

STAT606

Computing for Data Science and Statistics

Lecture 12: `matplotlib`

Plotting with `matplotlib`

`matplotlib` is a plotting library for use in Python

Similar to R's `ggplot2` and MATLAB's plotting functions

For MATLAB fans, `matplotlib.pyplot` implements MATLAB-like plotting:

http://matplotlib.org/users/pyplot_tutorial.html

Sample plots with code:

http://matplotlib.org/tutorials/introductory/sample_plots.html

Basic plotting: `matplotlib.pyplot.plot`

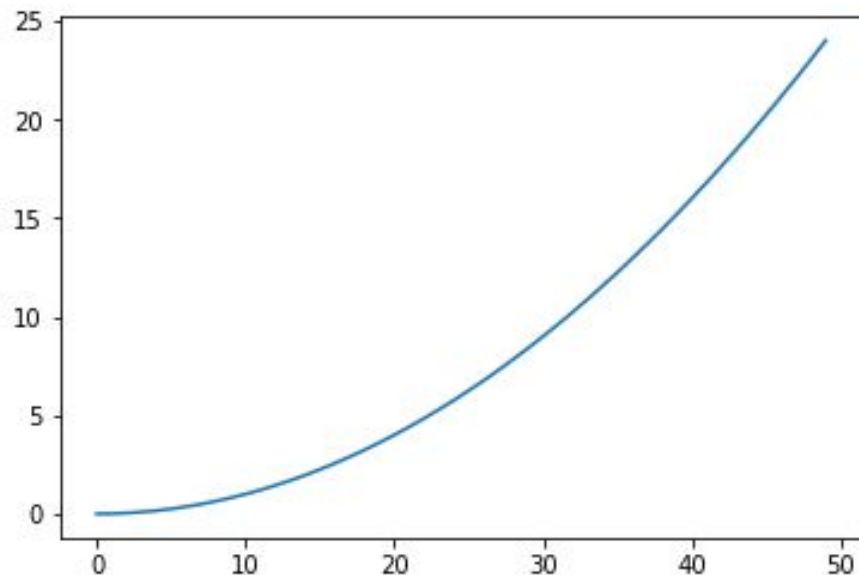
`matplotlib.pyplot.plot(x, y)`

plots `y` as a function of `x`.

`matplotlib.pyplot(t)`

sets x-axis to `np.arange(len(t))`

```
1 import matplotlib as mp
2 import matplotlib.pyplot as plt
3 %matplotlib inline
4 x = np.arange(0,5,0.1, dtype='float')
5 _ = plt.plot(x**2)
```

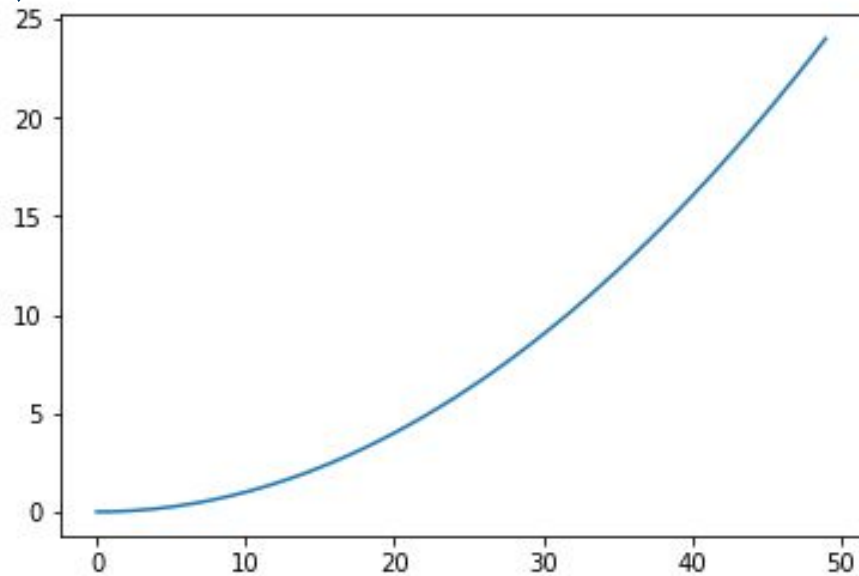


Basic plotting: `matplotlib.pyplot.plot`

Jupyter “magic” command to make images appear in-line.

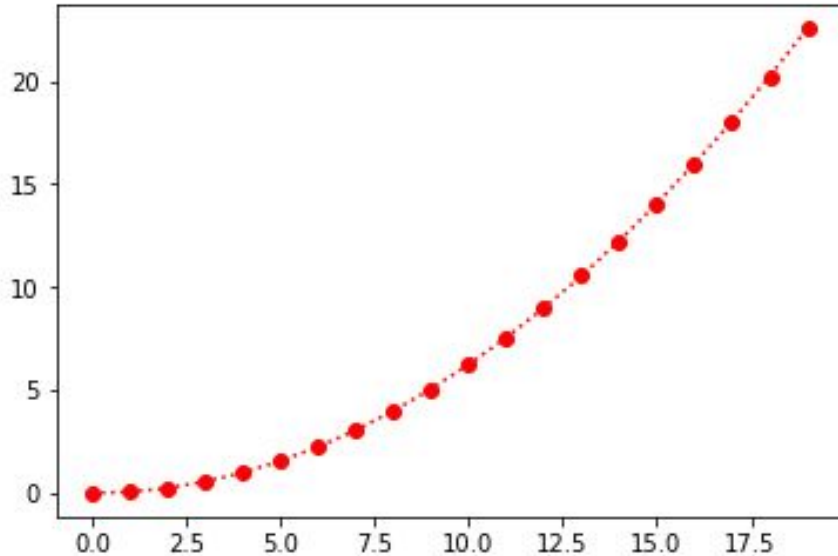
Reminder: Python `\_` is a placeholder, similar to MATLAB `\~`. Tells Python to treat this like variable assignment, but don't store result anywhere.

```
1 import matplotlib as mp
2 import matplotlib.pyplot as plt
3 %matplotlib inline
4 x = np.arange(0,5,0.1, dtype='float')
5 _ = plt.plot(x**2)
```



