STATS 507
Data Analysis in Python

Week6-2: SciPy and Matplotlib

Dr. Xian Zhang

Adapted from slides by Professor Jeffrey Regier

Part 2: SciPy

1. SciPy as Scientific Python

2. Matplotlib for visualization

What is SciPy

SciPy: Scientific Python

Designed and used for scientific and technical computing

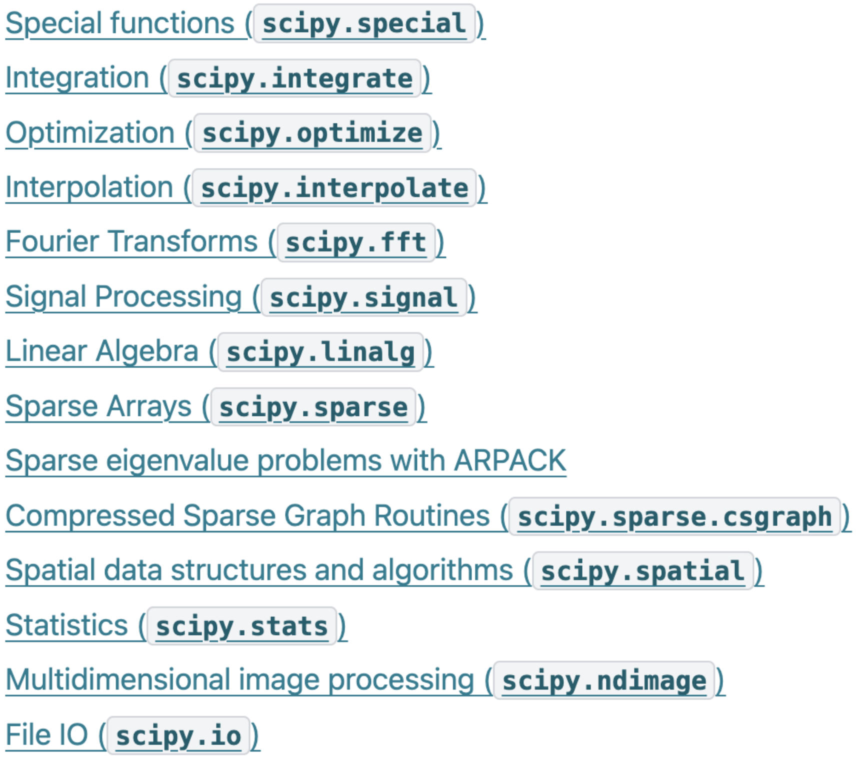
SciPy is a library (collection) of **algorithms** and **mathematical** tools built to work with NumPy arrays.

- Mathematics, science, and engineering.
- Can handle operations and data analysis that goes beyond what basic Python or even NumPy can handle.

https://docs.scipy.org/doc/scipy/tutorial/index.html#user-guide

When to use SciPy

SciPy is organized into **subpackages** covering different scientific computing domains.

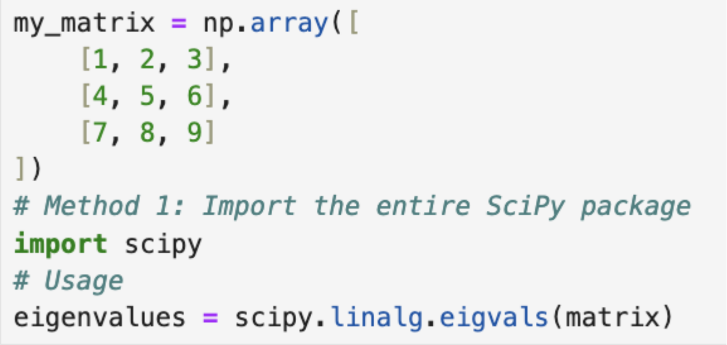


- Special functions (`scipy.special`)
- Integration (`scipy.integrate`)
- Optimization (`scipy.optimize`)
- Interpolation (`scipy.interpolate`)
- Fourier Transforms (`scipy.fft`)
- Signal Processing (`scipy.signal`)
- Linear Algebra (`scipy.linalg`)
- Sparse Arrays (`scipy.sparse`)
- Sparse eigenvalue problems with ARPACK
- Compressed Sparse Graph Routines (`scipy.sparse.csgraph`)
- Spatial data structures and algorithms (`scipy.spatial`)
- Statistics (`scipy.stats`)
- Multidimensional image processing (`scipy.ndimage`)
- File IO (`scipy.io`)

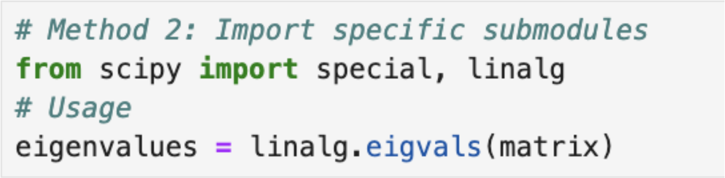
https://docs.scipy.org/doc/scipy/tutorial/index.html#user-guide

How to use SciPy: importing

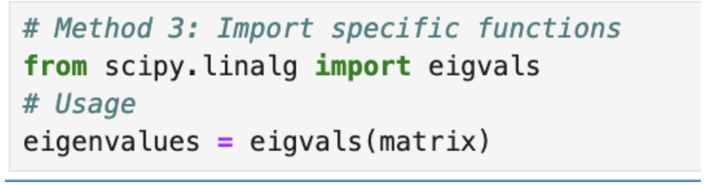
Everything in the namespaces of SciPy submodules is **public**. In general in Python, it is recommended to make use of namespaces. For example:



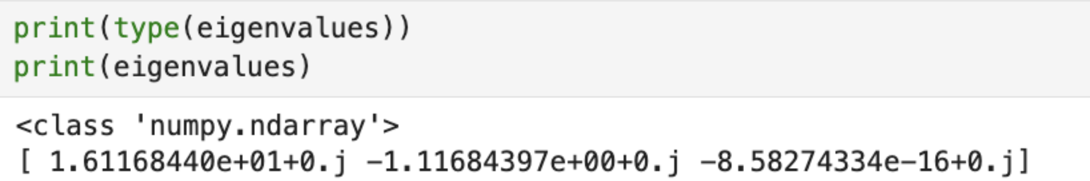
```
my_matrix = np.array([
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
])
# Method 1: Import the entire SciPy package
import scipy
# Usage
eigenvalues = scipy.linalg.eigvals(matrix)
```



```
# Method 2: Import specific submodules
from scipy import special, linalg
# Usage
eigenvalues = linalg.eigvals(matrix)
```



```
# Method 3: Import specific functions
from scipy.linalg import eigvals
# Usage
eigenvalues = eigvals(matrix)
```



```
print(type(eigenvalues))
print(eigenvalues)
```

```
<class 'numpy.ndarray'>
[ 1.61168440e+01+0.j -1.11684397e+00+0.j -8.58274334e-16+0.j]
```

https://docs.scipy.org/doc/scipy/tutorial/index.html#user-guide

SciPy Linear Algebra

There is overlap in the functionality provided by the SciPy and NumPy submodules. However:

- Slightly different from `numpy.linalg`. Always uses BLAS/LAPACK support, which are often highly optimized for specific hardware so could be **faster**.

- Unless you don’t want to add SciPy as a dependency to your NumPy program, use `scipy.linalg` instead of `numpy.linalg`

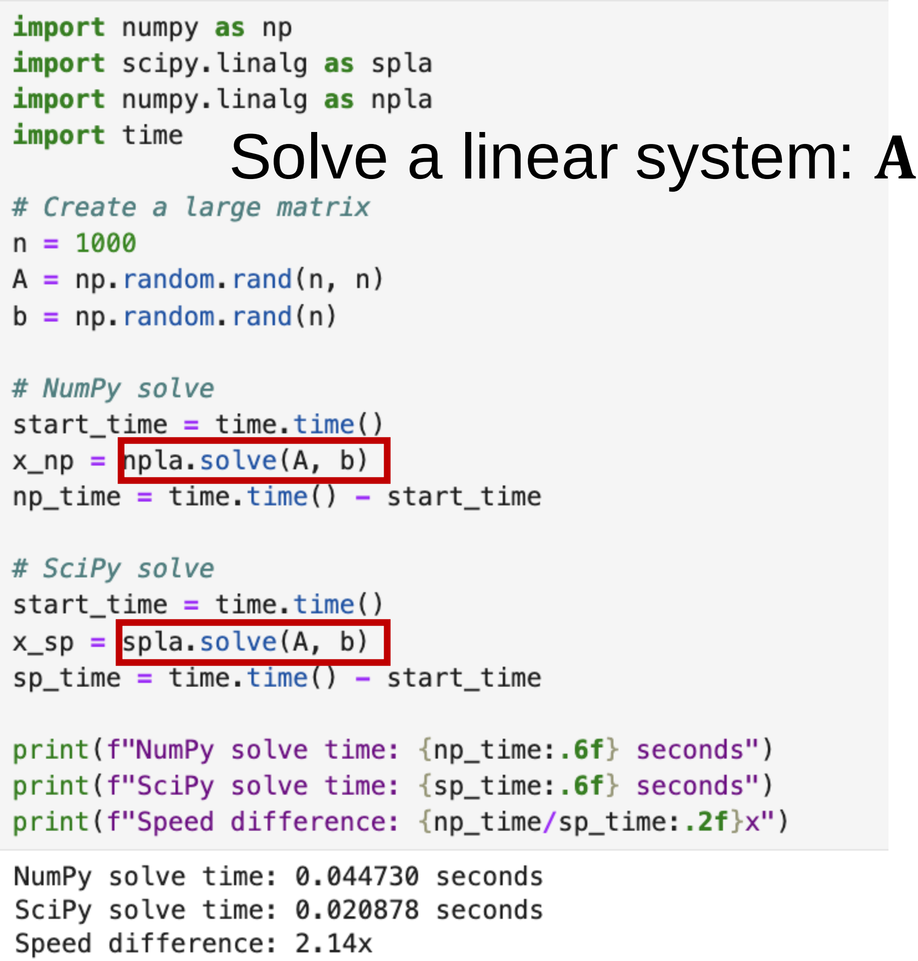
BLAS: basic linear algebra subprogram: https://www.netlib.org/blas/
LAPACK: linear algebra Package: https://www.netlib.org/lapack

https://docs.scipy.org/doc/scipy/tutorial/linalg.html

SciPy Linear Algebra

`scipy.linalg` contains all the functions in `numpy.linalg`

SciPy linear algebra could be **faster**.



```python
import numpy as np
import scipy.linalg as spla
import numpy.linalg as npla
import time

# Create a large matrix
n = 1000
A = np.random.rand(n, n)
b = np.random.rand(n)

# NumPy solve
start_time = time.time()
x_np = npla.solve(A, b)
np_time = time.time() - start_time

# SciPy solve
start_time = time.time()
x_sp = spla.solve(A, b)
sp_time = time.time() - start_time

print(f"NumPy solve time: {np_time:.6f} seconds")
print(f"SciPy solve time: {sp_time:.6f} seconds")
print(f"Speed difference: {np_time/sp_time:.2f}x")
```

```
NumPy solve time: 0.044730 seconds
SciPy solve time: 0.020878 seconds
Speed difference: 2.14x
```

