Practice 2:
Data Exploration

Practice 2.1: Matplotlib

Introduction

- Matplotlib is a Python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms.
- Matplotlib can be used in Python scripts, the Python and IPython shells, the Jupyter notebook, web application servers, and four graphical user interface toolkits.
- Webpage
- Tutorials
  - Examples tutorial

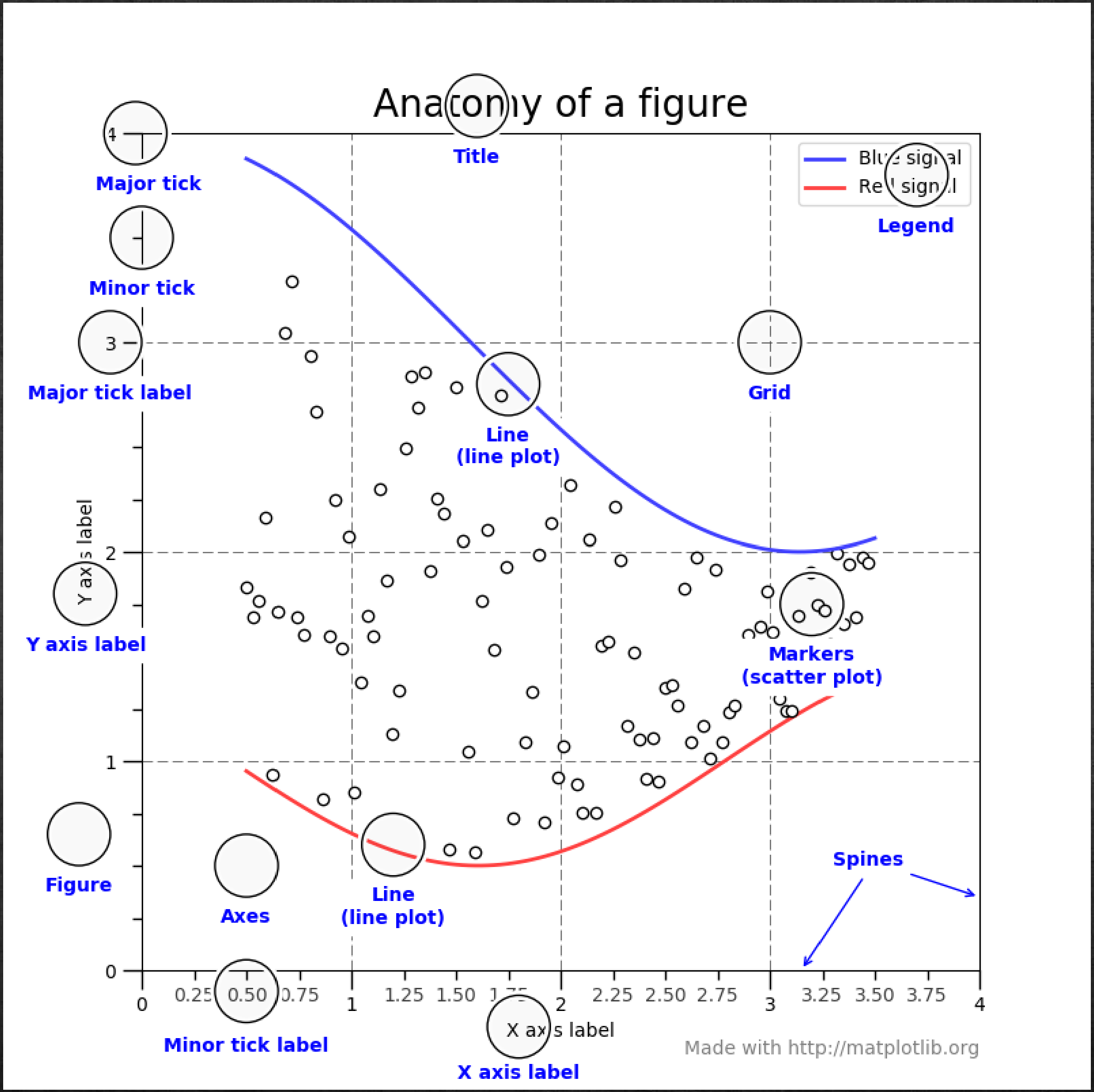
Matplotlib

- 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')
- 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.
- 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.

Starting

```
import matplotlib.pyplot as plt
import numpy as np
Import pandas as pd
```