Customizing plots

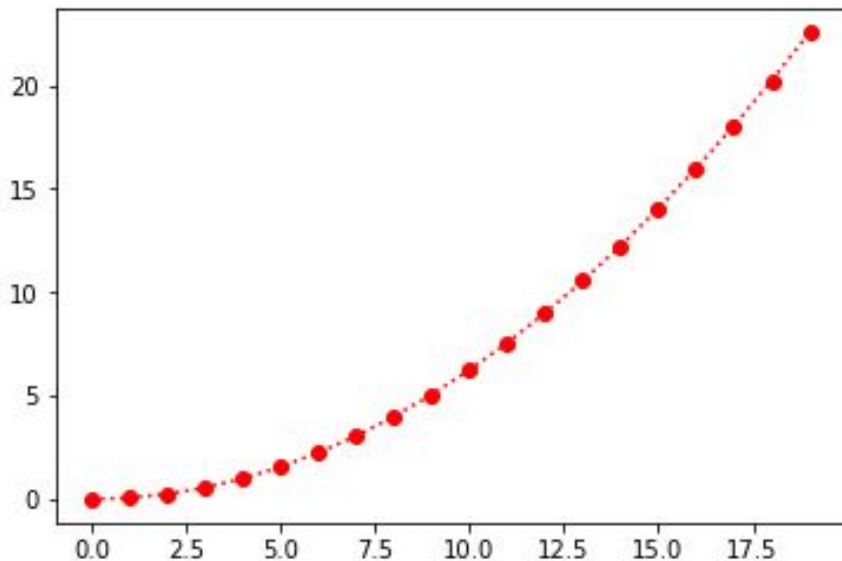
```
1 x = np.arange(0,5,0.25, dtype='float')  
2 _ = plt.plot(x**2, ':ro')
```



Second argument to `pyplot.plot` specifies line type, line color, and marker type.

Customizing plots

```
1 x = np.arange(0,5,0.25, dtype='float')
2 _ = plt.plot(x**2, color='red', linestyle=':', marker='o')
```

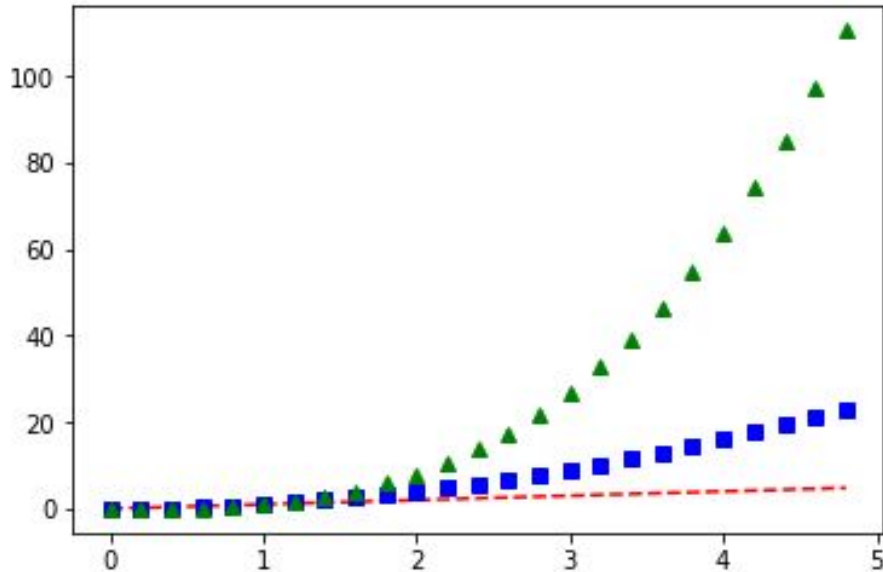


Long form of the command on the previous slide. Same plot!

A full list of the long-form arguments available to `pyplot.plot` are available in the table titled “Here are the available Line2D properties.”: http://matplotlib.org/users/pyplot_tutorial.html

Multiple lines in a single plot

```
1 t = np.arange(0., 5., 0.2)
2 # plt.plot(xvals, y1vals, traits1, y2vals, traits2, ... )
3 _ = plt.plot(t, t, 'r--', t, t**2, 'bs', t, t**3, 'g^')
```

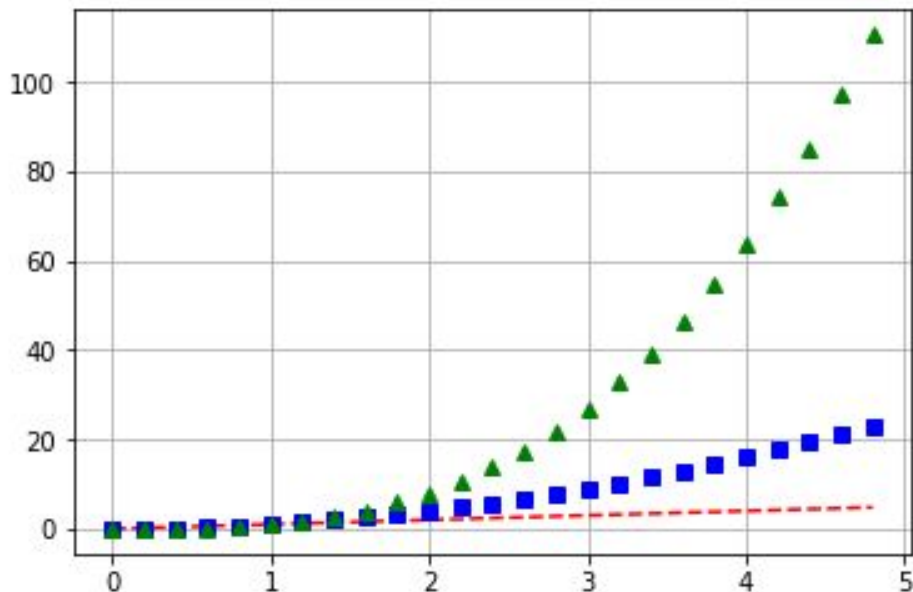


Note: more complicated specification of individual lines can be achieved by adding them to the plot one at a time.

Multiple lines in a single plot: long form

```
1 t = np.arange(0., 5., 0.2)
2 plt.grid()
3 plt.plot(t, t, 'r--')
4 plt.plot(t, t**2, 'bs')
5 plt.plot(t, t**3, 'g^')
6 _ = plt.show()
```

plt.grid turns grid lines on/off.

