = plt.figure()
5  ax = fig.add_subplot(111)
6  fig.subplots_adjust(top=0.85)
7
8  # Set titles for the figure and the subplot respectively
9  fig.suptitle('bold figure suptitle', fontsize=14, fontweight='
       bold')
10 ax.set_title('axes title')
11 ax.set_xlabel('xlabel')
12 ax.set_ylabel('ylabel')
13
14 # Set both x- and y-axis limits to [0, 10] instead of default [0, 1]
15 ax.axis([0, 10, 0, 10])
16
17 ax.text(3, 8, 'boxed italics text in data coords', style='
       italic',
18 ^^Ibbox={'facecolor': 'red', 'alpha': 0.5, 'pad': 10})
19 ax.text(2, 6, r'an equation: $E=mc^2$', fontsize=15)
```

Text properties and layout

1. The matplotlib.text.Text instances have a variety of properties
2. which can be configured via keyword arguments to the text commands (e.g., title(), xlabel() and text()).
3. get properties list via `plt.getp(ax.texts)`

Default Font

1. The base default font is controlled by a set of rcParams.

rcParam	usage
'font.family'	List of either names of font or {'cursive', 'fantasy', 'monospa
'font.style'	The default style, ex 'normal', 'italic'.
'font.variant'	Default variant, ex 'normal', 'small-caps' (untested)
'font.stretch'	Default stretch, ex 'normal', 'condensed' (incomplete)
'font.weight'	Default weight. Either string or integer
'font.size'	Default font size in points. Relative font sizes ('large', 'x-sm

Text with non-latin glyphs

1. Matplotlib still does not cover all of the glyphs that may be required by mpl users.
2. For example, DejaVu has no coverage of Chinese, Korean, or Japanese.
3. set the default font to be one that supports the code points you need, prepend the font name to 'font.family' or the desired alias lists
   1. ```
      matplotlib.rcParams['font.sans-serif'] = ['SimHei', 'sa
      ```
   2. or set it in your .matplotlibrc file:
      ```
      font.sans-serif: SimHei, Arial, sans-serif
      ```

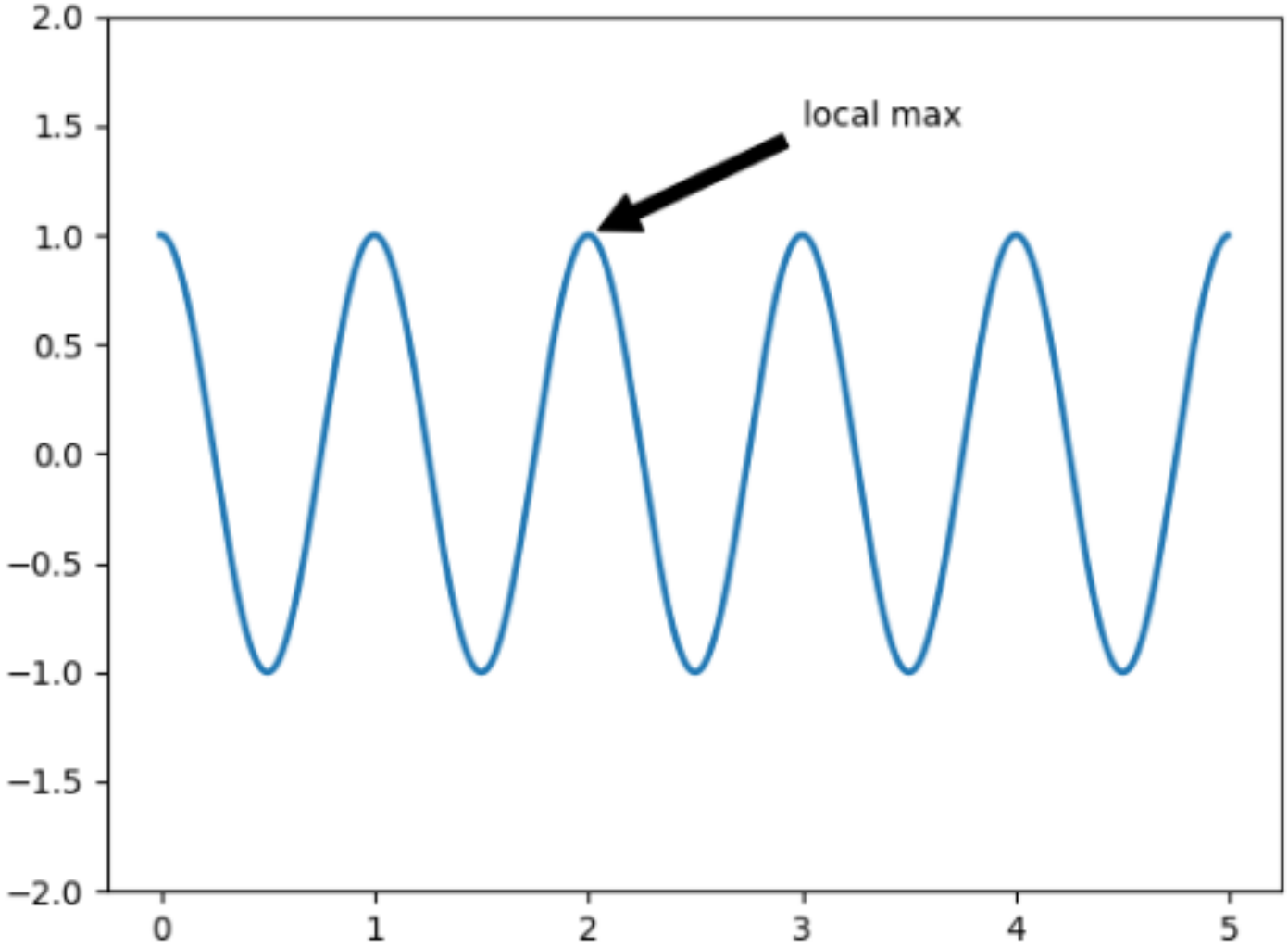

      ```

- Basic annotation
- Advanced Annotation

Basic annotation