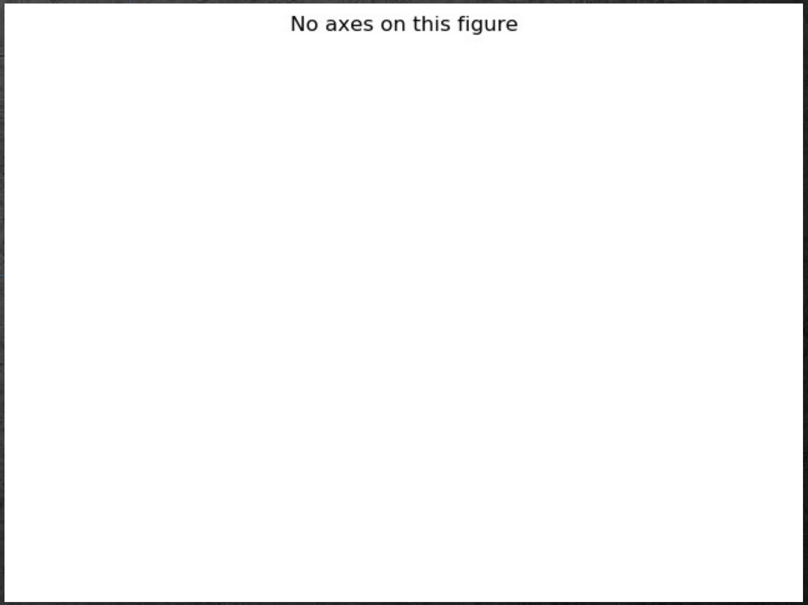
Parts of the figure



Figure

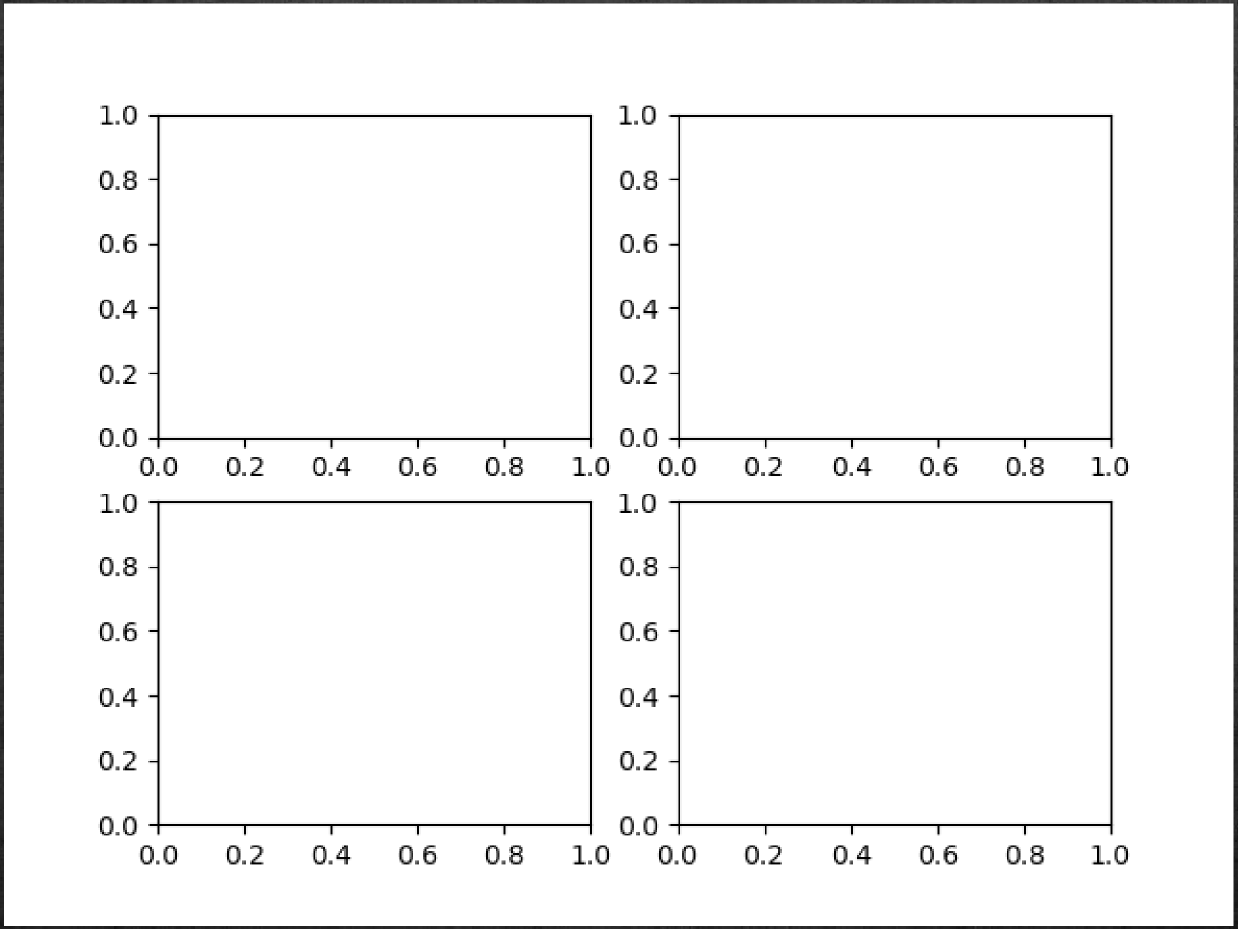
- The whole figure.
- The figure keeps track of all the child Axes, a smattering of 'special' artists (titles, figure legends, etc), and the canvas.
- A figure can have any number of Axes, but to be useful should have at least one.
- The easiest way to create a new figure is with pyplot:

```
>>> import matplotlib.pyplot as plt
>>> fig = plt.figure()
>>> fig.suptitle('No axes on this figure')
Text(0.5, 0.98, 'No axes on this figure')
>>> plt.show()
```



Figure

```
>>> fig, ax_lst = plt.subplots(2, 2)
```



Axes

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

Axis

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

Artist

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

Types of inputs to plotting functions

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

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

- and to convert a np.matrix

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

Matplotlib, pyplot and pylab: how are they related?