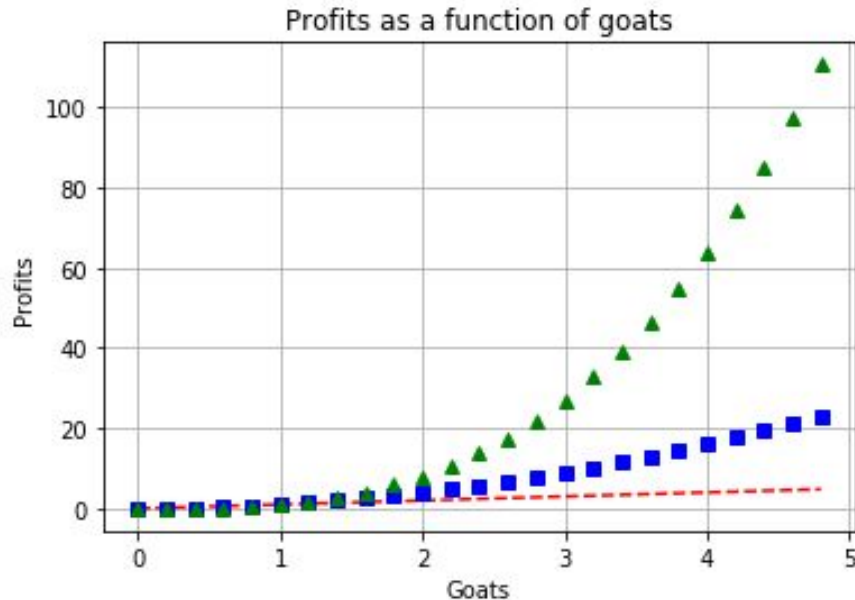


Note: same plot as previous slide, but specifying one line at a time so we could, if we wanted, use more complicated line attributes.

Titles and axis labels

```
1 t = np.arange(0., 5., 0.2)
2 plt.grid()
3 plt.plot(t, t, 'r--', t, t**2, 'bs', t, t**3, 'g^')
4 plt.title('Profits as a function of goats')
5 plt.xlabel('Goats')
6 plt.ylabel('Profits')
7 _ = plt.show()
```

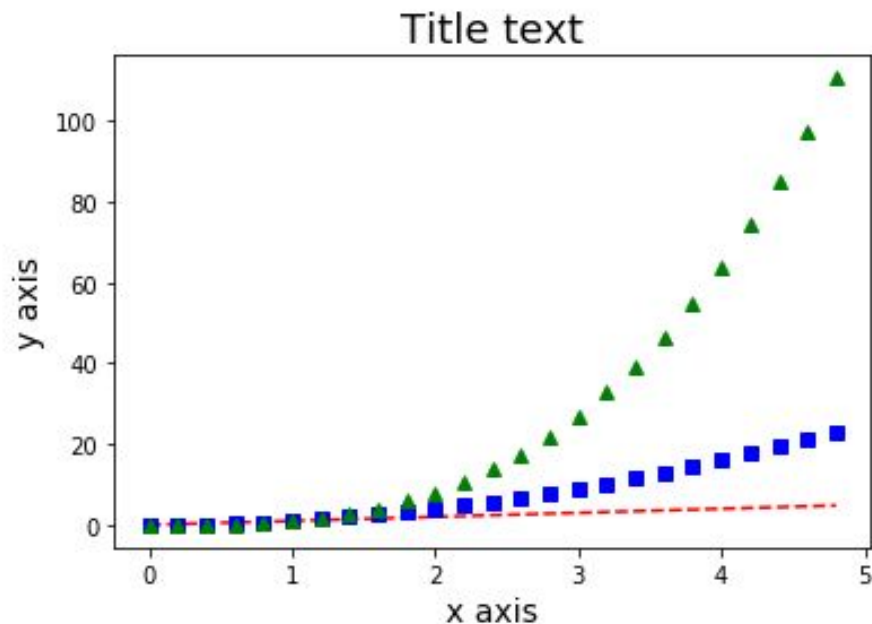
Specifying titles and axis labels couldn't be more straight-forward.



Titles and axis labels

```
1 t = np.arange(0., 5., 0.2)
2 plt.title('Title text', fontsize=18)
3 plt.xlabel('x axis', fontsize=14)
4 plt.ylabel('y axis', fontsize=14)
5 _ = plt.plot(t, t, 'r--', t, t**2, 'bs', t, t**3, 'g^')
```

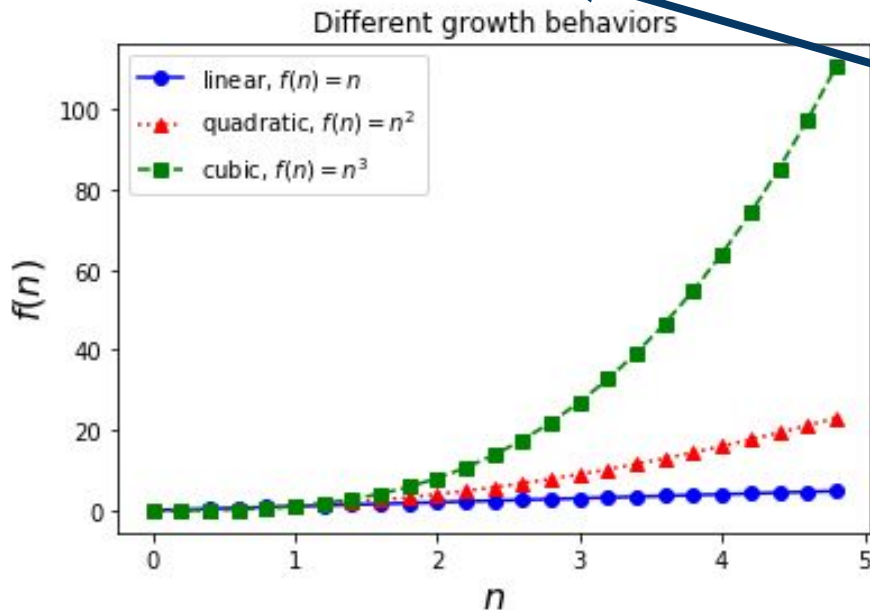
Change font sizes



Legends

```
1 plt.xlabel("$n$", fontsize=16) # set the axes labels
2 plt.ylabel("$f(n)$", fontsize=16)
3 plt.title("Different growth behaviors") # set the plot title
4 plt.plot(t, t, '-ob', label='linear, $f(n)=n$')
5 plt.plot(t, t**2, ':^r', label='quadratic, $f(n)=n^2$')
6 plt.plot(t, t**3, '--sg', label='cubic, $f(n)=n^3$')
7 _ = plt.legend(loc='best') # places legend at best location
```