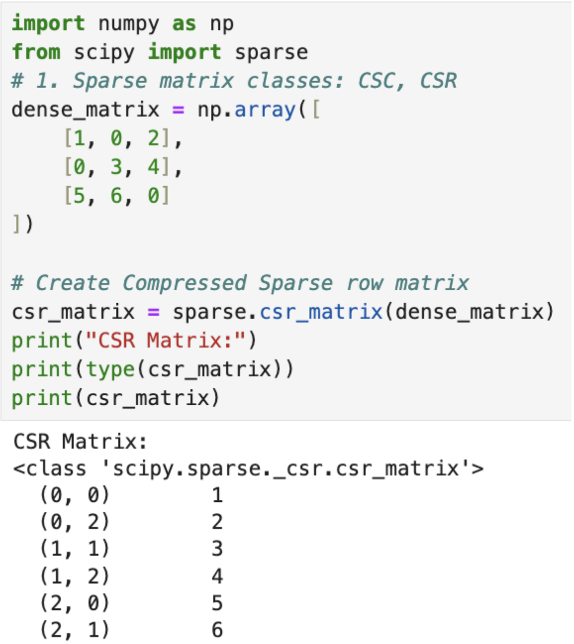
Solve a linear system: $\boldsymbol{A}x = b$

https://docs.scipy.org/doc/scipy/tutorial/linalg.html

SciPy Sparse

Sparse matrix classes: CSC, CSR, etc.

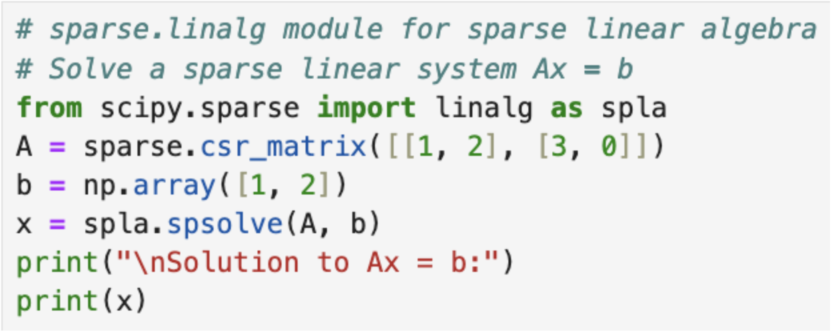
- functions to identify and build sparse matrices
- `sparse.linalg` module for sparse linear algebra
- `sparse.csgraph` for sparse graph routines



```
import numpy as np
from scipy import sparse
# 1. Sparse matrix classes: CSC, CSR
dense_matrix = np.array([
    [1, 0, 2],
    [0, 3, 4],
    [5, 6, 0]
])

# Create Compressed Sparse row matrix
csr_matrix = sparse.csr_matrix(dense_matrix)
print("CSR Matrix:")
print(type(csr_matrix))
print(csr_matrix)
```

```
CSR Matrix:
<class 'scipy.sparse._csr.csr_matrix'>
  (0, 0)	1
  (0, 2)	2
  (1, 1)	3
  (1, 2)	4
  (2, 0)	5
  (2, 1)	6
```



```
# sparse.linalg module for sparse linear algebra
# Solve a sparse linear system Ax = b
from scipy.sparse import linalg as spla
A = sparse.csr_matrix([[1, 2], [3, 0]])
b = np.array([1, 2])
x = spla.spsolve(A, b)
print("\nSolution to Ax = b:")
print(x)
```

Can be even more efficient
Since we can make use of
the structure

https://docs.scipy.org/doc/scipy/reference/sparse.html

More subroutines

More on this later!

Scipy Statistics

- Distributions
- Functions (Mean, median, mode, variance)
- Hypothesis tests
- …

Scipy Signal

- Continuous-time linear system
- Filtering
- …

Scipy IO

- Methods for loading and saving data
- Matlab files
- Matrix Market files (sparse matrices)
- Wav files

In-class practice

On SciPy Optimization

Part 3: Matplotlib

What is `matplotlib`

Matplotlib is a comprehensive **library** for creating static, animated, and interactive **visualizations** 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

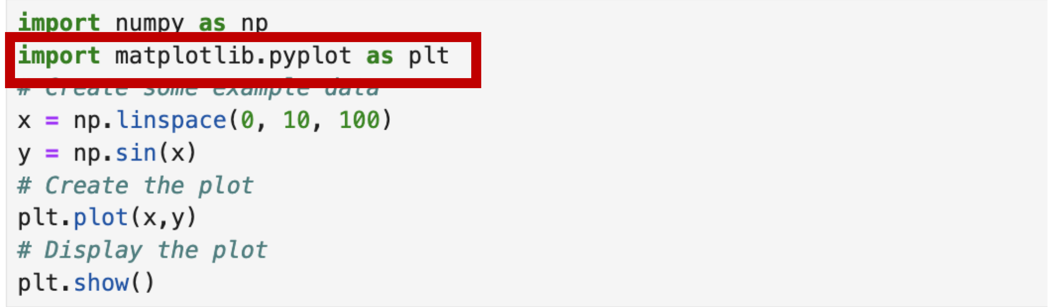
Basic plotting: `matplotlib.pyplot.plot`

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

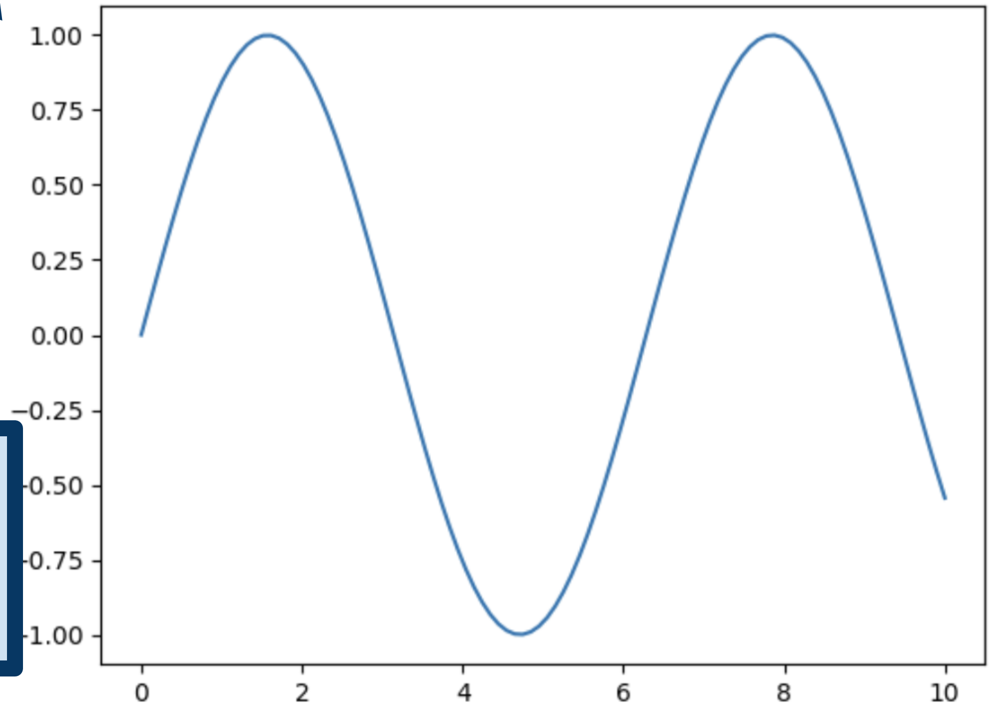
plots `y` as a function of `x`.

`matplotlib.pyplot.plot(y)`

default x-axis to `np.arrange(len(x))`



```
import numpy as np
import matplotlib.pyplot as plt
# Create some example data
x = np.linspace(0, 10, 100)
y = np.sin(x)
# Create the plot
plt.plot(x,y)
# Display the plot
plt.show()
```

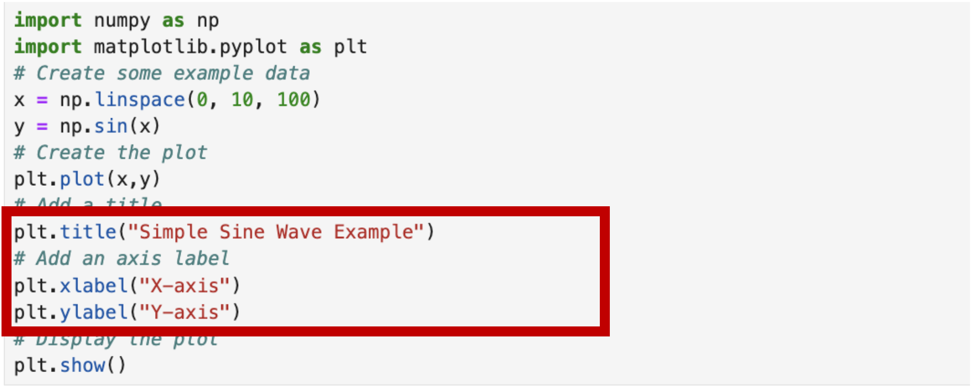


`matplotlib.pyplot` is the main plotting interface in Matplotlib, It provides an implicit, MATLAB-like way of plotting, often imported as `plt`

