STATS 507
Data Analysis in Python

Lecture 9: `numpy`, `scipy` and `matplotlib`

Some examples adapted from A. Tewari

Reminder!

If you don't already have a Flux/Fladoop username, request one promptly!

Make sure you can ssh to Fladoop: http://arc-ts.umich.edu/hadoop-user-guide/

UNIX/Linux/MacOS: you should be all set!

Windows:

install PuTTY:

https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html

and you may also want cygwin https://www.cygwin.com/

You also probably want to set up VPN to access Flux from off-campus:

http://its.umich.edu/enterprise/wifi-networks/vpn

Numerical computing in Python: `numpy`

One of a few increasingly-popular, free competitors to `MATLAB`

Numpy quickstart guide: https://docs.scipy.org/doc/numpy-dev/user/quickstart.html

For `MATLAB` fans:

https://docs.scipy.org/doc/numpy-dev/user/numpy-for-matlab-users.html

Closely related package `scipy` is for optimization

See https://docs.scipy.org/doc/

Installing packages

So far, we have only used built-in modules
- But there are many modules/packages that do not come preinstalled

Ways to install packages:
- At the conda prompt or in terminal: `conda install numpy`
  - https://conda.io/docs/user-guide/tasks/manage-pkgs.html
- Using pip (recommended): `pip install numpy`
  - https://pip.pypa.io/en/stable/
- Using UNIX/Linux package manager (not recommended)
- From source (not recommended)

Installing packages with `pip`

If you have both Python 2 and Python 3 installed, make sure you specify which one you want to install in!

```
keith@Steinhaus:~$ pip3 install beautifulsoup4
Collecting beautifulsoup4
  Downloading beautifulsoup4-4.6.0-py3-none-any.whl (86kB)
    100% |████████████████████████████████| 92kB 1.4MB/s
Installing collected packages: beautifulsoup4
Successfully installed beautifulsoup4-4.6.0
```

The above command installs the package `beautifulsoup4`. We will use that later in the semester. To install `numpy`, type the same command, but use `numpy` in place of `beautifulsoup4`.

`numpy` data types

`import … as …` lets us import a package and give it a shorter name.

```
import numpy as np

x = np.float32(3.1415)
type(x)
```

```
numpy.float32
```

```
x
```

```
3.1415
```

Note that this is not the same as a Python int.

```
x = np.int(8675309)
x
```

```
8675309
```

Five basic numerical data types:

- boolean (`bool`)
- integer (`int`)
- unsigned integer (`uint`)
- floating point (`float`)
- complex (`complex`)

Many more complicated data types are available

- e.g., each of the numerical types can vary in how many bits it uses
- https://docs.scipy.org/doc/numpy/user/basics.types.html

`numpy` data types

```
1 x = np.float64(3.1415)
2 x
```

3.1415

```
1 y = np.float32(3.1415)
2 type(y)
```

numpy.float32

As a rule, it's best never to check for equality of floats. Instead, check whether they are within some error tolerance of one another.

32-bit and 64-bit representations are distinct!

```
1 x==y
```

False

```
1 x==np.float64(y)
```

False

Data type followed by underscore uses the default number of bits. This default varies by system.

```
1 x = np.int_(8675309)
2 type(x)
```

numpy.int64

`numpy.array`: `numpy`'s version of Python array (i.e., list)

Can be created from a Python list…

```
1 np.array([1, 2, 3], dtype='uint')
```

```
array([1, 2, 3], dtype=uint64)
```

...by "shaping" an array…

```
1 np.zeros((2,3))
```

```
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
```

`np.zeros` and `np.ones` generate arrays of 0s or 1s, respectively.

...by "ranges"...

```
1 np.arange(2, 3, 0.1, dtype='float')
```

```
array([ 2. ,  2.1,  2.2,  2.3,  2.4,  2.5,  2.6,  2.7,  2.8,  2.9])
```

...or reading directly from a file

see https://docs.scipy.org/doc/numpy/user/basics.creation.html

`numpy` allows arrays of arbitrary dimension (tensors)

1-dimensional arrays:

```
x = np.arange(12) # x=[1,2,...,12]
x
```

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

2-dimensional arrays (matrices):

```
x.shape = (3,4) # now x is a 3-by-4 matrix
x # observe that shape fills the new matrix by row.
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

3-dimensional arrays (“3-tensor”):

```
x.shape = (2,3,2)
x # now x is a 2-by-3-by-2 "cube" of numbers
```

```
array([[[ 0,  1],
        [ 2,  3],
        [ 4,  5]],

       [[ 6,  7],
        [ 8,  9],
        [10, 11]]])
```

More on `numpy.arange` creation

`np.arange(x)`: array version of Python's `range(x)`, like `[0,1,2,...,x-1]`

`np.arange(x,y)`: array version of `range(x,y)`, like `[x,x+1,...,y-1]`

`np.arange(x,y,z)`: array of elements `[x,y)` in `z`-size increments.

Related useful functions, that give better/clearer control of start/endpoints and allow for multidimensional arrays:

- https://docs.scipy.org/doc/numpy/reference/generated/numpy.linspace.html
- https://docs.scipy.org/doc/numpy/reference/generated/numpy.ogrid.html
- https://docs.scipy.org/doc/numpy/reference/generated/numpy.mgrid.html

More on `numpy.arange` creation

`np.arange(x)`: array version of Python's `range(x)`, like `[0,1,2,...,x-1]`

`np.arange(x,y)`: array version of `range(x,y)`, like `[x,x+1,...,y-1]`

`np.arange(x,y,z)`: array of elements `[x,y)` in `z`-size increments.