1. The uses of the basic text() will place text at an arbitrary position on the Axes.
2. 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.
3. 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.
4. Both of these arguments are (x,y) tuples.

Basic annotation



coordinate systems

1. There are a variety of coordinate systems one can choose.
2. you can specify the coordinate system of xy and xytext with one of the following strings for xycoords and textcoords
3. (default is 'data')

argument	coordinate system
'figure points'	points from the lower left corner of the figure
'figure pixels'	pixels from the lower left corner of the figure
'figure fraction'	0,0 is lower left of figure and 1,1 is upper right
'axes points'	points from lower left corner of axes
'axes pixels'	pixels from lower left corner of axes
'axes fraction'	0,0 is lower left of axes and 1,1 is upper right
'data'	use the axes data coordinate system

例子

```
1  import matplotlib.pyplot as plt
2  ax = plt.subplot(111)
3
4  t = np.arange(0.0, 5.0, 0.01)
5  s = np.cos(2*np.pi*t)
6  line, = plt.plot(t, s, lw=2)
7
8  ax.annotate('local max', xy=(3, 1),  xycoords='data',
9  ^^I    xytext=(0.8, 0.95), textcoords='axes fraction',
10 ^^I    arrowprops=dict(facecolor='black', shrink=0.05),
11 ^^I    horizontalalignment='right', verticalalignment='top',
12 ^^I    )
13
14 plt.ylim(-2, 2)
15 plt.show()
```

argument `arrowprops`

1. you can enable drawing of an arrow from the text to the annotated point by giving a dictionary of arrow properties in the optional keyword argument `arrowprops`.

arrowprops key	description
width	the width of the arrow in points
frac	the fraction of the arrow length occupied by the head
headwidth	the width of the base of the arrow head in points
shrink	move the tip and base some percent away from the annota
**kwargs	any key for matplotlib.patches.Polygon, e.g., facecolor

- Basic annotation
- Advanced Annotation

Annotating with Text with Box

1. 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.