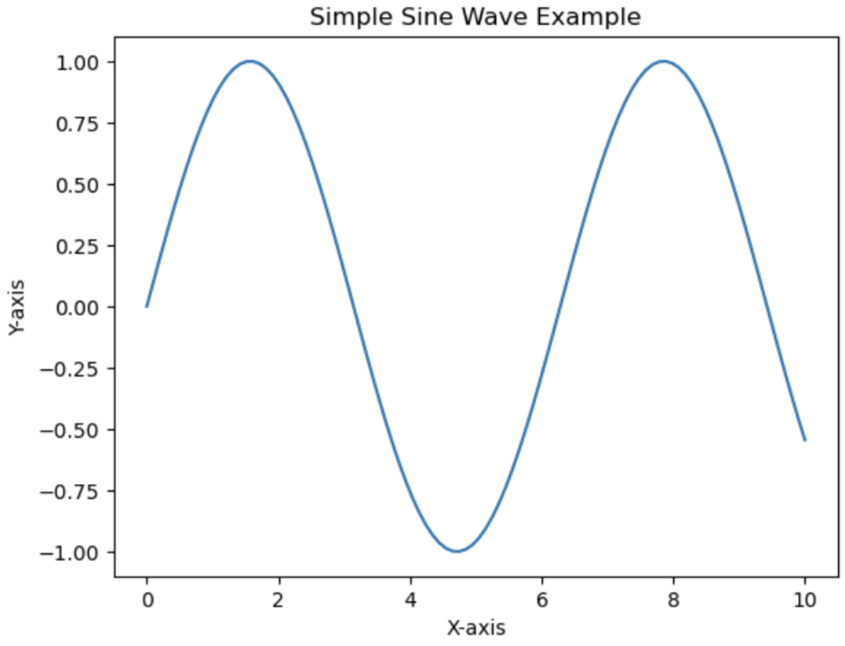
Basic plotting: `matplotlib.pyplot.plot`

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

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

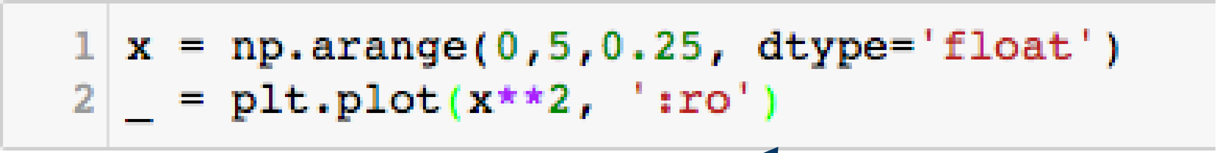


```
import numpy as np
import matplotlib.pyplot as plt
# Create some example data
x = np.linspace(0, 10, 100)
y = np.sin(x)
# Create the plot
plt.plot(x,y)
# Add a title
plt.title("Simple Sine Wave Example")
# Add an axis label
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
# Display the plot
plt.show()
```

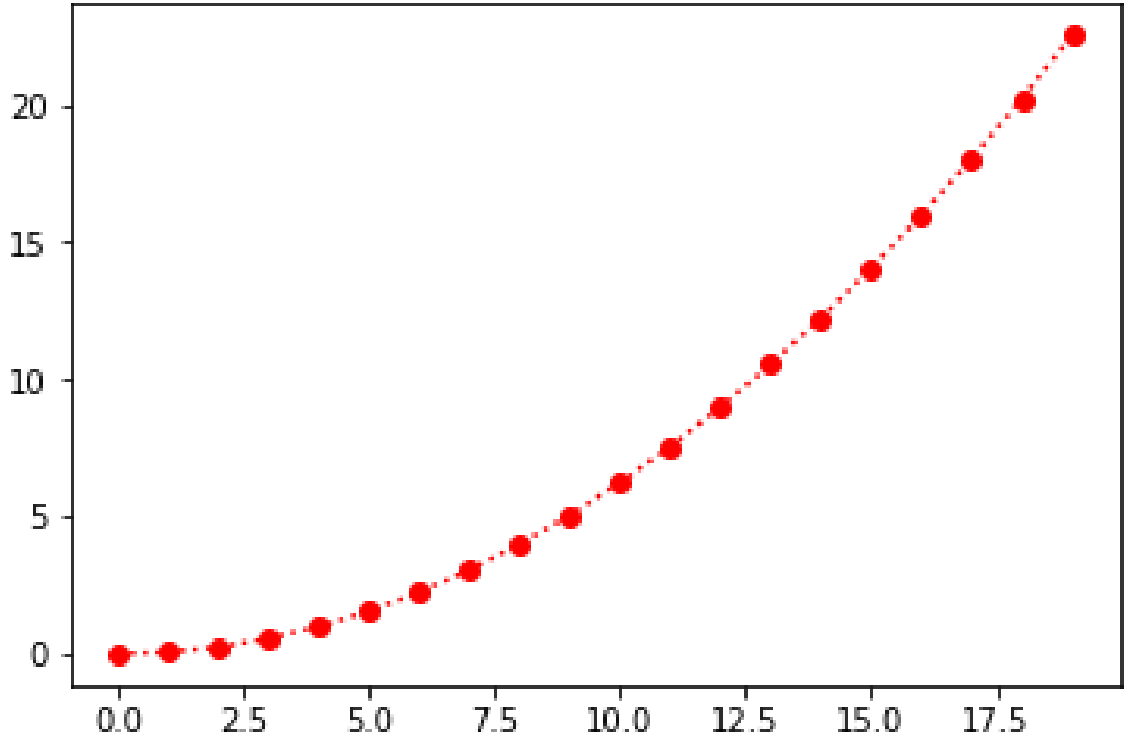


Specifying titles and axis labels couldn't be more straight-forward.

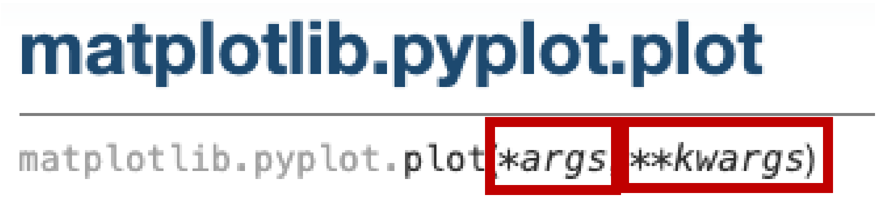
Customizing plots



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



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



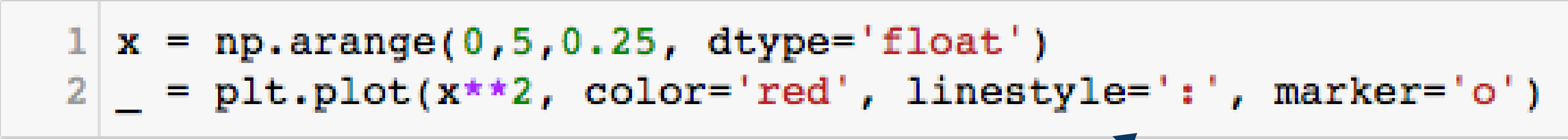
matplotlib.pyplot.plot

`matplotlib.pyplot.plot(*args, **kwargs)`

positional argument

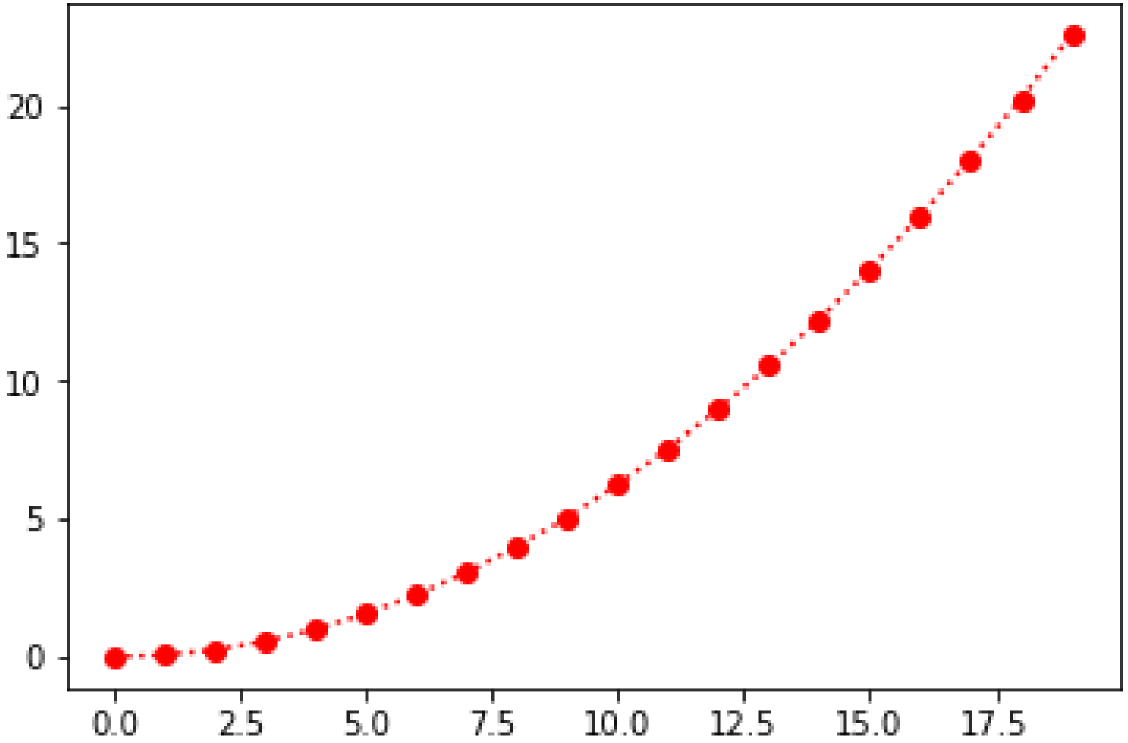
Optional keyword arguments for **customizing** the plot

Customizing plots



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

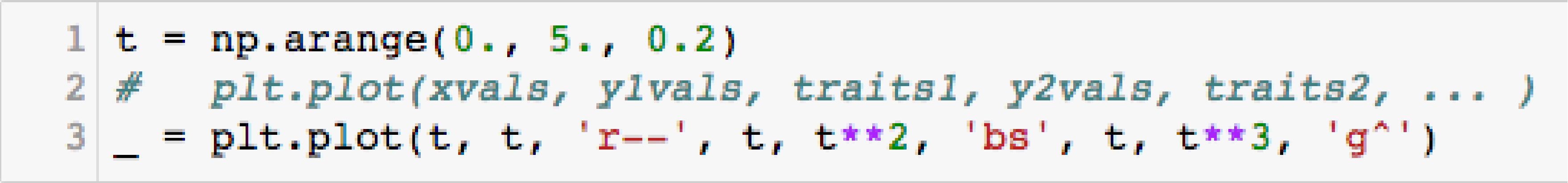
Long form of the command on the previous slide. Same plot!