```
1 np.arange(10)
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
1 np.arange(5,10)
```

```
array([5, 6, 7, 8, 9])
```

```
1 np.arange(0,1,0.1)
```

```
array([ 0. ,  0.1,  0.2,  0.3,  0.4,  0.5,  0.6,  0.7,  0.8,  0.9])
```

`numpy` array indexing is highly expressive

```
x = np.arange(10)
x[2:5]
```

```
array([2, 3, 4])
```

```
x[:-7]
```

```
array([0, 1, 2])
```

```
x[1:7:2]
```

```
array([1, 3, 5])
```

```
x[::2]
```

```
array([0, 2, 4, 6, 8])
```

Slices, strides, indexing from the end, etc.
Just like with Python lists.

Not very relevant to us right now…

...but this will come up again in a few weeks when we cover TensorFlow

More array indexing

Numpy allows MATLAB/R-like indexing by Booleans

```
x = np.arange(10)
x[x>7]
```

```
array([8, 9])
```

```
x[(x>7) or (x<2)]
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-373-6b519499a034> in <module>()
----> 1 x[(x>7) or (x<2)]

ValueError: The truth value of an array with more than one element is ambiguous. Use a.any() or a.all()
```

Believe it or not, this error is by design! The designers of `numpy` were concerned about ambiguities in Boolean vector operations, so they split the two operations into two separate methods, `x.any()` and `x.all()`

More array indexing

```
1 x = np.reshape(np.arange(1,13), (3,4))
2 x
```

```
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

```
1 x[1]
```

```
array([5, 6, 7, 8])
```

```
1 x[:,(1,3)]
```

```
array([[ 2,  4],
       [ 6,  8],
       [10, 12]])
```

```
1 x[(0,2),(1,3)]
```

```
array([ 2, 12])
```

If we specify fewer than the number of indices, `numpy` assumes we mean `:` in the remaining indices.

Warning: if you're used to MATLAB or R, this behavior will seem weird to you.

From the documentation: When the index consists of as many integer arrays as the array being indexed has dimensions, the indexing is straight forward, but different from slicing. Advanced indexes always are broadcast and iterated as *one*. https://docs.scipy.org/doc/numpy/reference/arrays.indexing.html#integer-array-indexing

Boolean operations: `np.any()`, `np.all()`

```
x = np.arange(10)
np.all(x>7)
```

```
False
```

Just like the `any` and `all` functions in Python proper.

```
np.any(x>7)
```

```
True
```

`axis` argument picks which axis along which to perform the Boolean operation. If left unspecified, it treats the array as a single vector.

```
np.any([x>7,x<2])
```

```
True
```

```
np.any([x>7,x<2], axis=1)
```

```
array([ True,  True], dtype=bool)
```

Setting `axis` to be the first (i.e., 0-th) axis yields the entrywise behavior we wanted.

```
np.any([x>7,x<2], axis=0)
```

```
array([ True,  True, False, False, False, False, False, False,  True,  True], dtype=bool)
```

Boolean operations: `np.logical_and()`

`numpy` also has built-in Boolean vector operations, which are simpler/clearer at the cost of the expressiveness of `np.any()`, `np.all()`.

```
x = np.arange(10)
x[np.logical_and(x>3,x<7)]
```

```
array([4, 5, 6])
```

```
np.logical_or(x<3,x>7)
```

```
array([ True,  True,  True, False, False, False, False, False,  True,  True], dtype=bool)
```

```
x[np.logical_xor(x>3,x<7)]
```

```
array([0, 1, 2, 3, 7, 8, 9])
```

```
x[np.logical_not(x>3)]
```

```
array([0, 1, 2, 3])
```

This is an example of a numpy “universal function” (ufunc), which we’ll discuss more in a few slides.

Random numbers in `numpy`

`np.random` contains methods for generating random numbers

```
np.random.random((2,3))
```

```
array([[ 0.61420793,  0.46363275,  0.22880783],
       [ 0.24268979,  0.13462754,  0.6026283 ]])
```

```
np.random.normal(0,1,20)
```

```
array([ 1.31323138,  0.76807767,  1.92180038, -0.34121468,  0.72572401,
        1.0273551 , -0.78435871,  0.42732636,  1.05947171,  0.23042635,
        0.3951938 ,  0.3595342 ,  0.14710555,  0.42279814,  0.84381846,
        1.06495165, -1.51074354, -0.16419861,  2.89275956, -1.18501386])
```

```
np.random.uniform(0,1,(2,4))
```

```
array([[ 0.08399452,  0.03934797,  0.3603464 ,  0.66361677],
       [ 0.33499095,  0.29427732,  0.14963153,  0.87892145]])
```

Lots more distributions:

https://docs.scipy.org/doc/numpy/reference/routines.random.html#distributions

`np.random.choice()`: random samples from data

`np.random.choice(x,[size,replace,p])`

Generates a sample of `size` elements from the array `x`, drawn with (`replace=True`) or without (`replace=False`) replacement, with element probabilities given by vector `p`.

```
x = np.arange(1,11)
for i in range(5):
    print np.random.choice(x,5,False,x/float(sum(x)))
```

```
[ 1  5 10  7  6]
[8 5 9 2 6]
[ 9  6  3  8 10]
[ 7  9 10  5  6]
[8 5 6 9 1]
```

`shuffle()` **vs** `permutation()`

`np.random.shuffle(x)`

randomly permutes entries of `x` in place so `x` itself is changed by this operation!

`np.random.permutation(x)`

returns a random permutation of `x` and `x` remains unchanged.

```
x = np.arange(10)
print x
```

```
[0 1 2 3 4 5 6 7 8 9]
```

```
np.random.shuffle(x)
print x # x is different, now.
```

```
[1 5 0 3 2 7 6 8 9 4]
```

```
print np.random.permutation(x)
```

```
[5 2 8 7 0 3 9 6 1 4]
```

```
print x # x is unchanged by permutation()
```

```
[1 5 0 3 2 7 6 8 9 4]
```

Statistics in `numpy`

`numpy` implements all the standard statistics functions you've come to expect

```
x = np.random.normal(0,1,100)
np.mean(x), np.median(x), np.std(x)
```

```
(-0.0627248756433586, -0.0526187335044152, 1.0556291754262765)
```

```
np.min(x), np.max(x), np.ptp(x) # ptp gets max-min
```

```
(-3.1029568746428113, 1.9628924810049164, 5.0658493556477282)
```

```
np.std(x), np.var(x)
```

```
(1.0556291754262765, 1.1143529560111607)
```

Statistics in `numpy` (cont'd)

Numpy deals with NaNs more gracefully than MATLAB/R:

```
x[5] = np.nan
np.mean(x)
```

```
nan
```

```
np.nanmin(x), np.nanmax(x), np.nanstd(x), np.nanvar(x)
```

```
(-3.102956874642811З,
 1.9628924810049164,
 1.0439479158102707,
 1.0898272509246081)
```

For more statistical functions, see:
https://docs.scipy.org/doc/numpy-1.8.1/reference/routines.statistics.html

Probability and statistics in `scipy`

Scipy is a distinct Python package, part of the `numpy` ecosystem.