2. The patch object associated with the text can be accessed by:
   `bb = t.get_bbox_patch()`
3. The return value is an instance of FancyBboxPatch and the patch properties like facecolor, edgewidth, etc. can be accessed and modified as usual.
4. To change the shape of the box, use the set_boxstyle method.
5. `pad` ：内边距

```
1 bbox_props = dict(boxstyle="rarrow,pad=0.3", fc="cyan", ec="b",
      lw=2)
2 t = ax.text(0.5, 0.5, "Direction", ha="center", va="center",
    rotation=45,
3 ^^I    size=15,
4 ^^I    bbox=bbox_props)
5 bb = t.get_bbox_patch()
6 bb.set_boxstyle("rarrow", pad=0.6)
```

box styles

Class	Name	Attrs
Circle	circle	pad=0.3
DArrow	darrow	pad=0.3
LArrow	larrow	pad=0.3
RArrow	rarrow	pad=0.3
Round	round	pad=0.3,rounding_size=None
Round4	round4	pad=0.3,rounding_size=None
Roundtooth	roundtooth	pad=0.3,tooth_size=None
Sawtooth	sawtooth	pad=0.3,tooth_size=None
Square	square	pad=0.3

Fancybox list

```
import matplotlib.pyplot as plt
import matplotlib.transforms as mtransforms
import matplotlib.patches as mpatch
from matplotlib.patches import FancyBboxPatch

styles = mpatch.BoxStyle.get_styles()
spacing = 1.2
figheight = (spacing * len(styles) + .5)
fig = plt.figure(figsize=(4 / 1.5, figheight / 1.5))
fontsize = 0.3 * 72

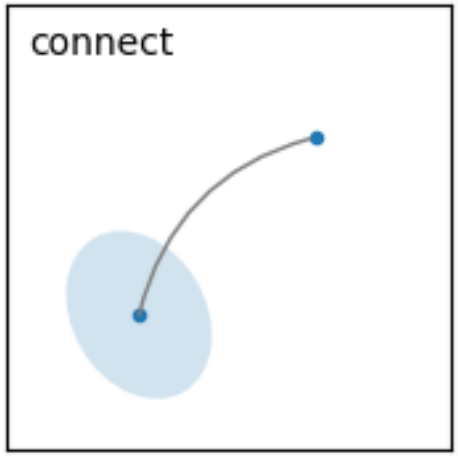
for i, stylename in enumerate(sorted(styles)):
    fig.text(0.5, (spacing * (len(styles) - i) - 0.5) /
    figheight, stylename,
^^I      ha="center",
^^I      size=fontsize,
^^I      transform=fig.transFigure,
^^I      bbox=dict(boxstyle=stylename, fc="w", ec="k"))

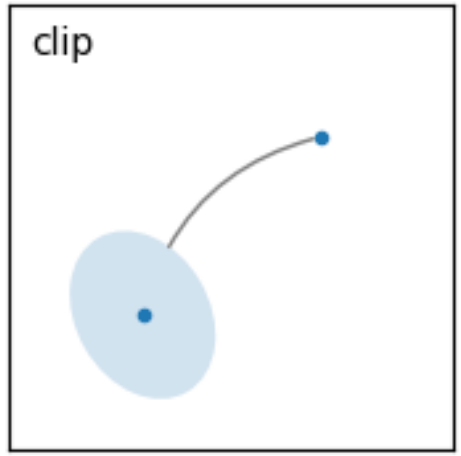
plt.show()
```

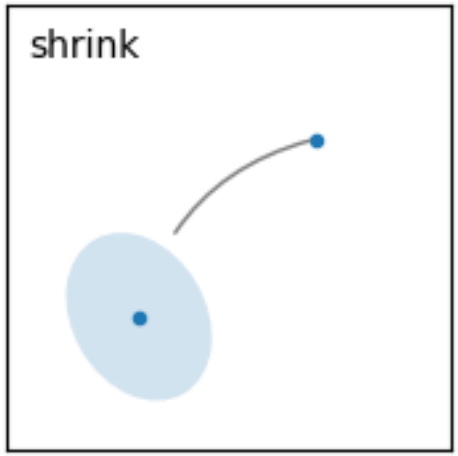
Annotating with Arrow