- Matplotlib is the whole package and matplotlib.pyplot is a module in Matplotlib.
- For functions in the pyplot module, there is always a "current" figure and axes (which is created automatically on request).
- For example, in the following example, the first call to plt.plot creates the axes, then subsequent calls to plt.plot add additional lines on the same axes, 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()
```

- pylab is deprecated
- For non-interactive plotting it is suggested to use pyplot to create the figures and then the OO interface for plotting.

Interactive modes

- Interactive mode: All plots are shown
  - plt.ion()
- Non-interactive mode: Plots are shown with plt.show()
  - plt.ioff()

Legends

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

Pyplot

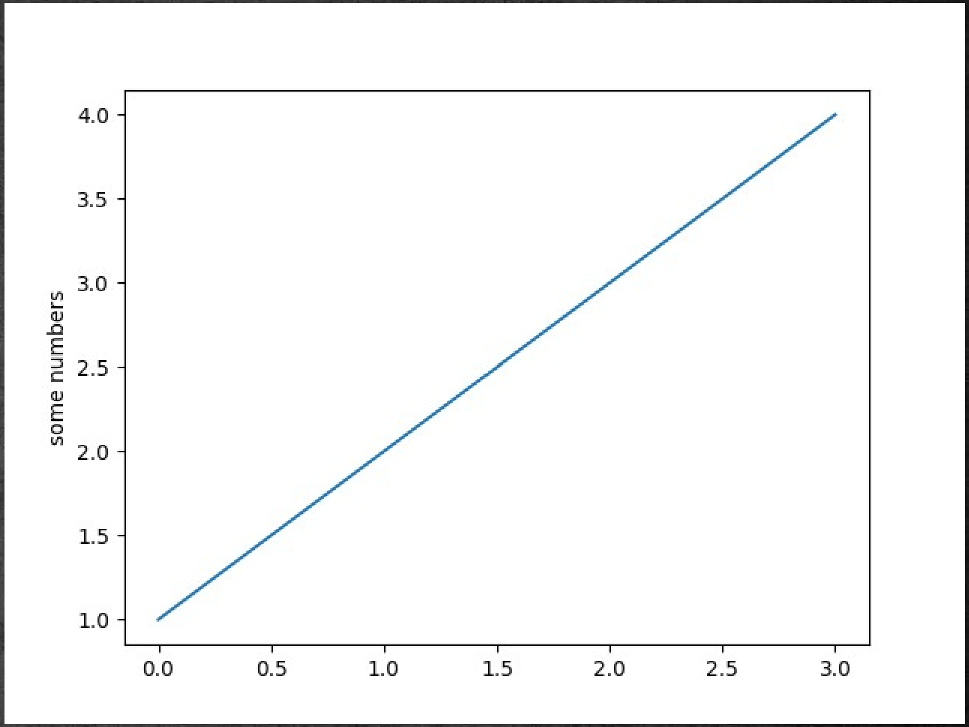
- matplotlib.pyplot is a collection of command style functions that make matplotlib work like MATLAB.