(Almost) all the distributions you could possibly ever want:

https://docs.scipy.org/doc/scipy/reference/stats.html#continuous-distributions
https://docs.scipy.org/doc/scipy/reference/stats.html#multivariate-distributions
https://docs.scipy.org/doc/scipy/reference/stats.html#discrete-distributions

More statistical functions (moments, kurtosis, statistical tests):

https://docs.scipy.org/doc/scipy/reference/stats.html#statistical-functions

```
import scipy.stats
x = np.random.normal(0,1,20)
scipy.stats.kstest(x, 'norm')
```

```
KstestResult(statistic=0.23182037538316391, pvalue=0.19897055187485568)
```

Second argument is the name of a distribution in `scipy.stats`

Kolmogorov-Smirnov test

Matrix-vector operations in `numpy`

```
A = np.reshape(np.arange(1,13), (3,4))
x = np.ones(4)
A*x
```

```
array([[  1.,   2.,   3.,   4.],
       [  5.,   6.,   7.,   8.],
       [  9.,  10.,  11.,  12.]])
```

Trying to multiply two arrays, and you get broadcast behavior, *not a* matrix-vector product.

```
y = np.ones(3)
A*y
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-83-86c92ad89b88> in <module>()
      1 y = np.ones(3)
----> 2 A*y

ValueError: operands could not be broadcast together with shapes (3,4) (3,)
```

Broadcast multiplication still requires that dimensions agree and all that.

```
np.reshape(y, (3,1))*A
```

```
array([[  1.,   2.,   3.,   4.],
       [  5.,   6.,   7.,   8.],
       [  9.,  10.,  11.,  12.]])
```

Matrix-vector operations in `numpy`

```
1 A = np.matrix(np.reshape(np.arange(1,13),(3,4)))
2 A
```

```
matrix([[ 1,  2,  3,  4],
        [ 5,  6,  7,  8],
        [ 9, 10, 11, 12]])
```

Create a numpy matrix from a numpy array. Can also create matrices from strings with MATLAB-like syntax. See documentation.

```
1 x = np.ones((4,1))
2 A*x
```

```
matrix([[10.],
        [26.],
        [42.]])
```

Now matrix-vector and vector-matrix multiplication work as we want.

```
1 y = np.ones((1,3))
2 y*A
```

```
matrix([[15., 18., 21., 24.]])
```

Numpy matrices support a whole bunch of useful methods. See documentation: https://docs.scipy.org/doc/numpy/reference/generated/numpy.matrix.html

`numpy`/`scipy` universal functions (`ufuncs`)

From the documentation:

> A universal function (or ufunc for short) is a function that operates on ndarrays in an element-by-element fashion, supporting array broadcasting, type casting, and several other standard features. That is, a ufunc is a “vectorized” wrapper for a function that takes a fixed number of scalar inputs and produces a fixed number of scalar outputs.
> https://docs.scipy.org/doc/numpy/reference/ufuncs.html

So ufuncs are vectorized operations, just like in R and MATLAB

`ufuncs` in action

List comprehensions are great, but they're not well-suited to numerical computing

```
x = range(10)
x**2
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-466-84f8296342ab> in <module>()
      1 x = range(10)
----> 2 x**2

TypeError: unsupported operand type(s) for ** or pow(): 'list' and 'int'
```

```
[x**2 for x in np.arange(10)]
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
x = np.arange(10)
x**2
```

```
array([ 0,  1,  4,  9, 16, 25, 36, 49, 64, 81])
```

Unlike Python lists, `numpy` arrays support vectorized operations.

Sorting with `numpy/scipy`

```
charray = np.array([c for c in 'Michigan']).reshape((2, 4))
print charray
```

```
[['M' 'i' 'c' 'h']
 ['i' 'g' 'a' 'n']]
```

ASCII rears its head-- capital letters are "earlier" than all lower-case by default.

```
np.sort(charray)
```

```
array([['M', 'c', 'h', 'i'],
       ['a', 'g', 'i', 'n']],
      dtype='|S1')
```

Sorting is along the "last" axis by default. Note contrast with `np.any()`. To treat the array as a single vector, `axis` must be set to `None`.

```
np.sort(charray, axis=1)
```

```
array([['M', 'c', 'h', 'i'],
       ['a', 'g', 'i', 'n']],
      dtype='|S1')
```

```
np.sort(charray, axis=0)
```

```
array([['M', 'g', 'a', 'h'],
       ['i', 'i', 'c', 'n']],
      dtype='|S1')
```

```
np.sort(charray, axis=None)
```

```
array(['M', 'a', 'c', 'g', 'h', 'i', 'i', 'n'],
      dtype='|S1')
```

Original array is unchanged by use of `np.sort()`, like Python's built-in `sorted()`

```
print charray
```

```
[['M' 'i' 'c' 'h']
 ['i' 'g' 'a' 'n']]
```

A cautionary note

`numpy`/`scipy` have several similarly-named functions with different behaviors!

Example: `np.amax`, `np.ndarray.max`, `np.maximum`

The best way to avoid these confusions is to

1) Read the documentation carefully
2) Test your code!