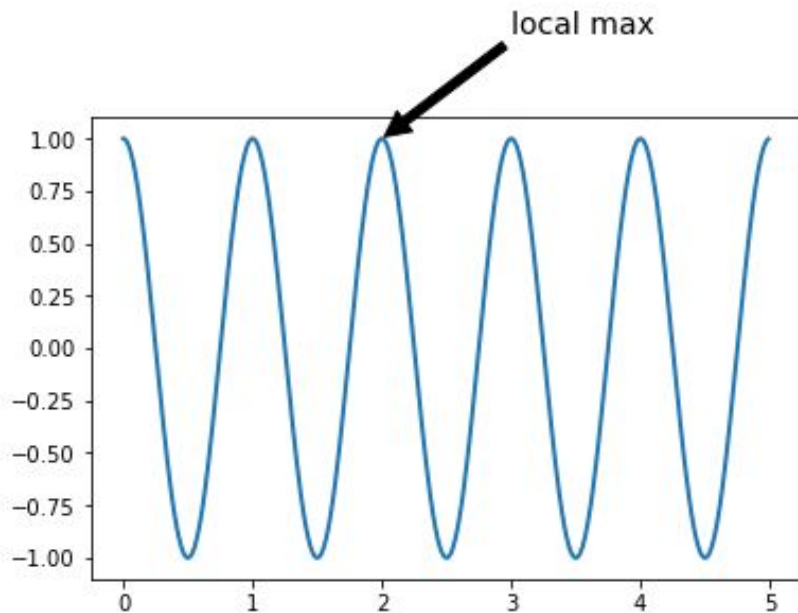
Can use LaTeX in labels, titles, etc.



`pyplot.legend` generates legend based on label arguments passed to `pyplot.plot`. `loc='best'` tells `pyplot` to place the legend where it thinks is best.

Annotating figures

```
1 t = np.arange(0.0, 5.0, 0.01)
2 s = np.cos(2*np.pi*t) #np.pi==3.14159...
3 plt.plot(t, s, lw=2) # plot the cosine.
4 # Annotate the figure with an arrow and text.
5 _ = plt.annotate('local max', xy=(2, 1), xytext=(3, 1.5),
6                 fontsize=14,
7                 arrowprops=dict(facecolor='black', shrink=0.02) )
```

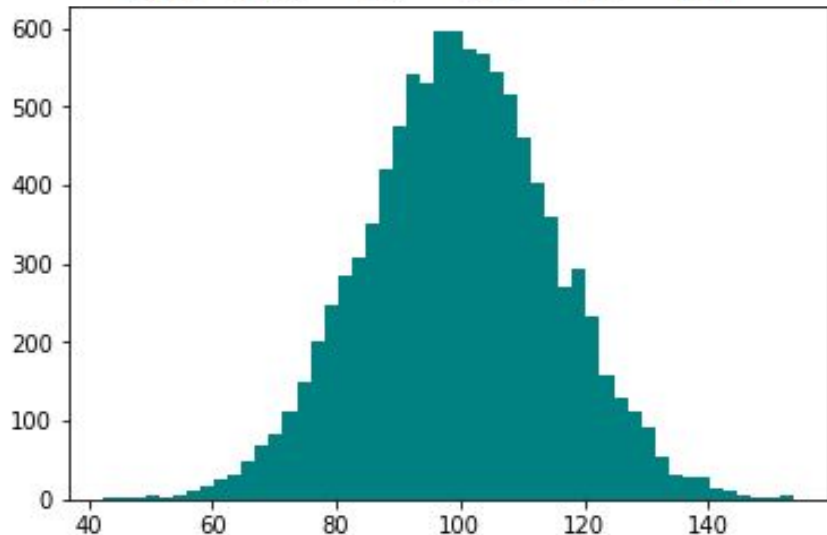


Specify text coordinates and coordinates of the arrowhead using the *coordinates of the plot itself*. This is pleasantly different from many other plotting packages, which require specifying coordinates in pixels or inches/cms.

Plotting histograms: `pyplot.hist()`

```
1 mu, sigma = (100, 15)
2 x = np.random.normal(mu, sigma, 10000)
3 # hist( data, nbins, ... )
4 (n, bins, patches) = plt.hist(x, 50, density=False, facecolor='teal')
5 n
```

```
array([ 1.,  1.,  2.,  4.,  3.,  5., 11., 18., 26., 30., 47.,
        68., 82., 113., 150., 201., 246., 285., 309., 352., 420., 475.,
        541., 529., 597., 595., 572., 566., 543., 515., 462., 404., 360.,
        270., 294., 233., 159., 128., 111., 92., 54., 32., 28., 28.,
        15., 11.,  5.,  2.,  1.,  4.] )
```



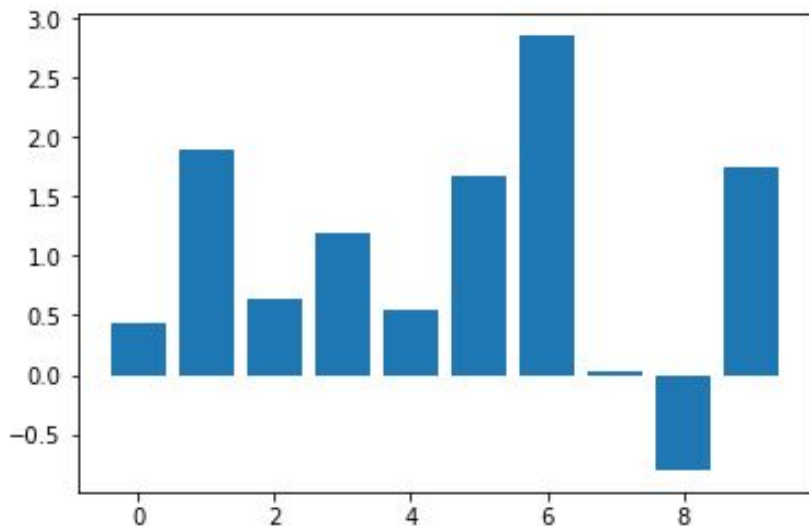
Bin counts. Note that if `density=True`, then these will be chosen so that the histogram “integrates” to 1.

https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.hist.html

Bar plots

```
bar(x, height, *, align='center', **kwargs)
```

```
1 t = np.arange(10)
2 s = np.random.normal(1,1,10)
3 _ = plt.bar(t, s, align='center')
```