- Each pyplot function makes some change to a figure: e.g., creates a figure, creates a plotting area in a figure, plots some lines in a plotting area, decorates the plot with labels, etc.
- In matplotlib.pyplot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

Pyplot

- Generating visualizations with pyplot is very quick

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

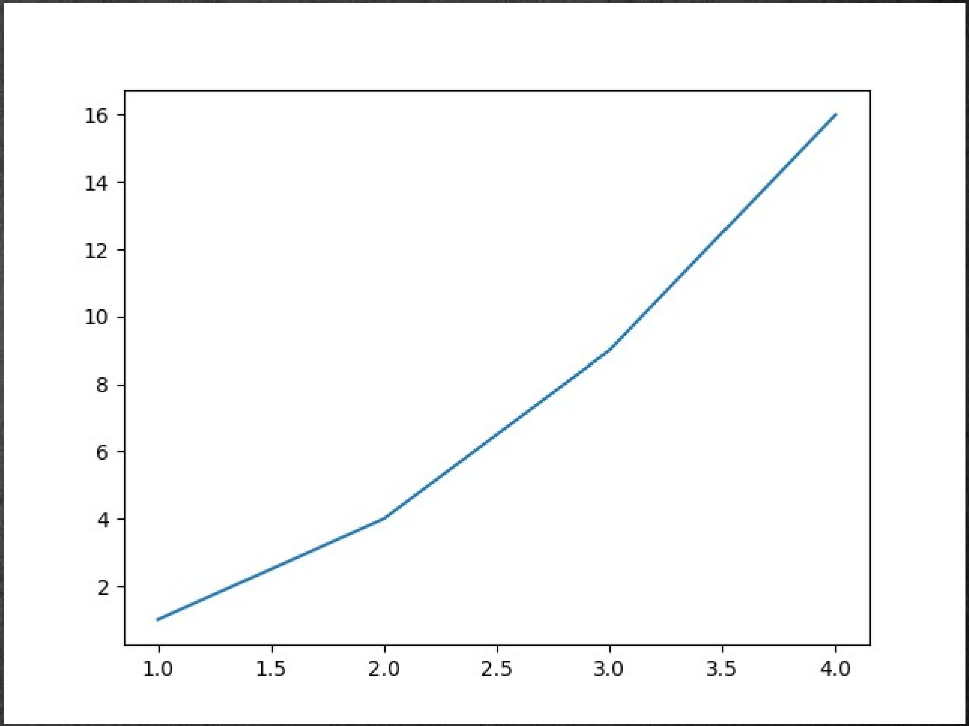
- If you provide a single list or array to the plot() command, matplotlib assumes it is a sequence of y values, and automatically generates the x values for you starting from 0.



Pyplot

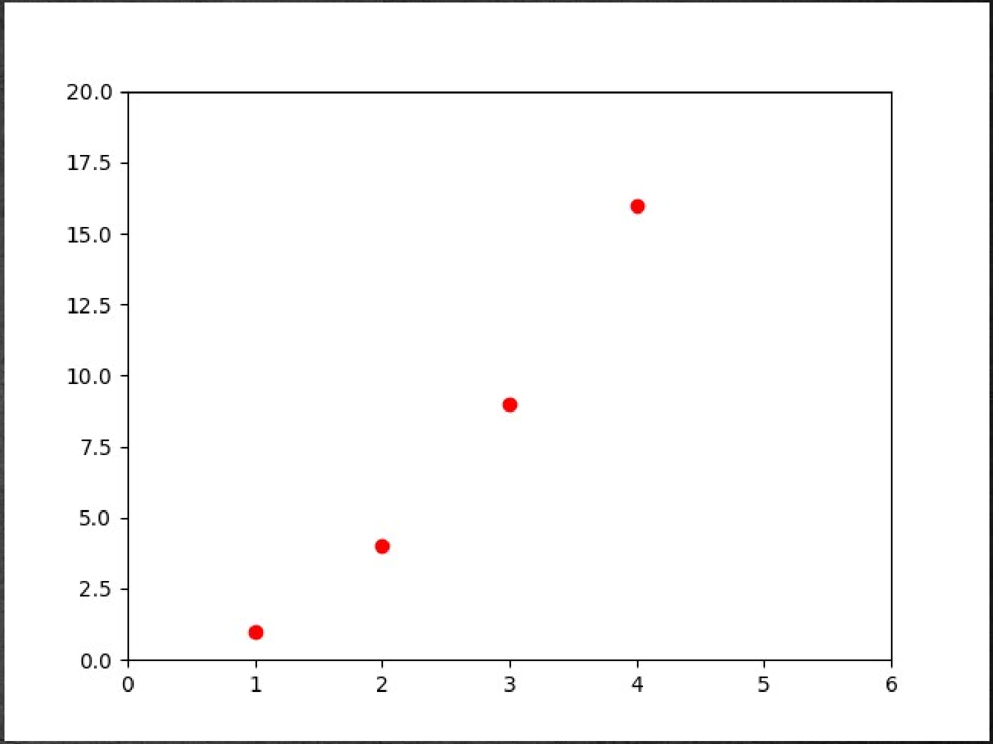
- plot() is a versatile command, and will take an arbitrary number of arguments.
- For example, to plot x versus y, you can issue the command

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



Formatting the style of your plot

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

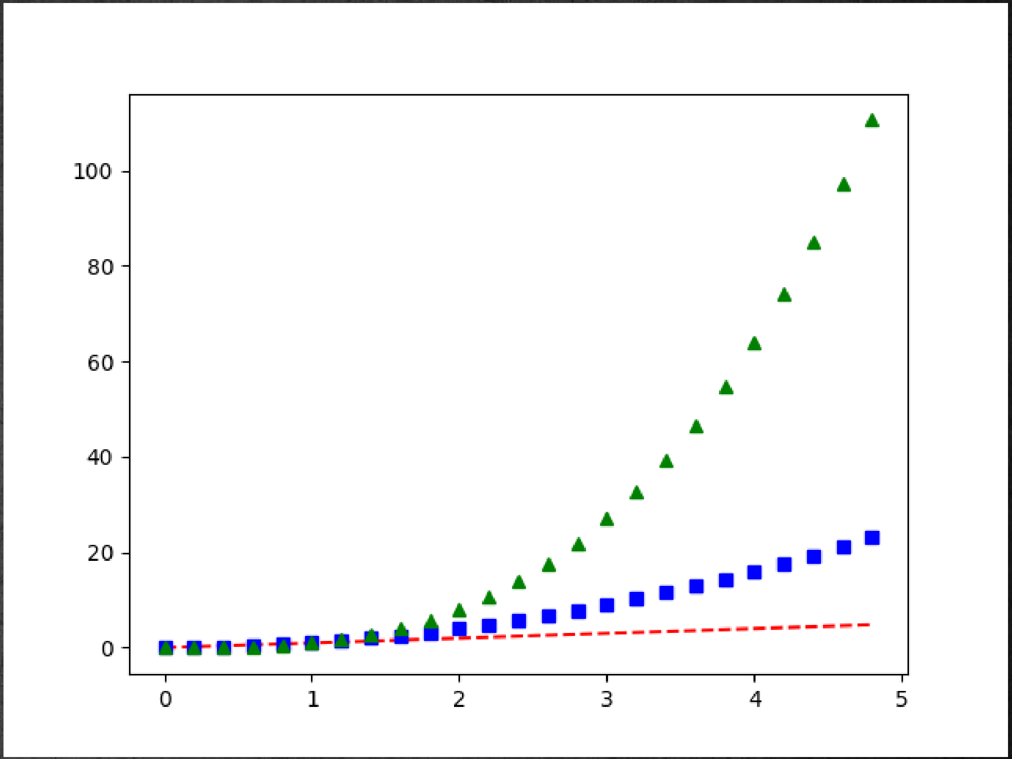