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`

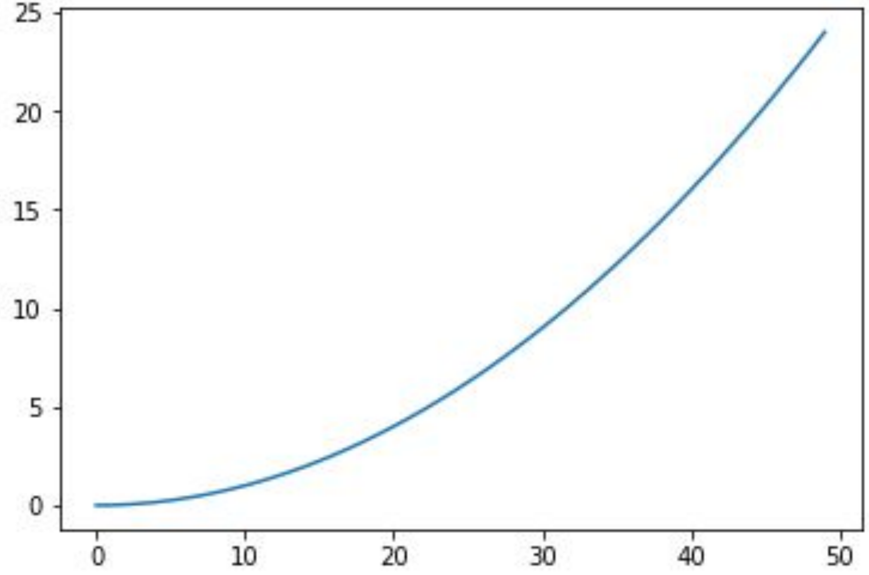
`matplotlib.pyplot.plot(x,y)`

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

`matplotlib.pyplot(t)`

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

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

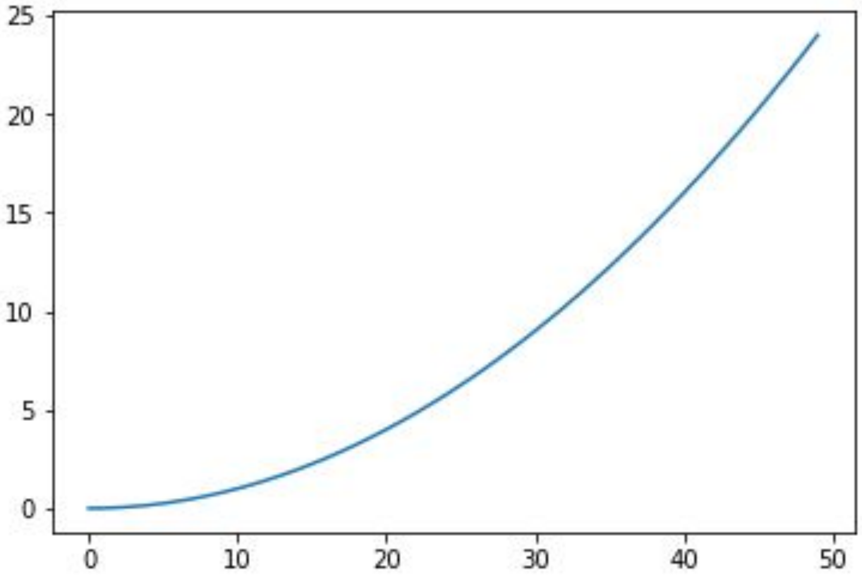


Basic plotting: `matplotlib.pyplot.plot`

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

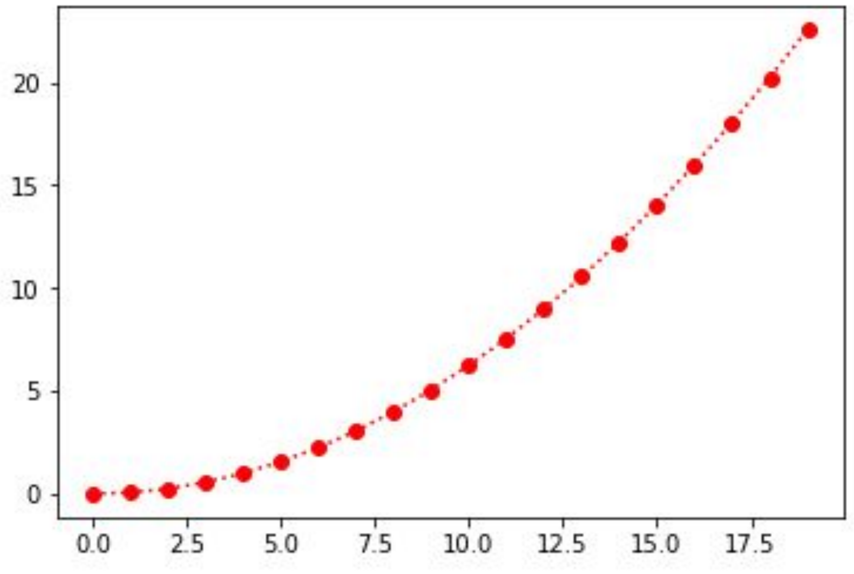
Reminder: Python ‘_’ is a placeholder, similar to MATLAB ‘~’. Tells Python to treat this like variable assignment, but don’t store result anywhere.

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



Customizing plots

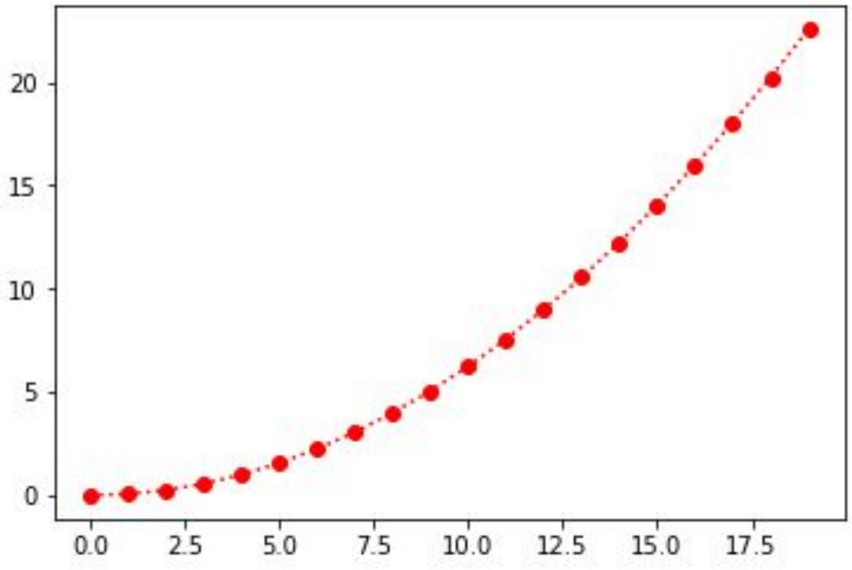
```
x = np.arange(0,5,0.25, dtype='float')
_ = plt.plot(x**2, ':ro')
```



Second argument to `pyplot.plot` specifies line type, line color, and marker type. Specify broader array of colors, line weights, markers, etc., using long-hand arguments.