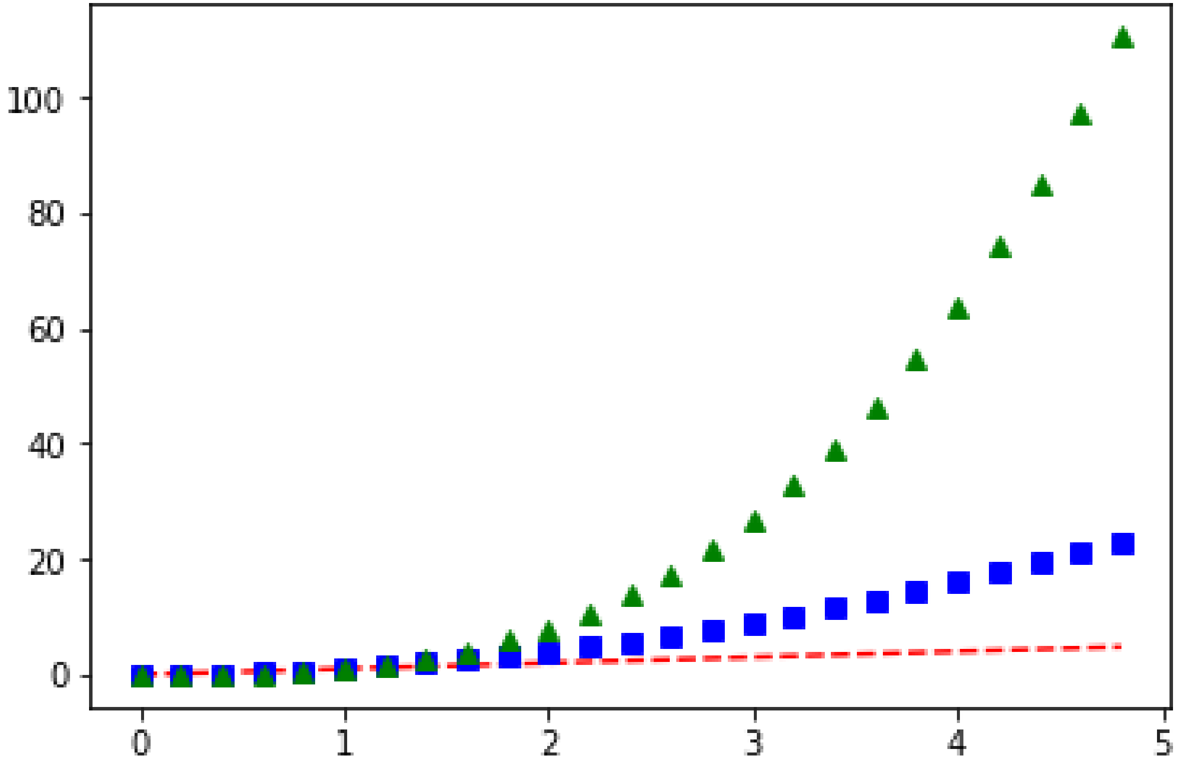
format **string** characters are accepted to control the color, line style or marker:

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

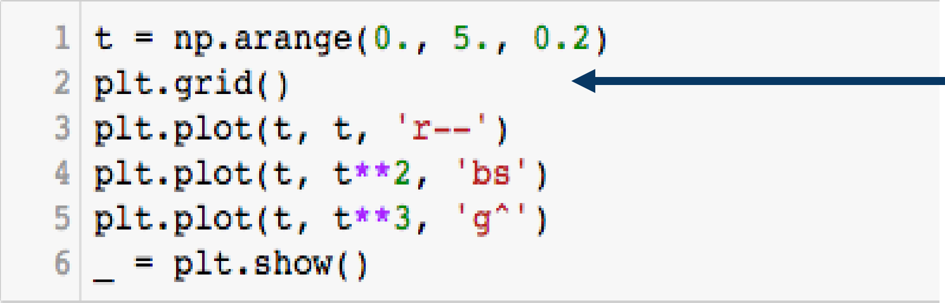


```
t = np.arange(0., 5., 0.2)
#   plt.plot(xvals, y1vals, traits1, y2vals, traits2, ... )
_ = plt.plot(t, t, 'r--', t, t**2, 'bs', t, t**3, 'g^')
```



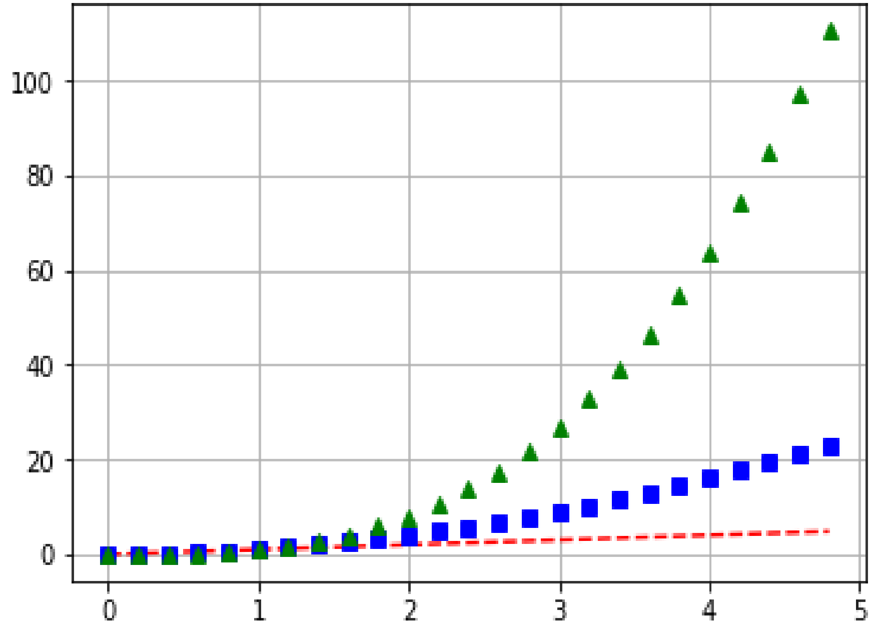
Note: more complicated specification of individual lines can be achieved by adding them to the plot one at a time.

Multiple lines in a single plot



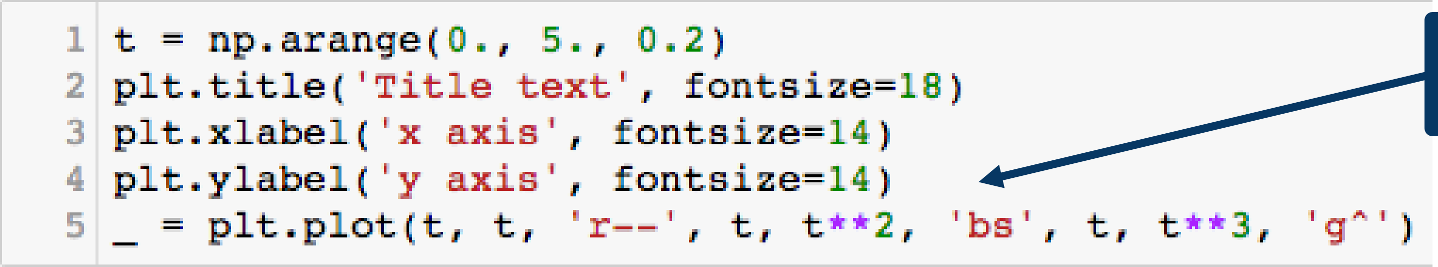
```
t = np.arange(0., 5., 0.2)
plt.grid()
plt.plot(t, t, 'r--')
plt.plot(t, t**2, 'bs')
plt.plot(t, t**3, 'g^')
_ = plt.show()
```

`plt.grid` to apply grid to the figure



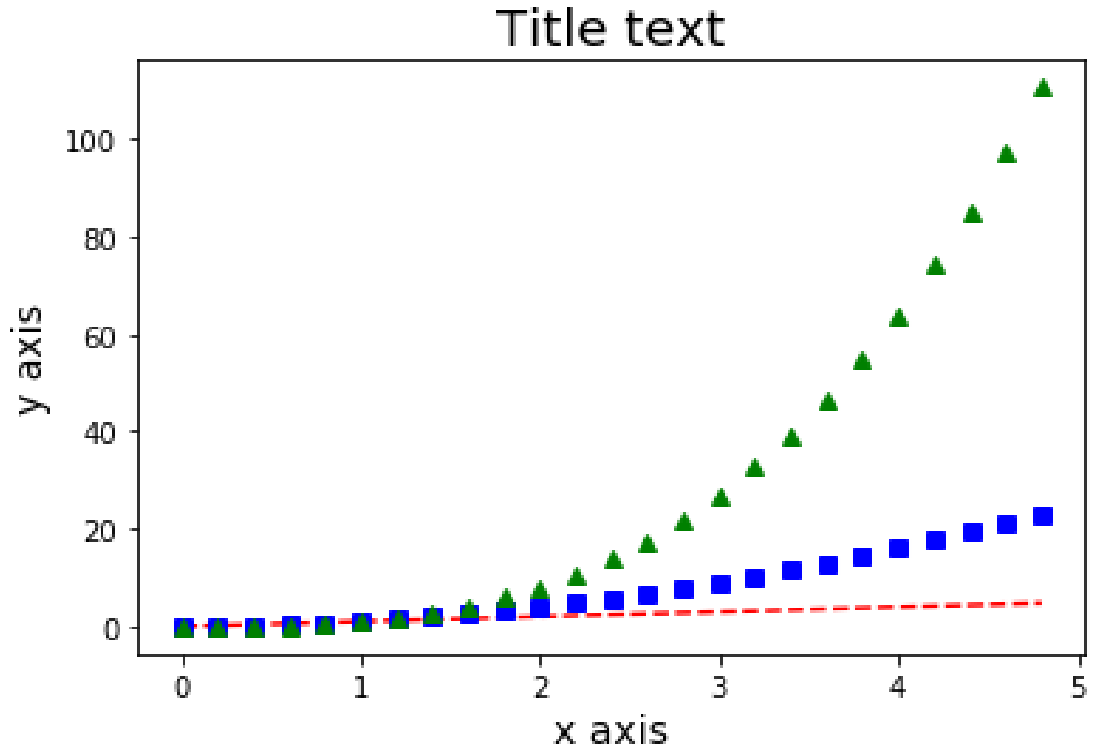
Note: same plot as previous slide, but specifying one line at a time.