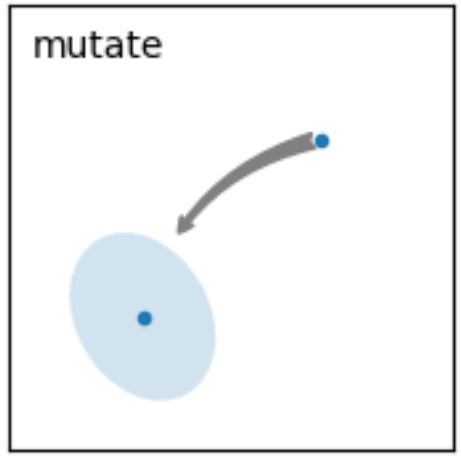
1. The arrow drawing takes a few steps.
   - a connecting path between two points are created. This is controlled by connectionstyle key value.
   - If patch object is given (patchA & patchB), the path is clipped to avoid the patch.
   - The path is further shrunk by given amount of pixels (shrinkA & shrinkB)
   - The path is transmuted to arrow patch, which is controlled by the arrowstyle key value.

示意图



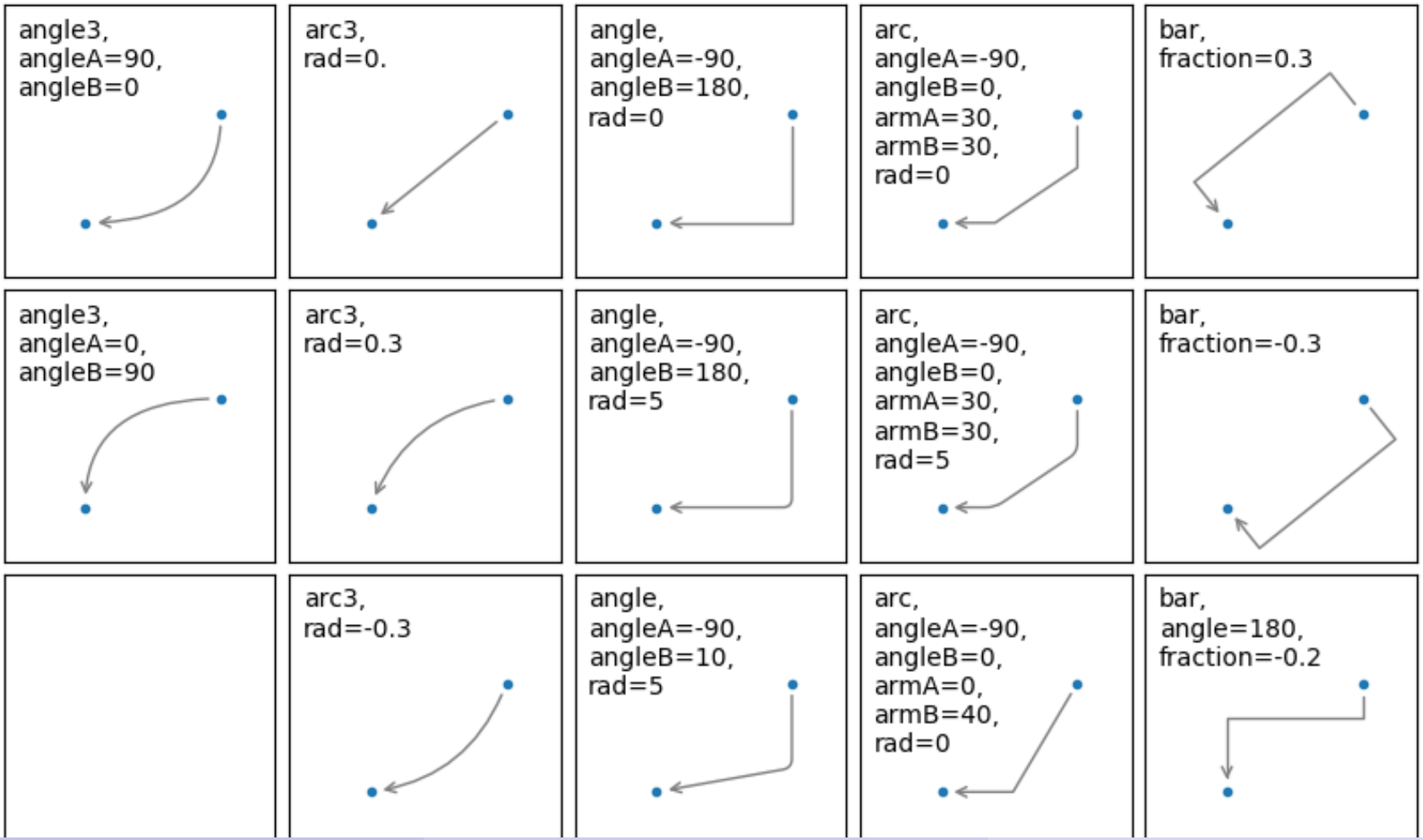






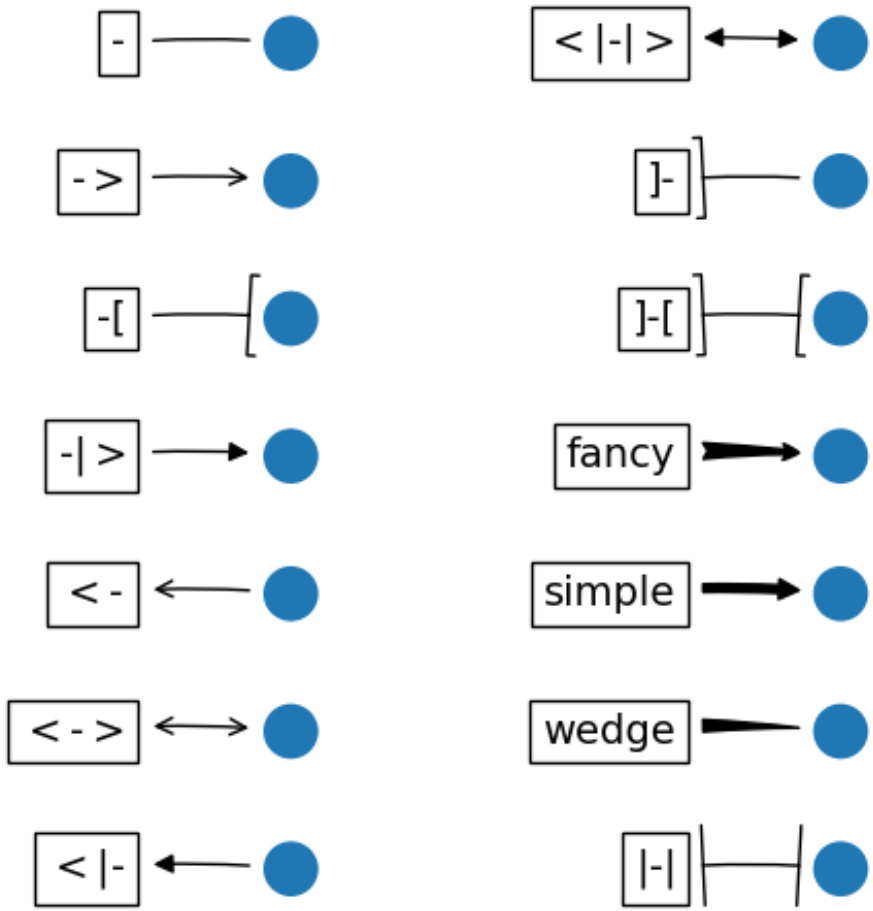
connectionstyle key

1. The creation of the connecting path between two points is controlled by connectionstyle key. The behavior of each connection style is (limitedly) demonstrated in the example below.