```
plt.plot([1, 2, 3, 4], [1, 4, 9, 16], 'ro')
plt.axis([0, 6, 0, 20])
plt.show()
```

Working with arrays

- 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.

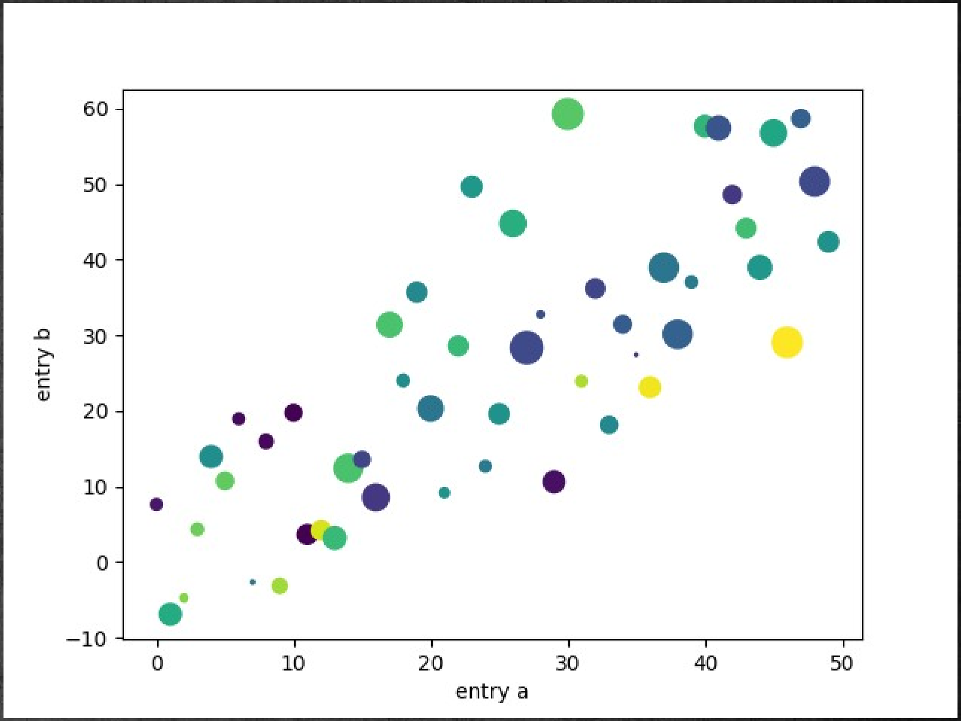
```
# evenly sampled time at 200ms intervals
t = np.arange(0., 5., 0.2)
# red dashes, blue squares and green triangles
plt.plot(t, t, 'r--', t, t**2, 'bs', t, t**3, 'g^')
plt.show()
```



Plotting with keyword strings

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

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

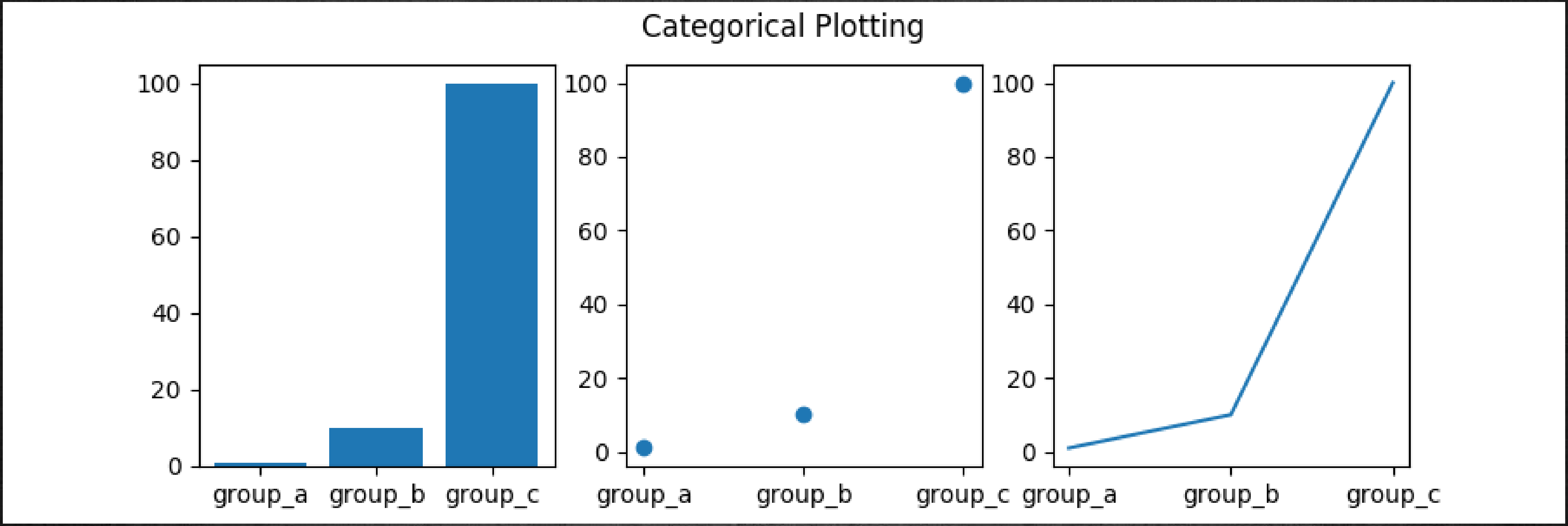


Plotting with categorical variables

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