Full set of available arguments to

`bar(...)` can be found at

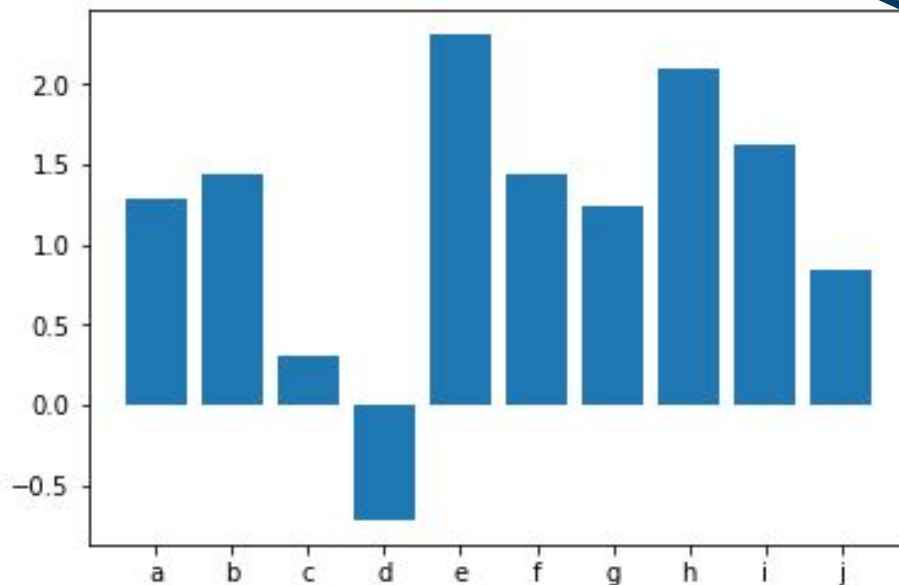
http://matplotlib.org/api/_as_gen/matplotlib.pyplot.bar.html#matplotlib.pyplot.bar

Horizontal analogue given by `barh`

http://matplotlib.org/api/_as_gen/matplotlib.pyplot.barh.html#matplotlib.pyplot.barh

Tick labels

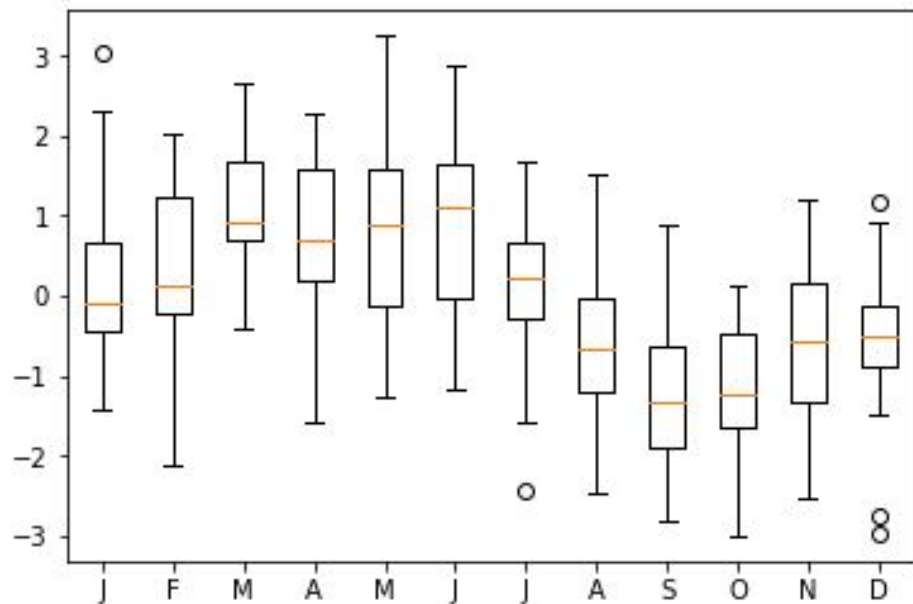
```
1 import string
2 t = np.arange(10)
3 s = np.random.normal(1,1,10)
4 mylabels = list(string.ascii_lowercase[0:len(t)])
5 _ = plt.bar(t, s, tick_label=mylabels, align='center')
```



Can specify what the x-axis tick labels should be by using the `tick_label` argument to plot functions.

Box & whisker plots

```
1 K=12; n=25
2 draws = np.zeros((n,K))
3 for k in range(K):
4     mu = np.sin(2*np.pi*k/K)
5     draws[:,k] = np.random.normal(mu,1,n)
6 _ = plt.boxplot(draws, labels=list('JFMAMJJASOND'))
```



`plt.boxplot(x, ...)` : `x` is the data.
Many more optional arguments are available,
most to do with how to compute medians,
confidence intervals, whiskers, etc. See
http://matplotlib.org/api/as_gen/matplotlib.pyplot.boxplot.html#matplotlib.pyplot.boxplot

Pie Charts

Don't use pie charts!

A table is nearly always better than a dumb pie chart; the only worse design than a pie chart is several of them, for then the viewer is asked to compare quantities located in spatial disarray both within and between charts [...]
Given their low [information] density and failure to order numbers along a visual dimension, pie charts should never be used.

Edward Tufte

The Visual Display of Quantitative Information

But if you must...

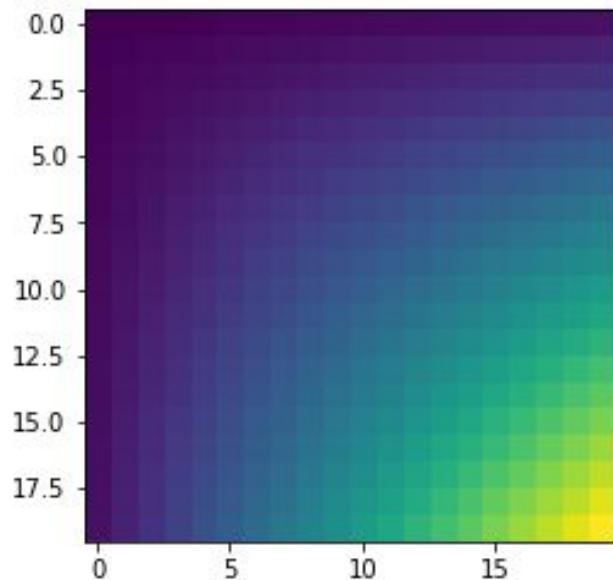
```
pyplot.pie(x, ... )
```

http://matplotlib.org/api/_as_gen/matplotlib.pyplot.pie.html#matplotlib.pyplot.pie



Heatmaps and tiling

```
1 n=20
2 x = np.arange(1,n+1)
3 M = x*np.reshape(x, (n,1))
4 _ = plt.imshow(M)
```