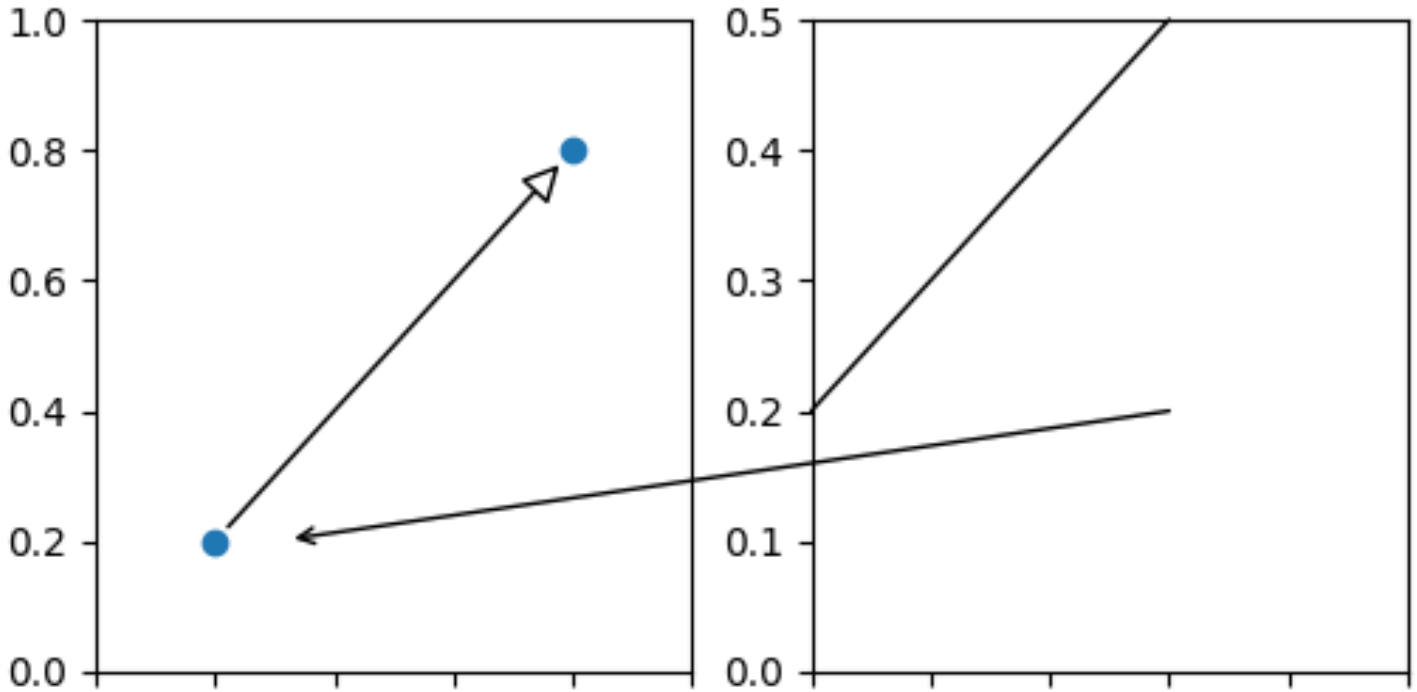
arrowstyle

1. The connecting path (after clipping and shrinking) is then mutated to an arrow patch, according to the given `arrowstyle`.



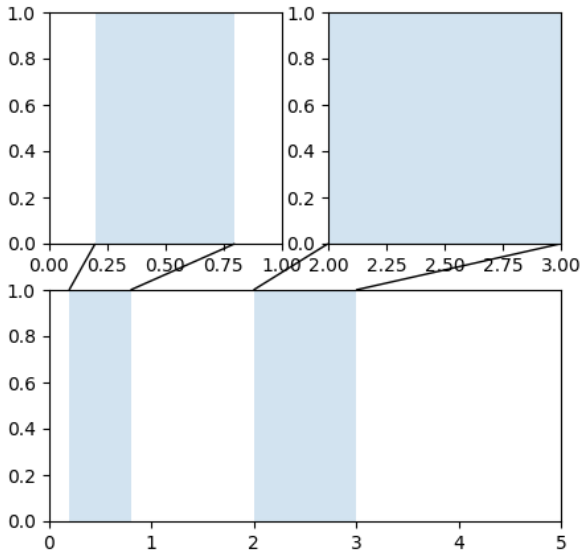
Using ConnectionPatch

1. The ConnectionPatch is like an annotation without text. While the annotate function is recommended in most situations, the `ConnectionPatch` is useful when you want to connect points in different axes.