```
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()
```

Plotting with categorical variables

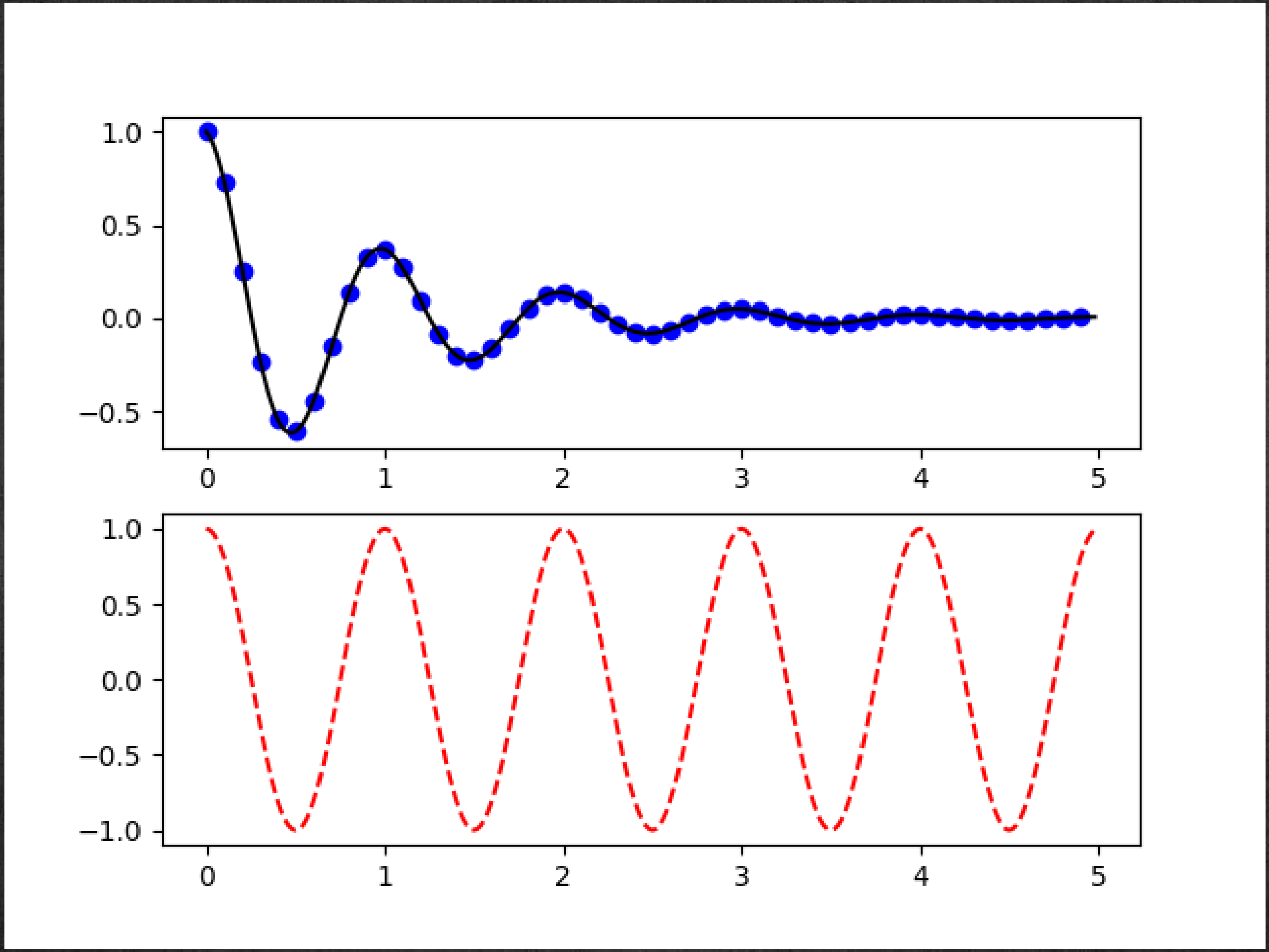


Working with multiple figures and axes

- 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.