Customizing plots

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

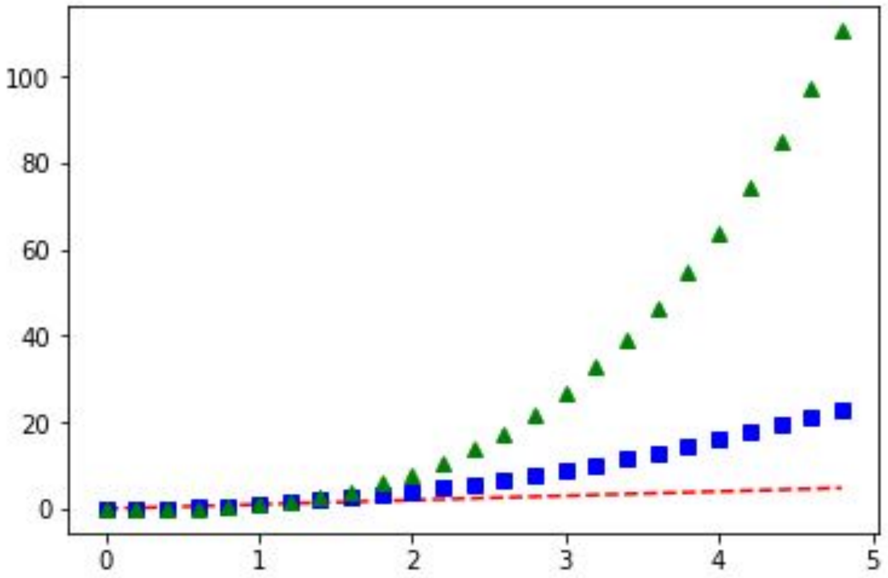


Long form of the command on the previous slide. Same plot!

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^')
```

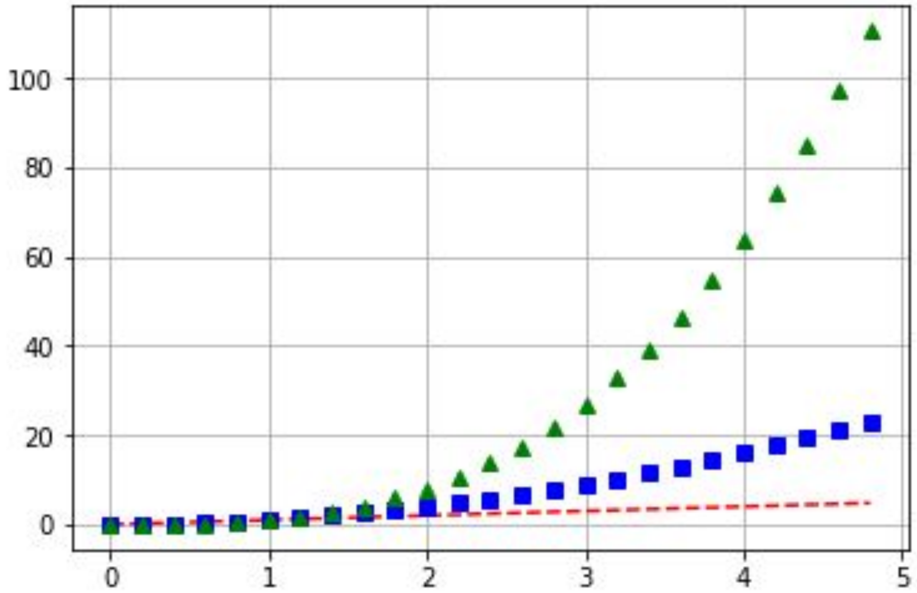


Note: more complicated specification of individual lines can be achieved by adding them to the plot one at a time.

Multiple lines in a single plot: long form

```
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` turns grid lines on/off.

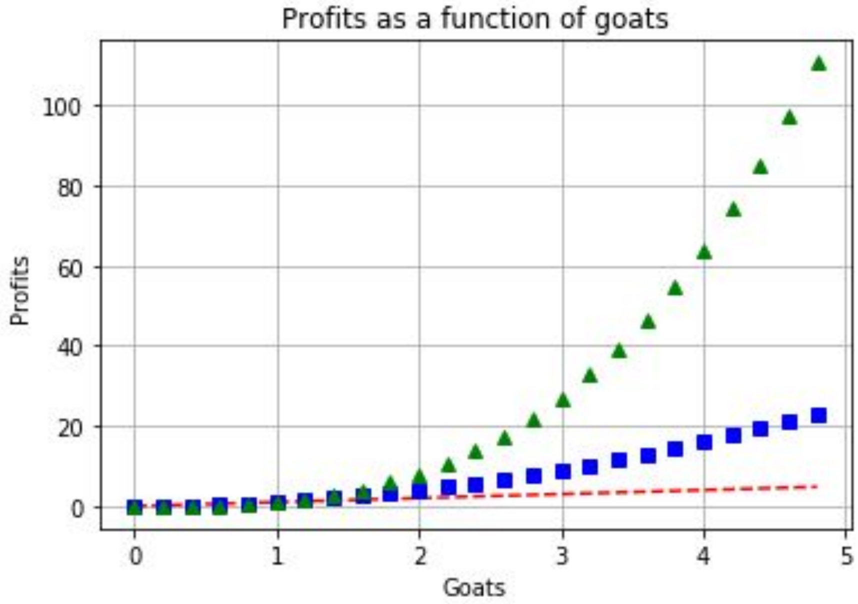


Note: same plot as previous slide, but specifying one line at a time so we could, if we wanted, use more complicated line attributes.

Titles and axis labels

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

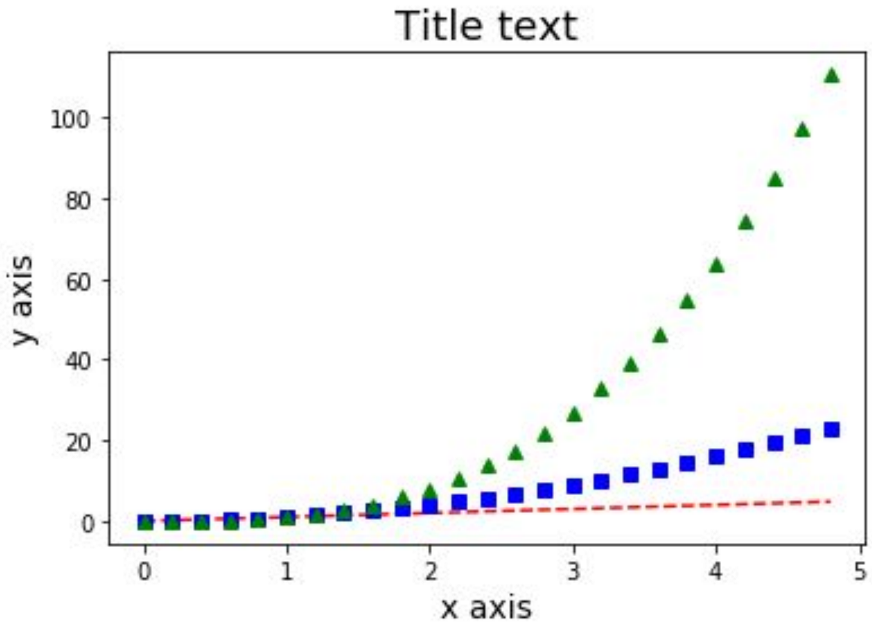
Specifying titles and axis labels couldn't be more straight-forward.