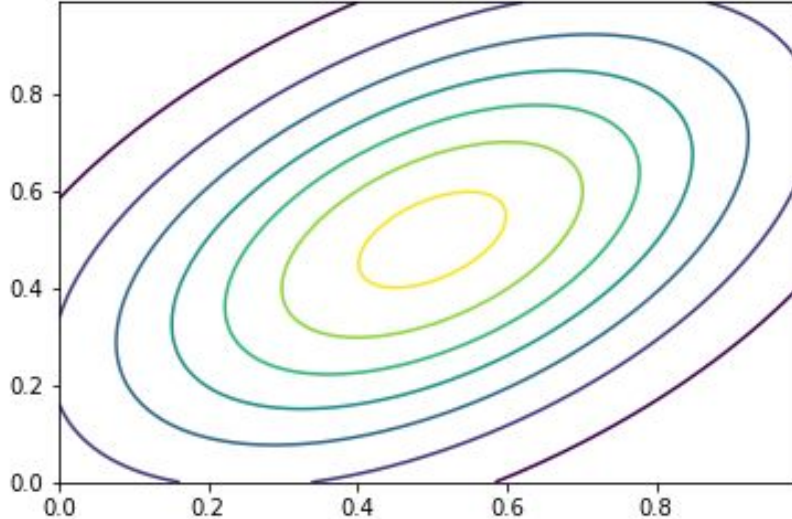


`imshow` is matplotlib analogue of MATLAB's `imagesc`, R's `image`. Lots of optional extra arguments for changing scale, color scheme, etc. See documentation: https://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.imshow

Drawing contours

```
1 mu=np.array([0.5,0.5])
2 Sigma=np.array([[0.1,0.05],[0.05,0.1]])
3 mvn1 = scipy.stats.multivariate_normal(mu,Sigma)
4
5 x, y = np.mgrid[0:1:.01, 0:1:.01]
6 pos = np.empty(x.shape + (2,))
7 pos[:, :, 0] = x; pos[:, :, 1] = y
8
9 _ = plt.contour(x, y, mvn1.pdf(pos))
```

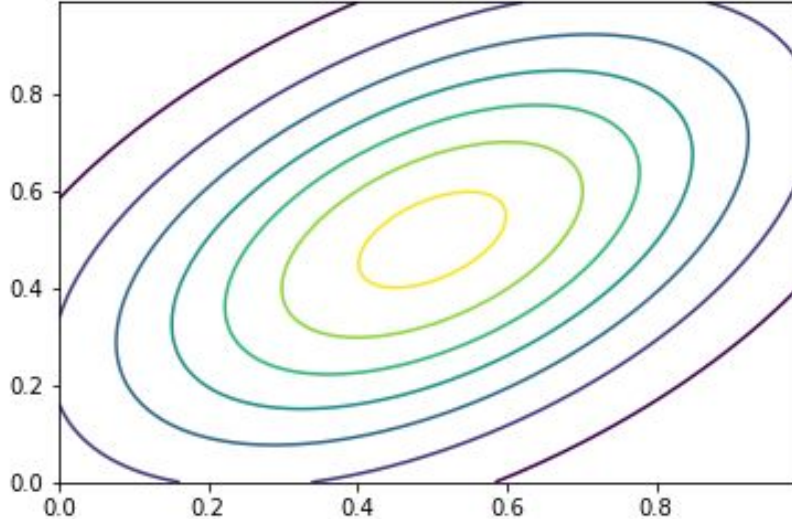
These three lines create an object, `mvn1`, representing a multivariate normal distribution.



Drawing contours

```
1 mu=np.array([0.5,0.5])
2 Sigma=np.array([[0.1,0.05],[0.05,0.1]])
3 mvn1 = scipy.stats.multivariate_normal(mu,Sigma)
4
5 x, y = np.mgrid[0:1:.01, 0:1:.01]
6 pos = np.empty(x.shape + (2,))
7 pos[:, :, 0] = x; pos[:, :, 1] = y
8
9 _ = plt.contour(x, y, mvn1.pdf(pos))
```

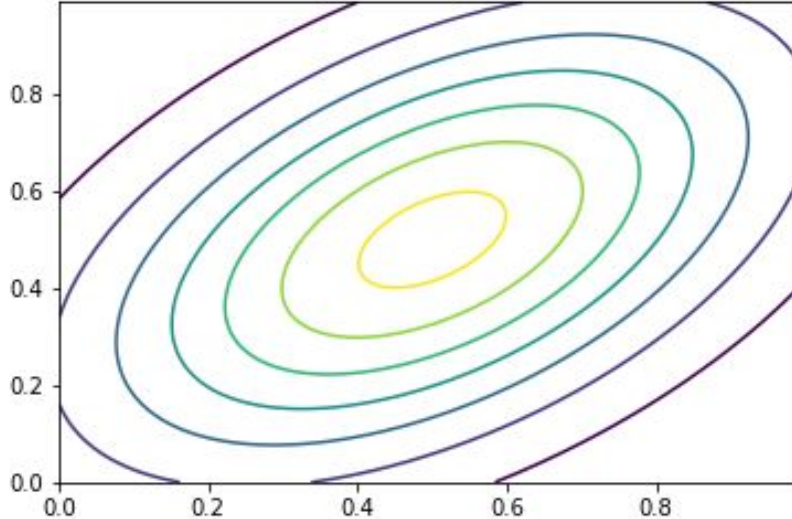
`mgrid` is short for “mesh grid”. Note the syntax: square brackets instead of parentheses. `mgrid` is an object, not a function!



Drawing contours

```
1 mu=np.array([0.5,0.5])
2 Sigma=np.array([[0.1,0.05],[0.05,0.1]])
3 mvn1 = scipy.stats.multivariate_normal(mu,Sigma)
4
5 x, y = np.mgrid[0:1:.01, 0:1:.01]
6 pos = np.empty(x.shape + (2,))
7 pos[:, :, 0] = x; pos[:, :, 1] = y
8
9 _ = plt.contour(x, y, mvn1.pdf(pos))
```

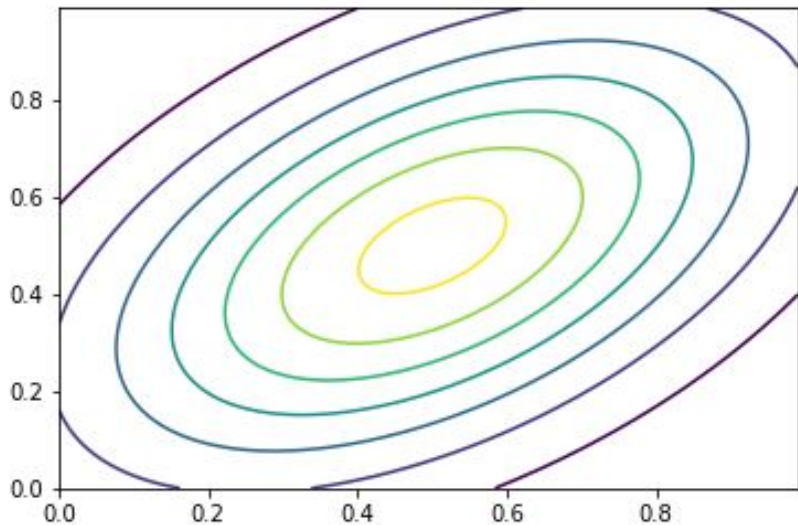
Here, `mgrid` stores a grid of (x,y) pairs, so this line actually generates a 100-by-100 grid of (x,y) coordinates, hence the tuple assignment.