```
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



Working with multiple figures and axes

- 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.
- The subplot() command specifies numrows, numcols, plot_number where plot_number 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, i.e., 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.

Working with multiple figures and axes

- You can create multiple figures by using multiple figure() calls with an increasing figure number.

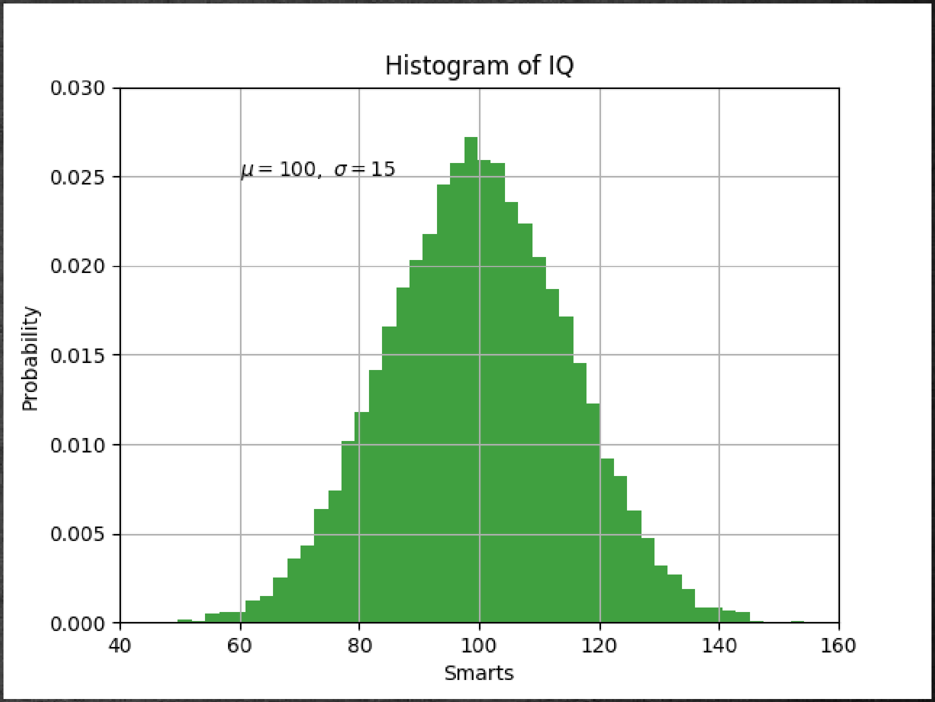
```
import matplotlib.pyplot as plt
plt.figure(1)                # the first figure
plt.subplot(211)             # the first subplot in the first figure
plt.plot([1, 2, 3])
plt.subplot(212)             # the second subplot in the first figure
plt.plot([4, 5, 6])
plt.figure(2)                # a second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

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

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 (

```
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

- matplotlib accepts TeX equation expressions in any text expression. For example to write the expression $\sigma_i=15$ in the title, you can write a TeX expression surrounded by dollar signs:
  `plt.title(r'$\sigma_i=15$')`
- The r preceding the title string is important -- it signifies that the string is a raw string and not to treat backslashes as python escapes.
- matplotlib has a built-in TeX expression parser and layout engine, and ships its own math fonts.
- Thus you can use mathematical text across platforms without requiring a TeX installation.