Titles and axis labels

```
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

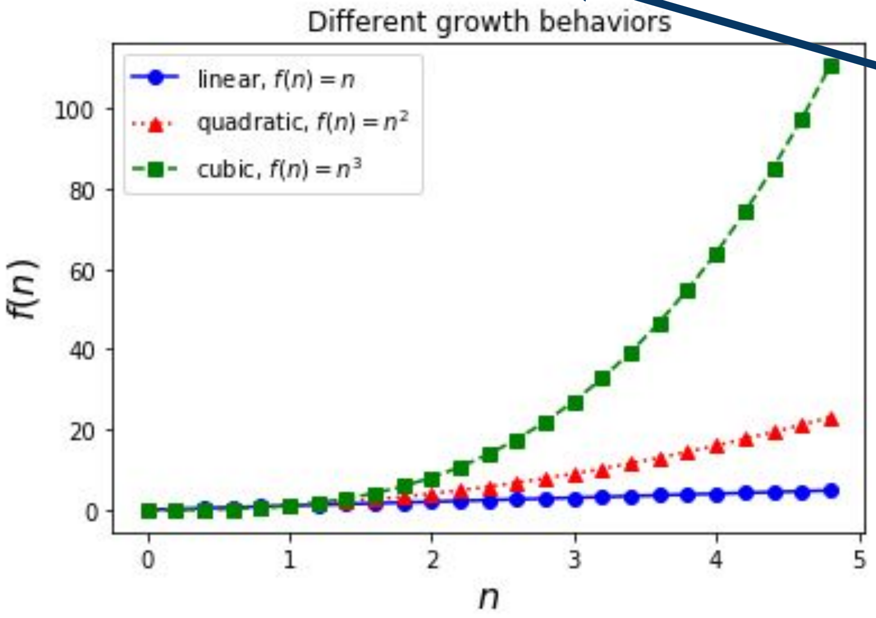


Legends

```
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
```

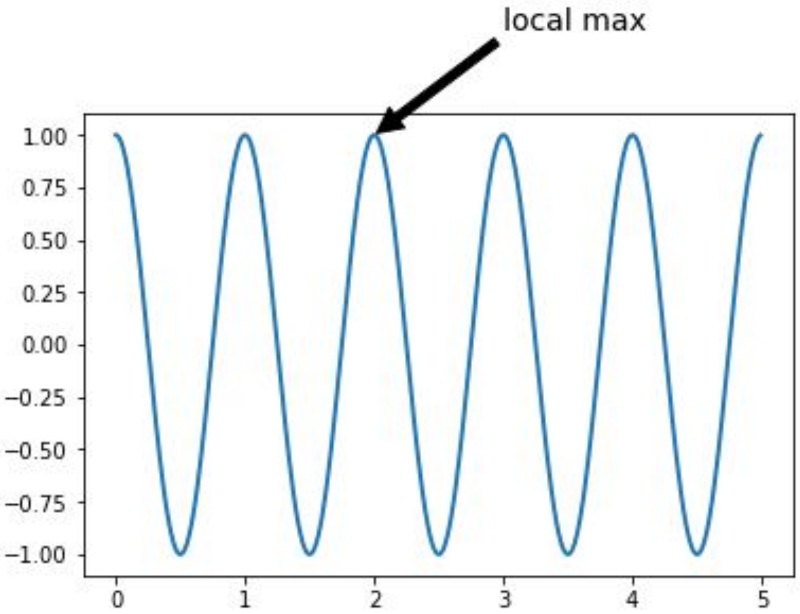
Can use LaTeX in labels, titles, etc.

`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

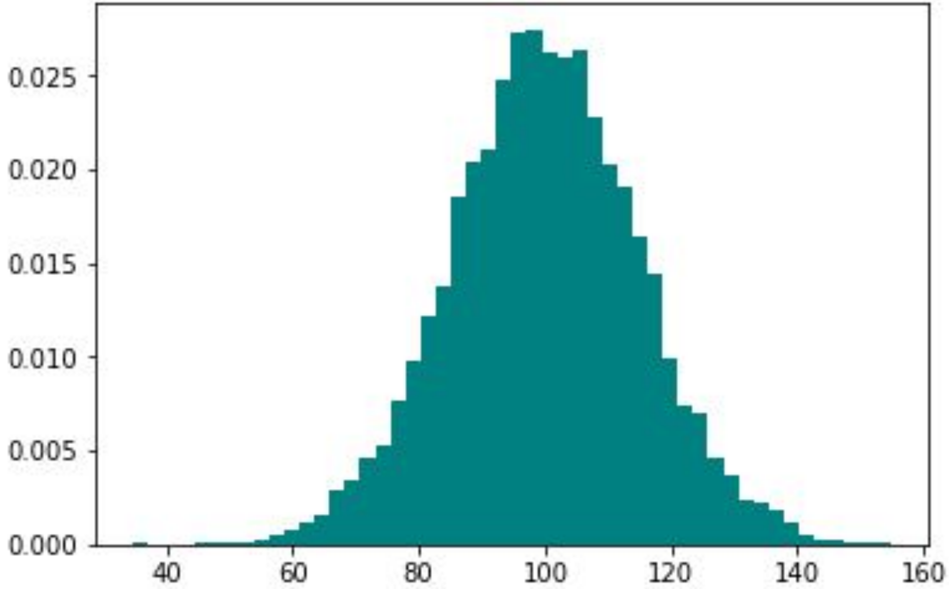
```
t = np.arange(0.0, 5.0, 0.01)
s = np.cos(2*np.pi*t) #np.pi==3.14159...
plt.plot(t, s, lw=2) # plot the cosine.
# Annotate the figure with an arrow and text.
_ = plt.annotate('local max', xy=(2, 1), xytext=(3, 1.5),
             fontsize=14,
             arrowprops=dict(facecolor='black', shrink=0.02)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



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!