Titles and axis labels

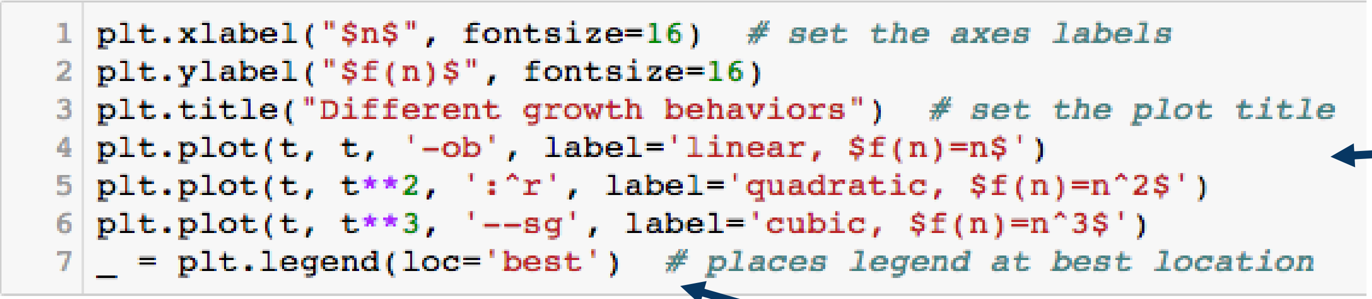


```python
t = np.arange(0., 5., 0.2)
plt.title('Title text', fontsize=18)
plt.xlabel('x axis', fontsize=14)
plt.ylabel('y axis', fontsize=14)
_ = plt.plot(t, t, 'r--', t, t**2, 'bs', t, t**3, 'g^')
```