2. `https://matplotlib.org/gallery/userdemo/connect_simple01.html`



Zoom effect between Axes

1. `mpl_toolkits.axes_grid1.inset_locator` defines some patch classes useful for interconnecting two axes.
2. Understanding the code requires some knowledge of how mpl's transform works.
3. `https://matplotlib.org/gallery/subplots_axes_and_figures/axes_zoom_effect.html`



How is mplot3d different from MayaVi?

1. 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.
2. 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.

Axes3D object

1. An Axes3D object is created just like any other axes using the `projection='3d'` keyword.
2. Create a new matplotlib.figure.Figure and add a new axes to it of type Axes3D:

```
1 import matplotlib.pyplot as plt
2 from mpl_toolkits.mplot3d import Axes3D
3 fig = plt.figure()
4 ax = fig.add_subplot(111, projection='3d')
```

一个例子

```
from mpl_toolkits.mplot3d import Axes3D
import numpy as np
import matplotlib.pyplot as plt

plt.rcParams['legend.fontsize'] = 10
fig = plt.figure()
ax = fig.gca(projection='3d')

# Prepare arrays x, y, z
theta = np.linspace(-4 * np.pi, 4 * np.pi, 100)
z = np.linspace(-2, 2, 100)
r = z**2 + 1
x = r * np.sin(theta)
y = r * np.cos(theta)

ax.plot(x, y, z, label='parametric curve')
ax.legend()

plt.show()
```

支持的 3D 图形类型

1. https://matplotlib.org/tutorials/toolkits/mplot3d.html#sphx-glr-tutorials-toolkits-mplot3d-py
2. Line plots :
   `Axes3D.plot(self, xs, ys, *args, zdir='z', **kwargs)`
3. Scatter plots :
   `Axes3D.scatter(self, xs, ys, zs=0, zdir='z', s=20, c=No`
4. Wireframe plots :
   `Axes3D.plot_wireframe(self, X, Y, Z, *args, **kwargs)`
5. Surface plots :
   `Axes3D.plot_surface(self, X, Y, Z, *args, norm=None, vm`
6. Tri-Surface plots :
   `Axes3D.plot_trisurf(self, *args, color=None, norm=None,`
7. Contour plots :
   `Axes3D.contour(self, X, Y, Z, *args, extend3d=False, st`

支持的 3D 图形类型

8. Filled contour plots :
   `Axes3D.contourf(self, X, Y, Z, *args, zdir='z', offset=`
9. Polygon plots :
   `Axes3D.add_collection3d(self, col, zs=0, zdir='z')`
10. Bar plots :
    `Axes3D.bar(self, left, height, zs=0, zdir='z', *args, *`
11. Quiver :
    `Axes3D.quiver(X, Y, Z, U, V, W, /, length=1, arrow_leng`
12. 2D plots in 3D
13. Text :
    `Axes3D.text(self, x, y, z, s, zdir=None, **kwargs)`
14. 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.

其他常见作图包

1. Pandas is handy for simple plots but you need to be willing to learn matplotlib to customize.
2. Seaborn can support some more complex visualization approaches but still requires matplotlib knowledge to tweak. The color schemes are a nice bonus.
3. ggplot ggplot is a plotting system for Python based on R's ggplot2 and the Grammar of Graphics. It is built for making profressional looking, plots quickly with minimal code.
4. Bokeh is an interactive visualization library for modern web browsers. It provides elegant, concise construction of versatile graphics, and affords high-performance interactivity over large or streaming datasets.
5. Mayavi: 3D scientific data visualization and plotting in Python.
6. Turtle graphics is a popular way for introducing programming to kids.