Plotting histograms: `pyplot.hist()`

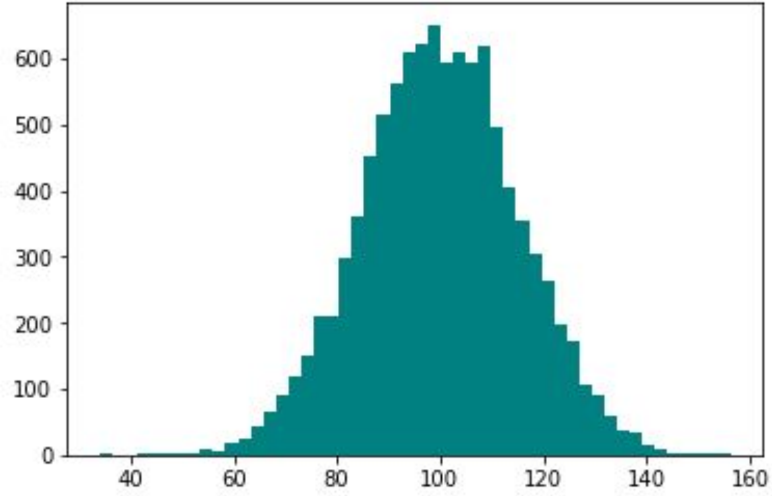
```
mu, sigma = 100, 15
x = np.random.normal(mu,sigma,10000)
# hist( data, nbins, ...)
(n,bins,patches) = plt.hist(x, 50, normed=1, facecolor='teal')
```



Plotting histograms: `pyplot.hist()`

```
(n,bins,patches) = plt.hist(x, 50, normed=0, facecolor='teal')
n
```

```
array([   1.,    0.,    0.,    1.,    1.,    1.,    3.,    3.,    9.,
          7.,   17.,   24.,   43.,   65.,   89.,  120.,  149.,  210.,
        211.,  298.,  362.,  452.,  515.,  563.,  609.,  622.,  651.,
        594.,  608.,  593.,  620.,  496.,  406.,  355.,  304.,  264.,
        197.,  172.,  105.,   90.,   60.,   38.,   34.,   16.,    8.,
          3.,    3.,    4.,    3.,    1.])
```

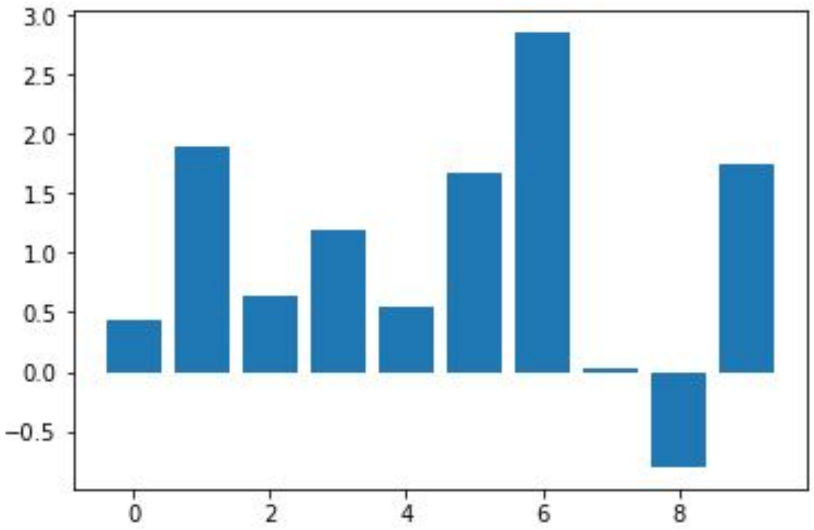


Bin counts. Note that if `normed=1`, then these will be proportions between 0 and 1 instead of counts.

Bar plots

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

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

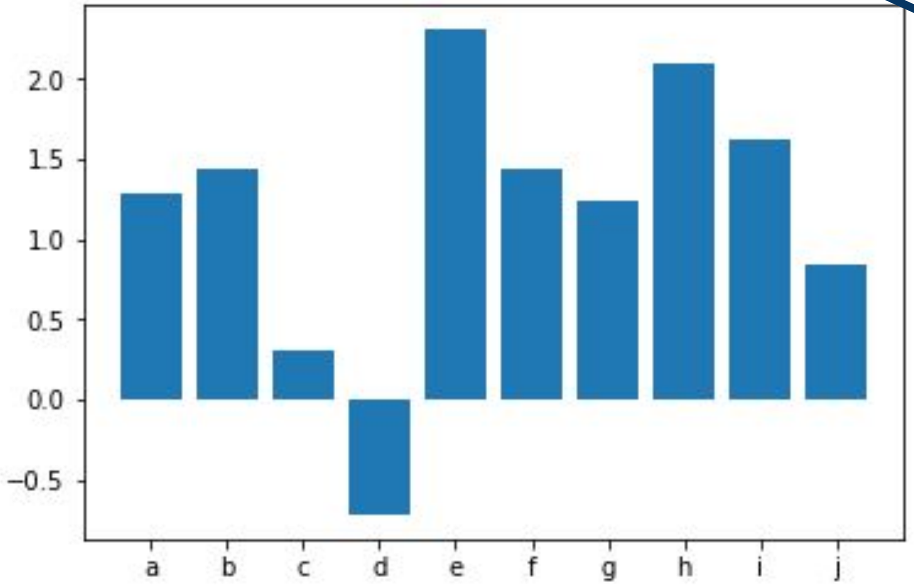


Full set of available arguments to `bar(...)` can be found at http://matplotlib.org/api/_as_gen/matplotlib.pyplot.bar.html#matplotlib.pyplot.bar