Change font sizes

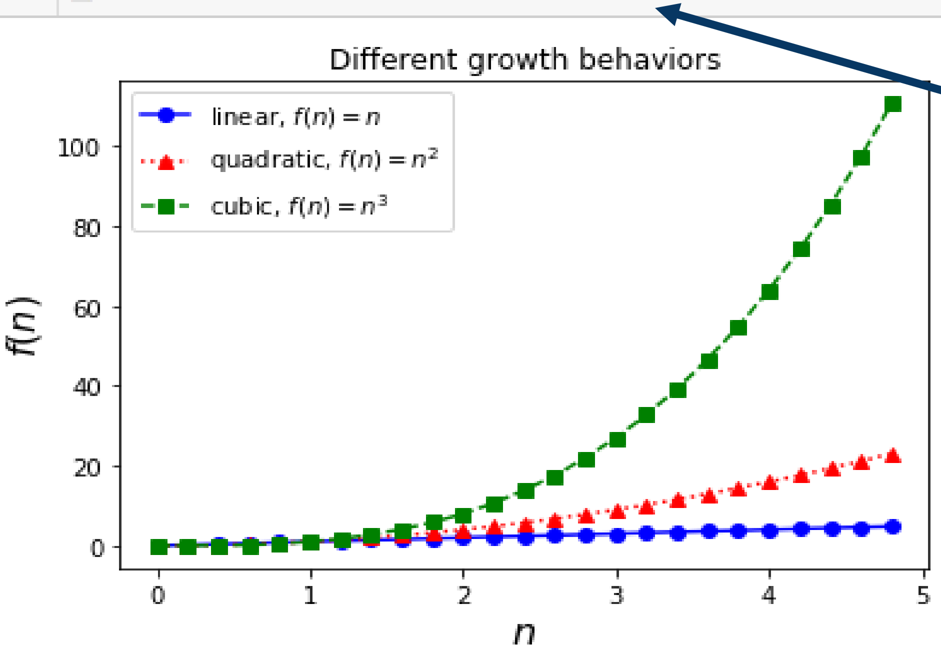


Legends



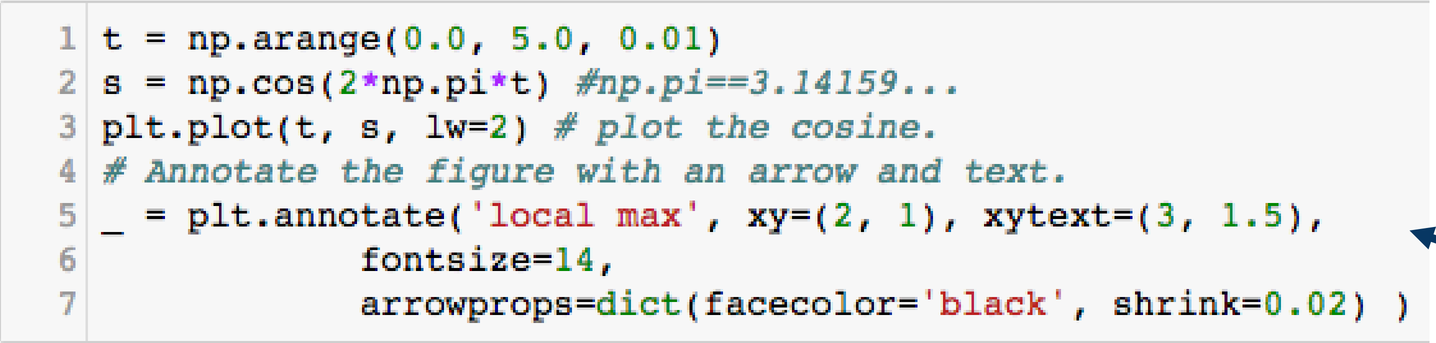
```
plt.xlabel("$n$", fontsize=16)  # set the axes labels
plt.ylabel("$f(n)$", fontsize=16)
plt.title("Different growth behaviors")  # set the plot title
plt.plot(t, t, '-ob', label='linear, $f(n)=n$')
plt.plot(t, t**2, ':^r', label='quadratic, $f(n)=n^2$')
plt.plot(t, t**3, '--sg', label='cubic, $f(n)=n^3$')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

Can use LaTeX in labels, titles, etc.

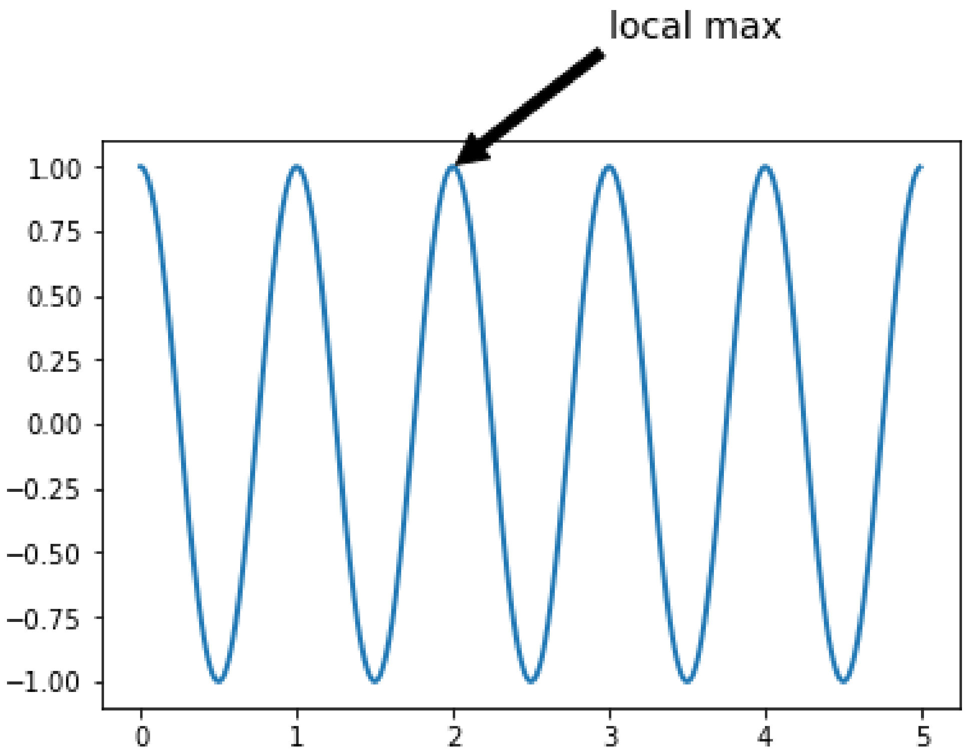


`pyplot.legend` generates legend based on label arguments passed to `pyplot.plot`. `loc='best'` tells `pyplot` to place the legend where it thinks is best.

Annotating figures



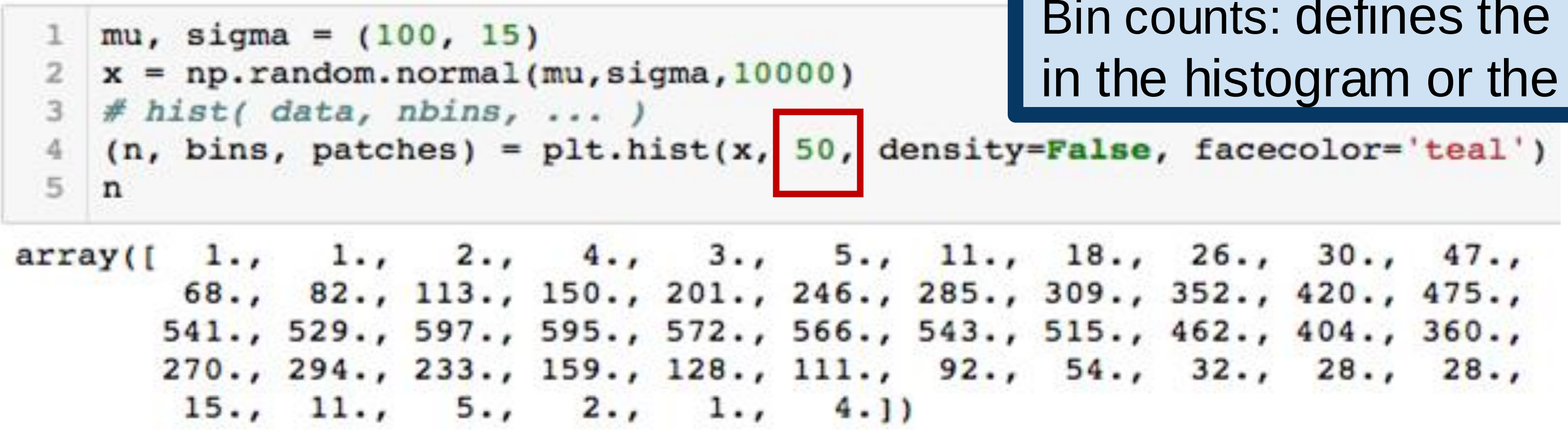
```python
t = np.arange(0.0, 5.0, 0.01)
s = np.cos(2*np.pi*t) #np.pi==3.14159...
plt.plot(t, s, lw=2) # plot the cosine.
# Annotate the figure with an arrow and text.
_ = plt.annotate('local max', xy=(2, 1), xytext=(3, 1.5),
      fontsize=14,
      arrowprops=dict(facecolor='black', shrink=0.02) )
```



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.

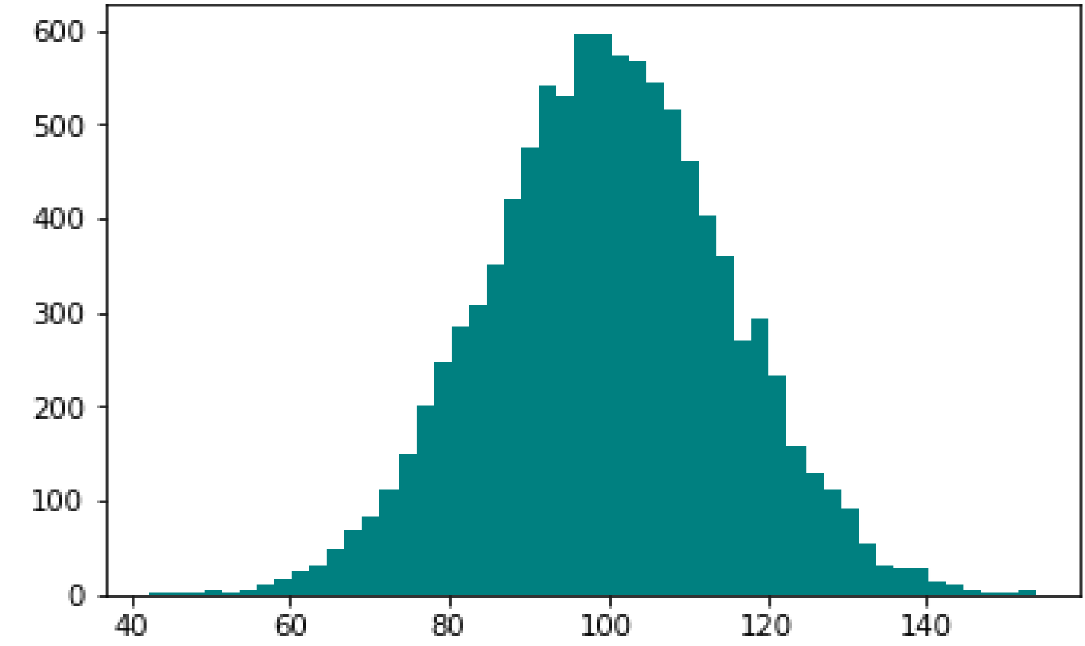
Plotting histograms: `pyplot.hist()`

Bin counts: defines the number of equal-width bins in the histogram or the bin edges.



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

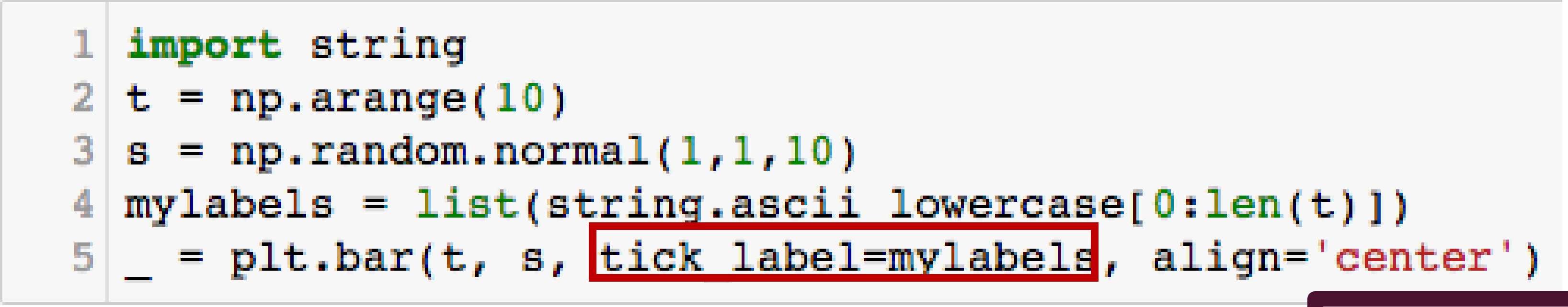
```
array([   1.,    1.,    2.,    4.,    3.,    5.,   11.,   18.,   26.,   30.,   47.,
         68.,   82.,  113.,  150.,  201.,  246.,  285.,  309.,  352.,  420.,  475.,
        541.,  529.,  597.,  595.,  572.,  566.,  543.,  515.,  462.,  404.,  360.,
        270.,  294.,  233.,  159.,  128.,  111.,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



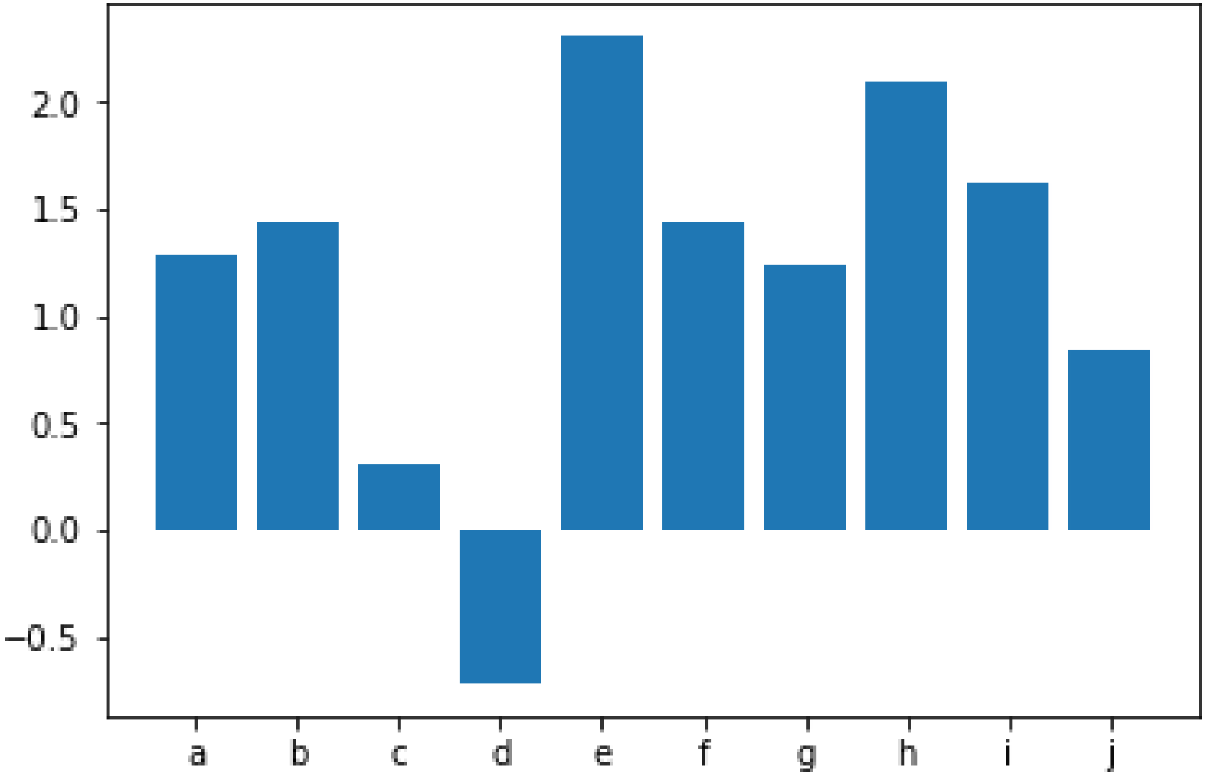
Note that if `density=True`, then these will be chosen so that the histogram “integrates” to 1. It is normalized so the total area equals 1.