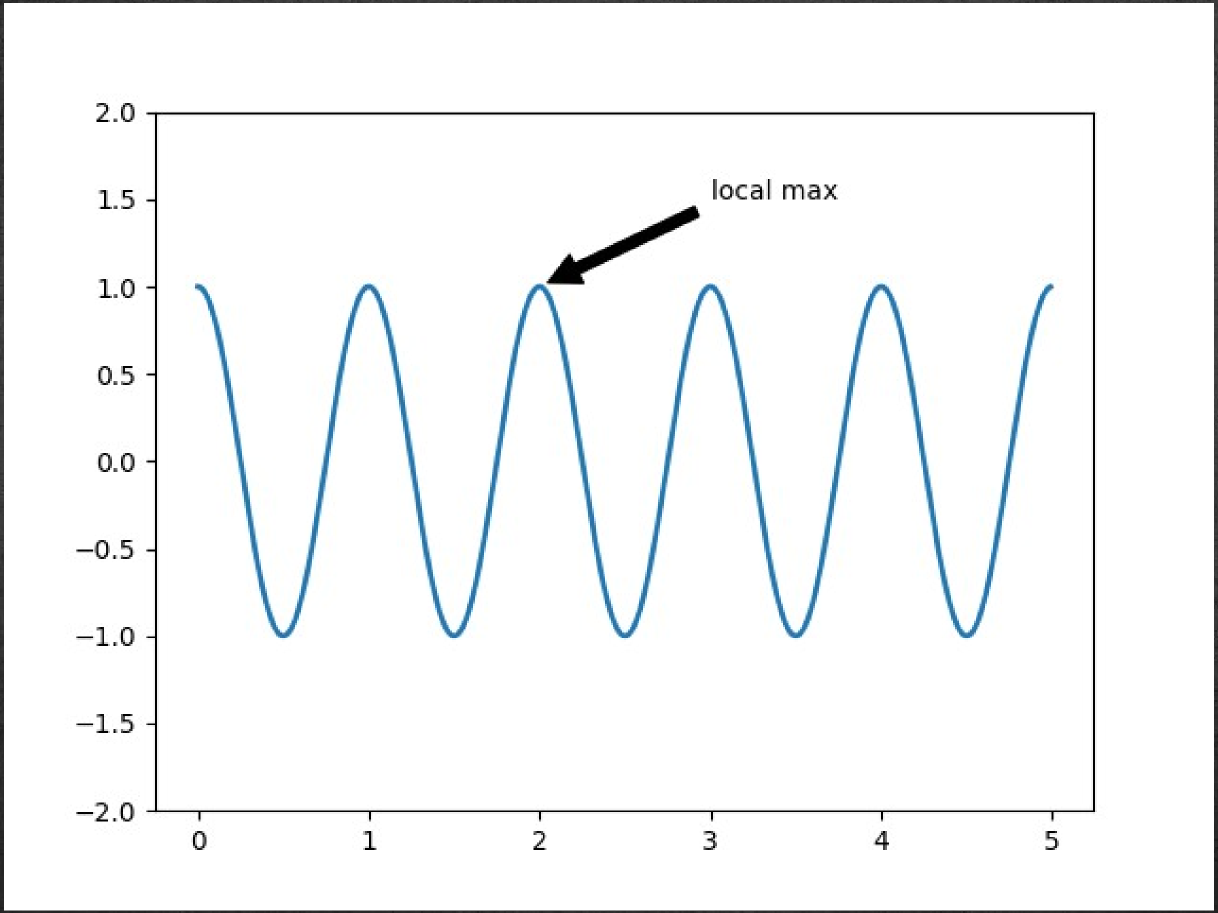
Annotating text

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

```
ax = plt.subplot(111)
t = np.arange(0.0, 5.0, 0.01)
s = np.cos(2*np.pi*t)
line, = plt.plot(t, s, lw=2)
plt.annotate('local max', xy=(2, 1), xytext=(3, 1.5),
             arrowprops=dict(facecolor='black', shrink=0.05),
             )
plt.ylim(-2, 2)
plt.show()
```

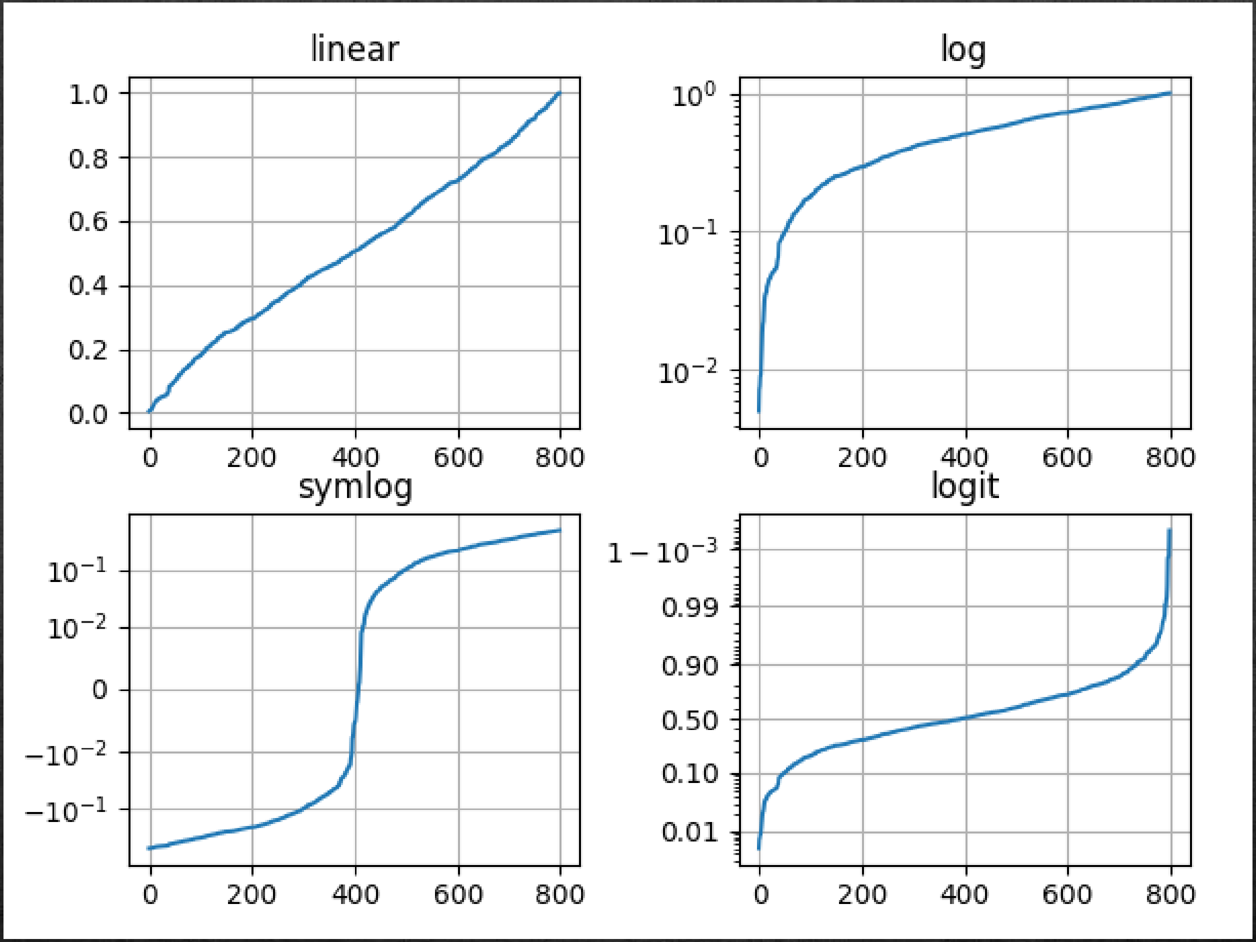
Annotating text

- 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 -- see Basic annotation and Advanced Annotation for details.



Logarithmic and other nonlinear axes

- matplotlib.pyplot supports not only linear axis scales, but also logarithmic and logit scales.
  plt.xscale('log')


linear
log
symlog
logit
10^0
10^{-1}
10^{-2}
10^{-1}
10^{-2}
-10^{-2}
-10^{-1}
$1-10^{-3}$