Drawing contours

```
1 mu=np.array([0.5,0.5])
2 Sigma=np.array([[0.1,0.05],[0.05,0.1]])
3 mvn1 = scipy.stats.multivariate_normal(mu,Sigma)
4
5 x, y = np.mgrid[0:1:.01, 0:1:.01]
6 pos = np.empty(x.shape + (2,))
7 pos[:, :, 0] = x; pos[:, :, 1] = y
8
9 _ = plt.contour(x, y, mvn1.pdf(pos))
```

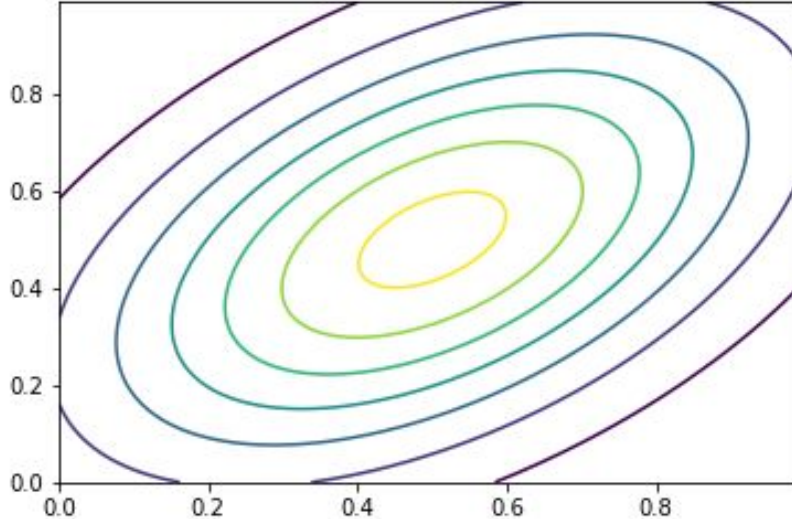
`pos` is a 3-dimensional array. Like a box of numbers. We're going to plot a surface, but at each (x,y) coordinate, the surface value depends on both x and y.



Drawing contours

```
1 mu=np.array([0.5,0.5])
2 Sigma=np.array([[0.1,0.05],[0.05,0.1]])
3 mvn1 = scipy.stats.multivariate_normal(mu,Sigma)
4
5 x, y = np.mgrid[0:1:.01, 0:1:.01]
6 pos = np.empty(x.shape + (2,))
7 pos[:, :, 0] = x; pos[:, :, 1] = y
8
9 _ = plt.contour(x, y, mvn1.pdf(pos))
```

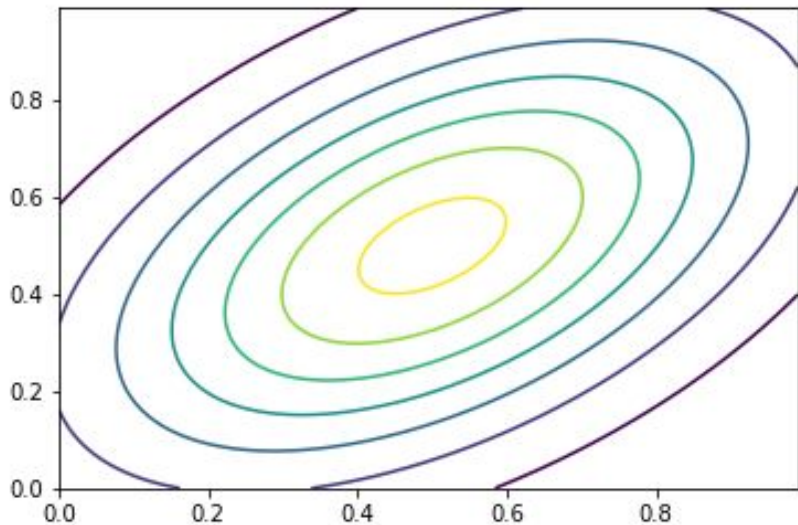
The reason for building `pos` the way we did is apparent if we read the documentation for `scipy.stats.(dist).pdf`.



Drawing contours

```
1 mu=np.array([0.5,0.5])
2 Sigma=np.array([[0.1,0.05],[0.05,0.1]])
3 mvn1 = scipy.stats.multivariate_normal(mu,Sigma)
4
5 x, y = np.mgrid[0:1:.01, 0:1:.01]
6 pos = np.empty(x.shape + (2,))
7 pos[:, :, 0] = x; pos[:, :, 1] = y
8
9 _ = plt.contour(x, y, mvn1.pdf(pos))
```

`matplotlib.contour` takes a set of x coordinates, a set of y coordinates, and an array of their corresponding values.



`matplotlib.contour` offers plenty of optional arguments for changing color schemes, spacing of contour lines, etc.
https://matplotlib.org/api/contour_api.html

Subplots

`subplot(nrows, ncols, plot_number)`

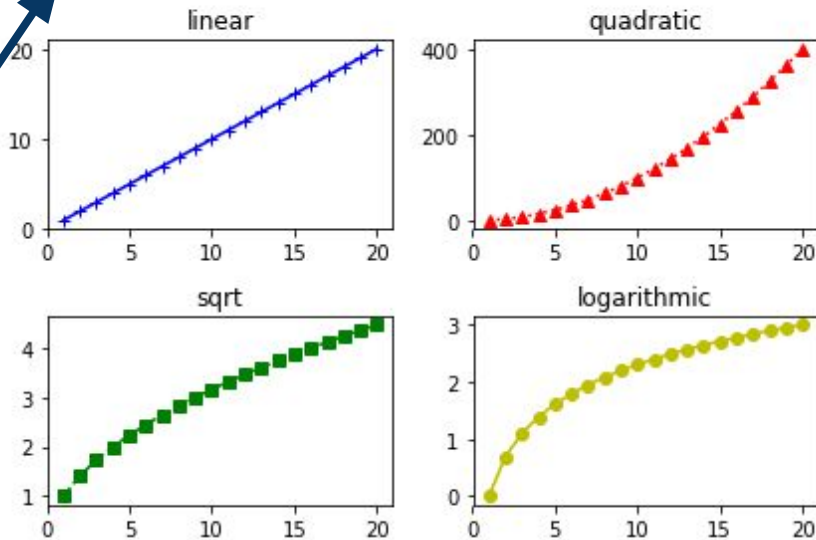
Shorthand: `subplot(XYZ)`

Makes an X-by-Y plot

Picks out the Z-th plot

Counting in row-major order

```
1 t=np.arange(20)+1
2 plt.subplot(221)
3 plt.plot(t,t,'-+b')
4 plt.title('linear')
5 plt.subplot(222)
6 plt.title('quadratic')
7 plt.plot(t, t**2, ':-^r')
8 plt.subplot(223)
9 plt.title('sqrt')
10 plt.plot(t,np.sqrt(t), '--sg')
11 plt.subplot(224)
12 plt.title('logarithmic')
13 plt.plot(t,np.log(t), '-oy')
14 _ = plt.tight_layout()
```



`tight_layout()` automatically tries to clean things up so that subplots don't overlap. Without this command in this example, the labels "sqrt" and "logarithmic" overlap with the x-axis tick labels in the first row.