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

Bar plots



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



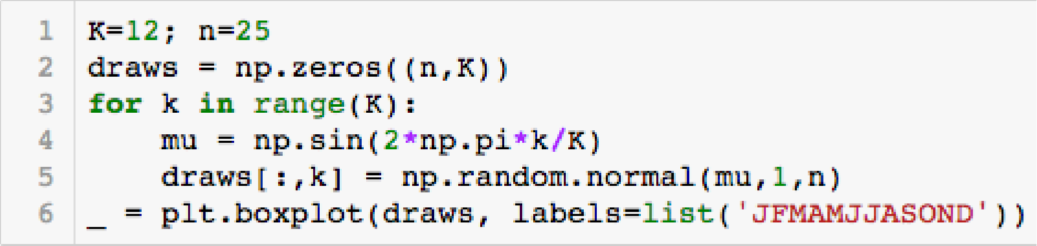
Full set of available arguments to `bar(...)` can be found at http://matplotlib.org/api/_as_gen/matplotlib.pyplot.bar.html#matplotlib.pyplot.bar

Horizontal analogue given by `barh` http://matplotlib.org/api/_as_gen/matplotlib.pyplot.barh.html#matplotlib.pyplot.barh

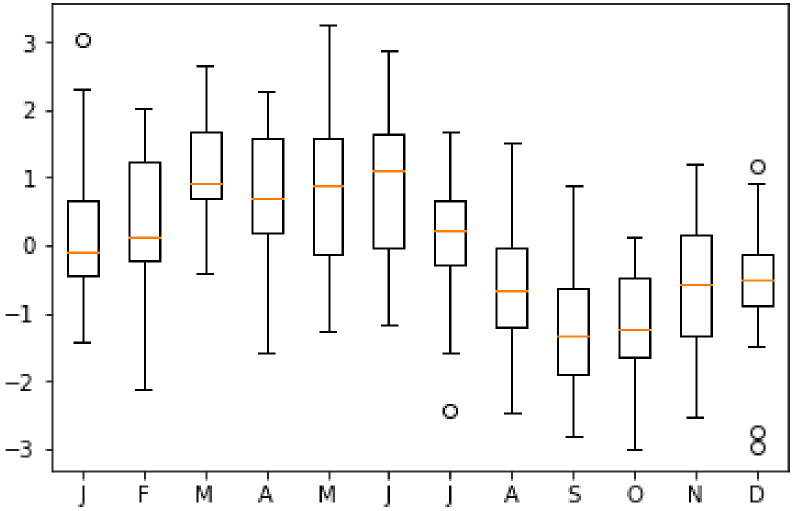
Can specify what the x-axis tick labels should be by using the `tick_label` argument to plot functions.

Box & whisker plots

Draw a box and whisker plot. The box extends from the first quartile (Q1: a statistical measure that represents the 25th percentile of a dataset.) to the third quartile (Q3) of the data, with a line at the median.



```
K=12; n=25
draws = np.zeros((n,K))
for k in range(K):
    mu = np.sin(2*np.pi*k/K)
    draws[:,k] = np.random.normal(mu,1,n)
_ = 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

Subplots

`subplot(nrows, ncols, plot_number)`

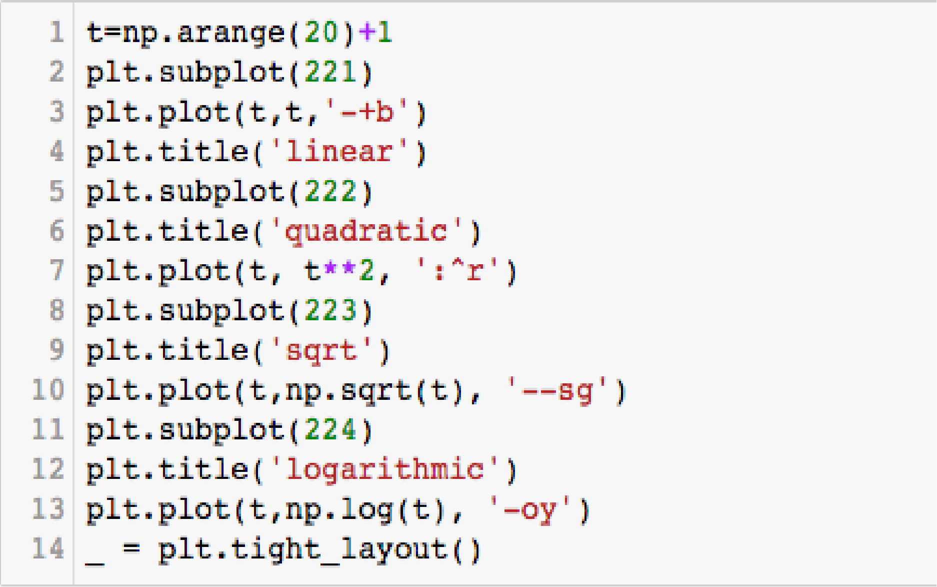
Shorthand: `subplot(XYZ)`

Makes an X-by-Y plot

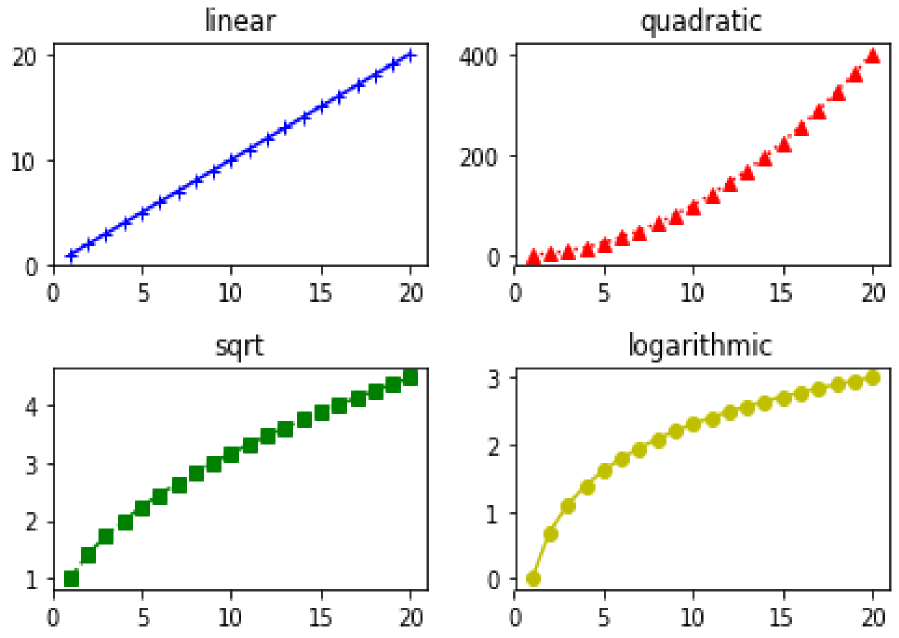
Picks out the Z-th plot

Counting in row-major order

`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.



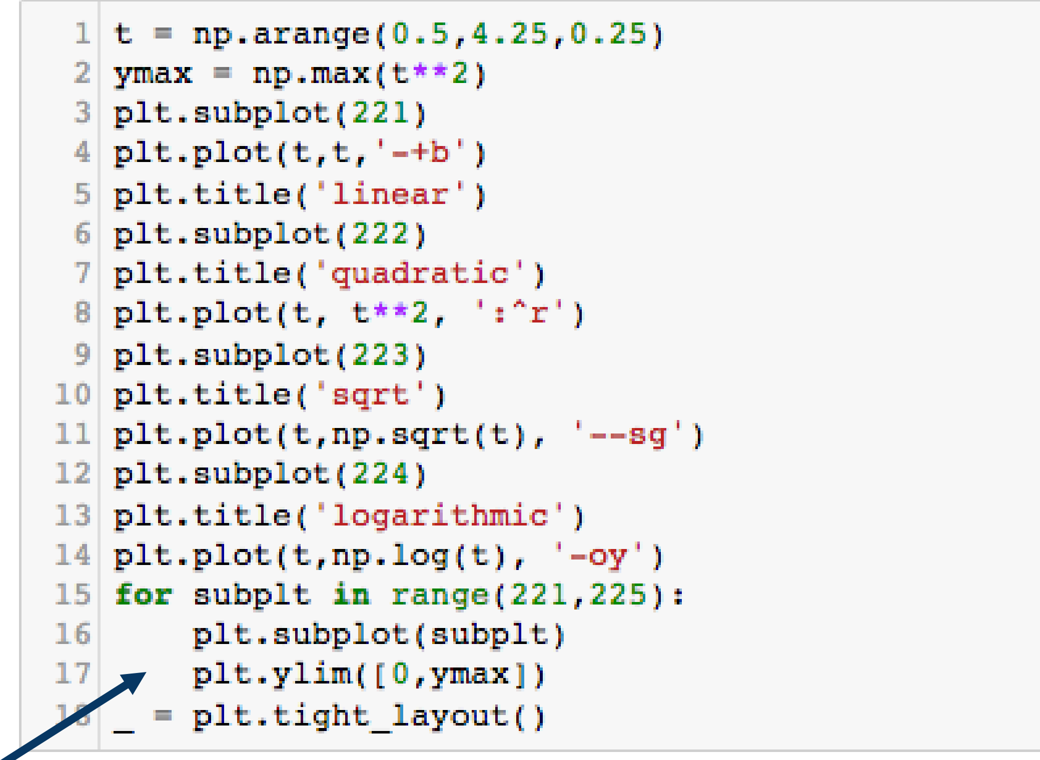
```
t=np.arange(20)+1
plt.subplot(221)
plt.plot(t,t,'-+b')
plt.title('linear')
plt.subplot(222)
plt.title('quadratic')
plt.plot(t, t**2, ':^r')
plt.subplot(223)
plt.title('sqrt')
plt.plot(t,np.sqrt(t), '--sg')
plt.subplot(224)
plt.title('logarithmic')
plt.plot(t,np.log(t), '-oy')
_ = plt.tight_layout()
```



Specifying axis ranges

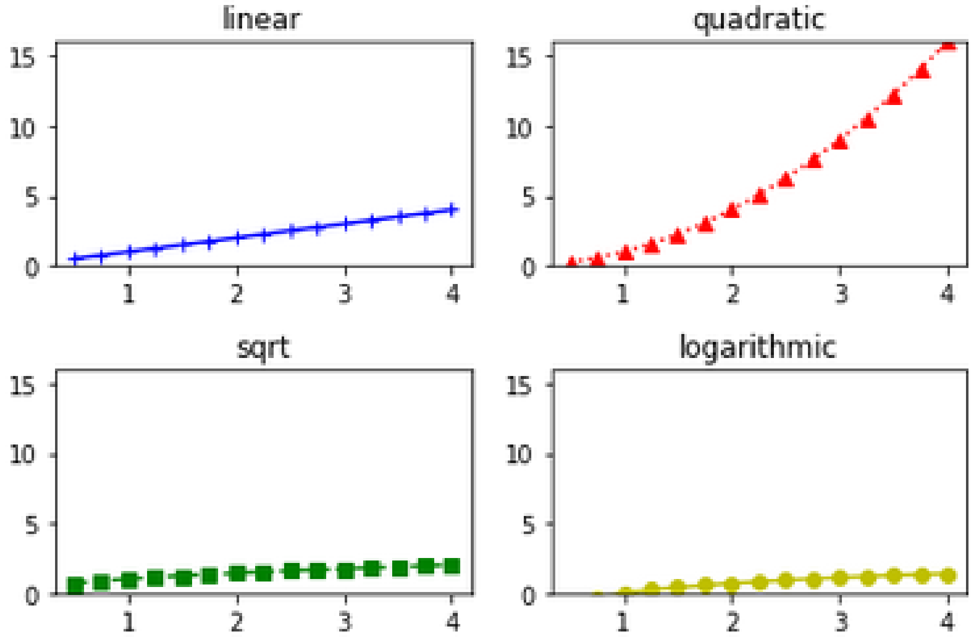
`plt.ylim([lower,upper])` sets y-axis limits

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



```
t = np.arange(0.5,4.25,0.25)
ymax = np.max(t**2)
plt.subplot(221)
plt.plot(t,t,'-+b')
plt.title('linear')
plt.subplot(222)
plt.title('quadratic')
plt.plot(t, t**2, ':^r')
plt.subplot(223)
plt.title('sqrt')
plt.plot(t,np.sqrt(t), '--sg')
plt.subplot(224)
plt.title('logarithmic')
plt.plot(t,np.log(t), '-oy')
for subplt in range(221,225):
    plt.subplot(subplt)
    plt.ylim([0,ymax])
_ = plt.tight_layout()
```

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



Nonlinear axis

Scale the axes with

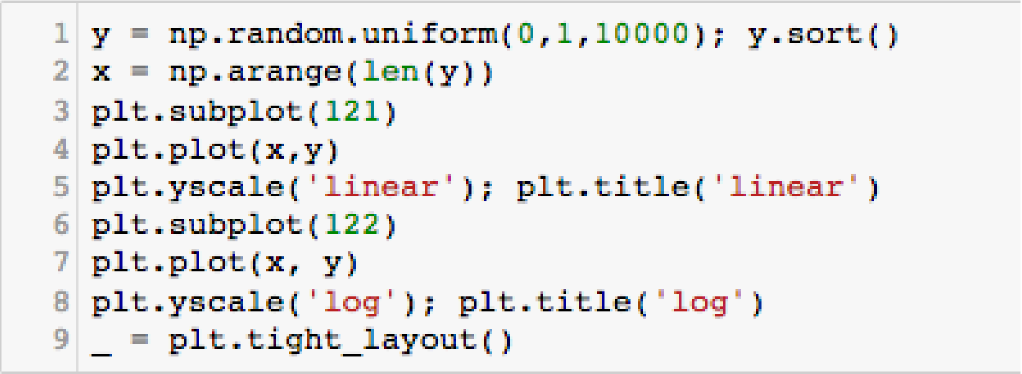
`plt.xscale` and `plt.yscale`

Built-in scales:

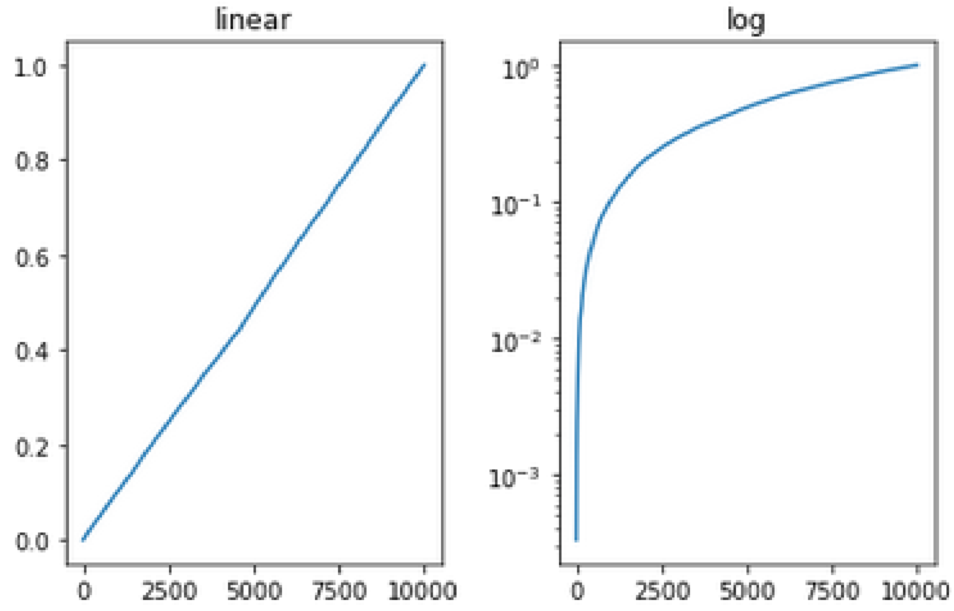
- Linear (‘linear’)
- Log (‘log’)
- Symmetric log (‘symlog’)
- Logit (‘logit’)

Can also specify customized scales:

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



```
y = np.random.uniform(0,1,10000); y.sort()
x = np.arange(len(y))
plt.subplot(121)
plt.plot(x,y)
plt.yscale('linear'); plt.title('linear')
plt.subplot(122)
plt.plot(x, y)
plt.yscale('log'); plt.title('log')
_ = plt.tight_layout()
```

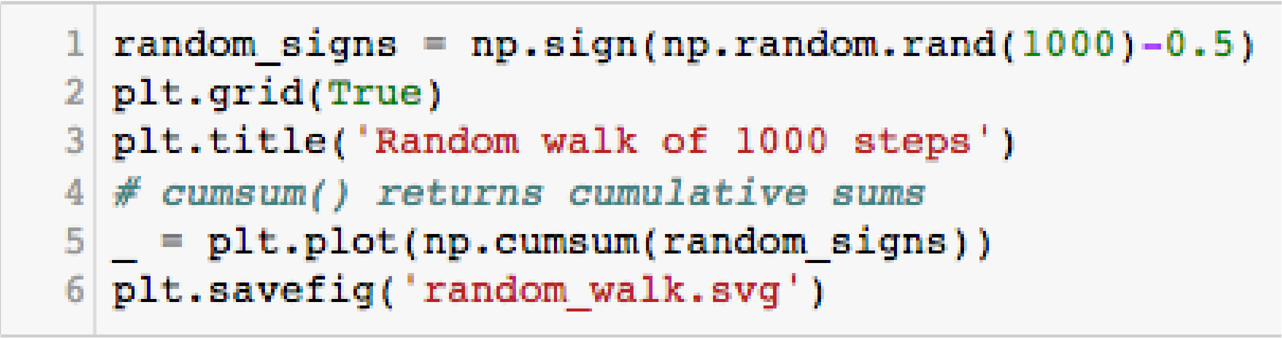


Saving images

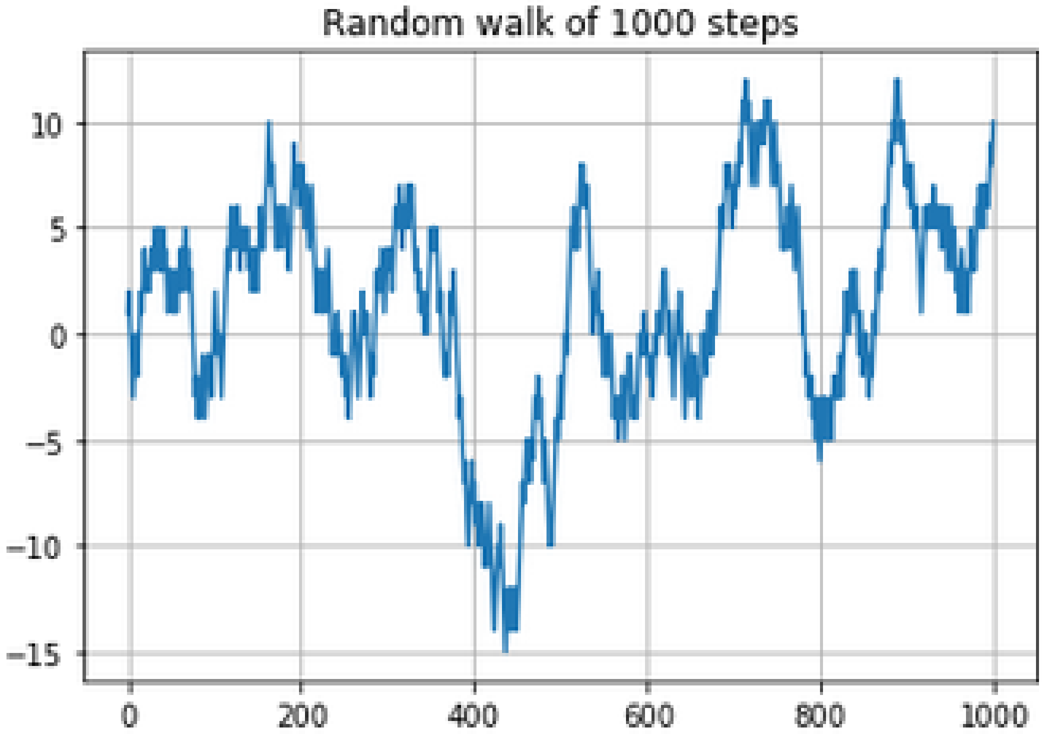
`plt.savefig(filename)` will try to **<u>automatically</u>** figure out what file type you want based on the file extension.

Or 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



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



Animations

`matplotlib.animate` package generates animations

We 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:

https://matplotlib.org/stable/api/animation_api.html

In-class practice

Other things

HW5 due Today.

HW6 is out and due after Fall Break.

Coming next:

- Midterm recap
- Midterm