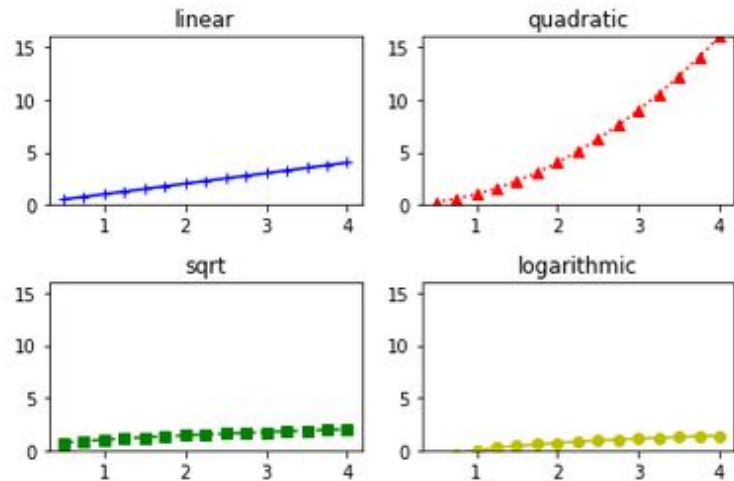
Specifying axis ranges

`plt.ylim([lower, upper])` sets y-axis limits

`plt.xlim([lower, upper])` for x-axis

For-loop goes through all of the subplots and sets their y-axis limits

```
1 t = np.arange(0.5, 4.25, 0.25)
2 ymax = np.max(t**2)
3 plt.subplot(221)
4 plt.plot(t, t, '-+b')
5 plt.title('linear')
6 plt.subplot(222)
7 plt.title('quadratic')
8 plt.plot(t, t**2, '^r')
9 plt.subplot(223)
10 plt.title('sqrt')
11 plt.plot(t, np.sqrt(t), '--sg')
12 plt.subplot(224)
13 plt.title('logarithmic')
14 plt.plot(t, np.log(t), '-oy')
15 for subplt in range(221, 225):
16     plt.subplot(subplt)
17     plt.ylim([0, ymax])
18 _ = plt.tight_layout()
```



Nonlinear axes

Scale the axes with `plt.xscale` and `plt.yscale`

Built-in scales:

Linear (`'linear'`)

Log (`'log'`)

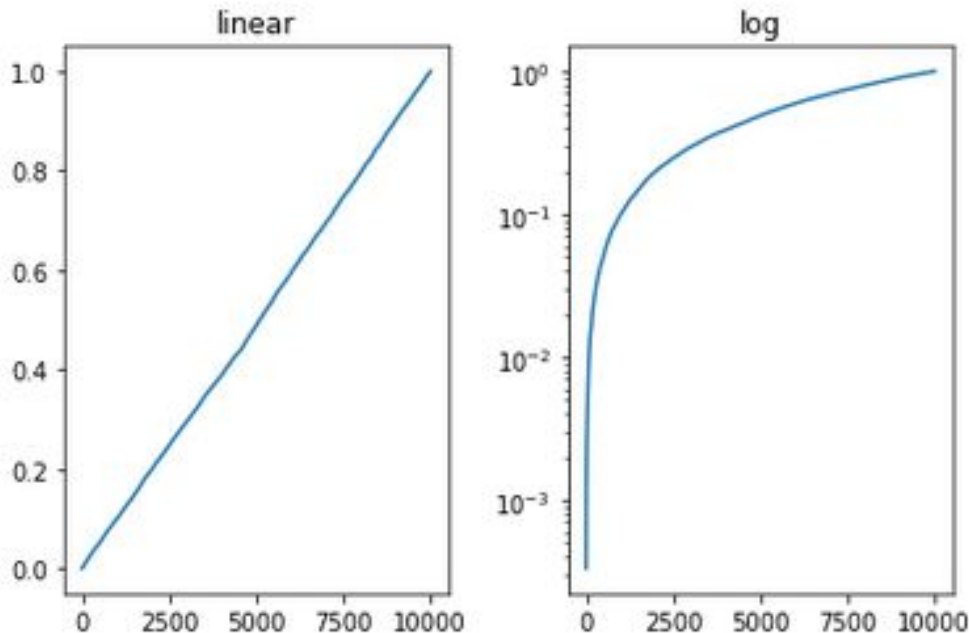
Symmetric log (`'symlog'`)

Logit (`'logit'`)

Can also specify customized scales:

https://matplotlib.org/devel/add_new_projection.html#adding-new-scales

```
1 y = np.random.uniform(0,1,10000); y.sort()
2 x = np.arange(len(y))
3 plt.subplot(121)
4 plt.plot(x,y)
5 plt.yscale('linear'); plt.title('linear')
6 plt.subplot(122)
7 plt.plot(x, y)
8 plt.yscale('log'); plt.title('log')
9 _ = plt.tight_layout()
```



Saving images

`plt.savefig(filename)` will try to automatically figure out what file type you want based on the file extension.

Can make it explicit using

```
plt.savefig('filename',  
          format='fmt')
```

Options for specifying resolution, padding, etc:

https://matplotlib.org/api/_as_gen/matplotlib.pyplot.savefig.html

```
1 random_signs = np.sign(np.random.rand(1000)-0.5)
2 plt.grid(True)
3 plt.title('Random walk of 1000 steps')
4 # cumsum() returns cumulative sums
5 _ = plt.plot(np.cumsum(random_signs))
6 plt.savefig('random_walk.svg')
```



Animations

`matplotlib.animate` package generates animations

I won't require you to make any, but they're fun to play around with (and they can be a great visualization tool)

The details are a bit tricky, so I recommend starting by looking at some of the example animations here: http://matplotlib.org/api/animation_api.html#examples