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

Tick labels

```
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')
```

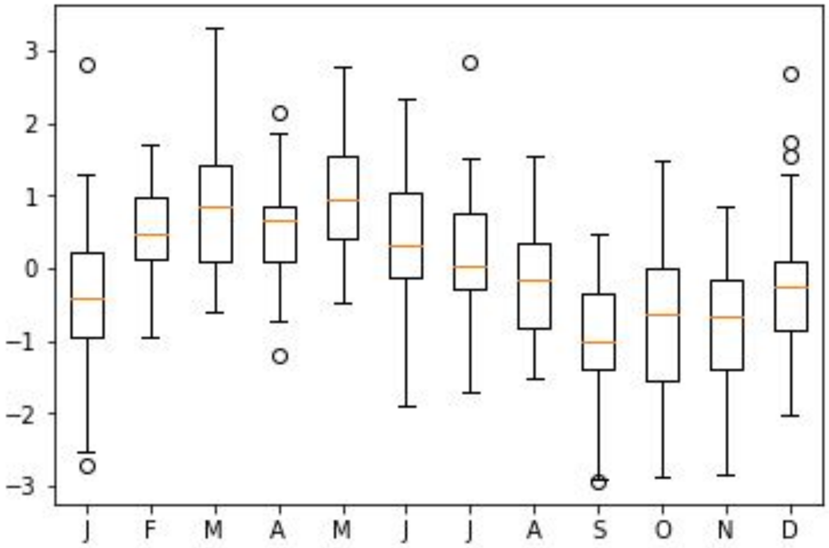


Can specify what the x-axis tick labels should be by using the `tick_label` argument to plot functions.

Box & whisker plots

```
K=12; n=25
draws = np.zeros((n,K))
means = np.zeros((n,1))
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*

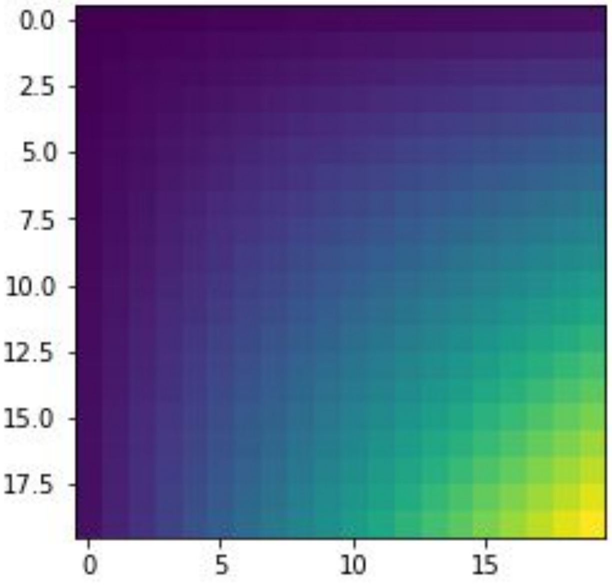
But if you must…

`pyplot.pie(`*`x`*`, … )`

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

Heatmaps and tiling

```python
n=20
x = np.arange(1,n+1)
M = x*np.reshape(x,(n,1))
_ = 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

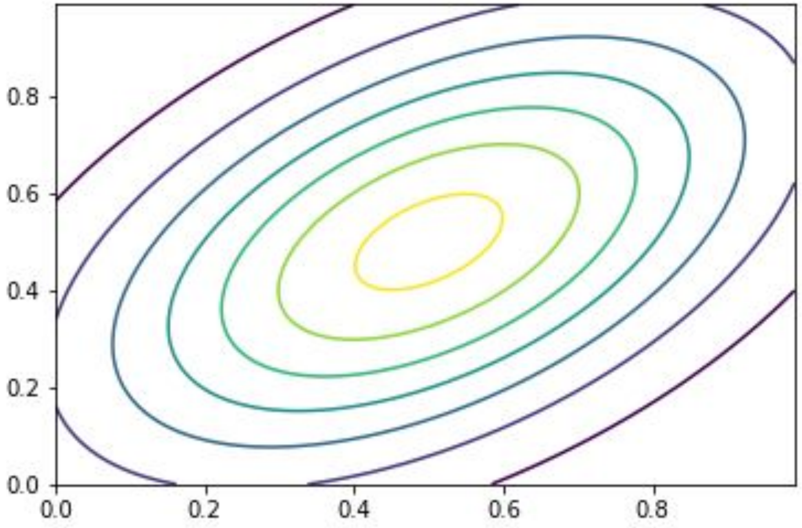
Drawing contours

```
mu=np.array([0.5,0.5])
Sigma=np.array([[0.1,0.05],[0.05,0.1]])
mvn1 = scipy.stats.multivariate_normal(mu,Sigma)

x, y = np.mgrid[0:1:.01, 0:1:.01]
pos = np.empty(x.shape + (2,))
pos[:, :, 0] = x; pos[:, :, 1] = y

_ = plt.contour(x, y, mvn1.pdf(pos))
```

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



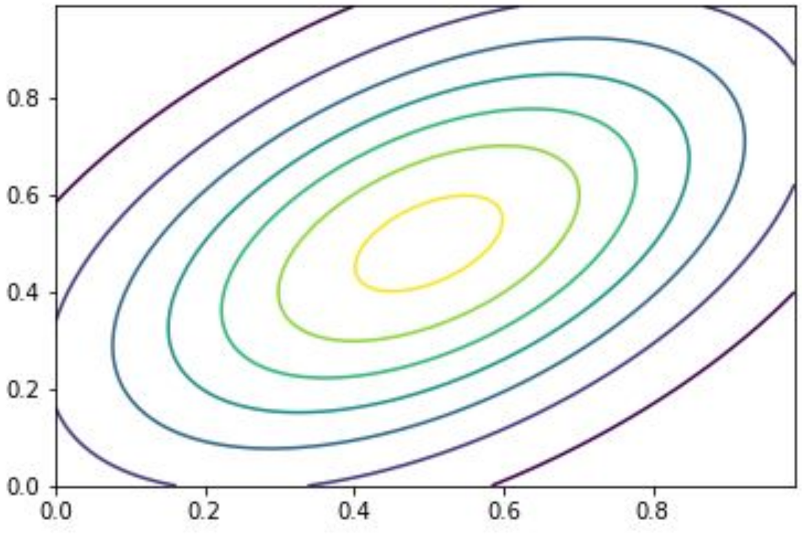
Drawing contours

```
mu=np.array([0.5,0.5])
Sigma=np.array([[0.1,0.05],[0.05,0.1]])
mvn1 = scipy.stats.multivariate_normal(mu,Sigma)

x, y = np.mgrid[0:1:.01, 0:1:.01]
pos = np.empty(x.shape + (2,))
pos[:, :, 0] = x; pos[:, :, 1] = y

_ = 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!



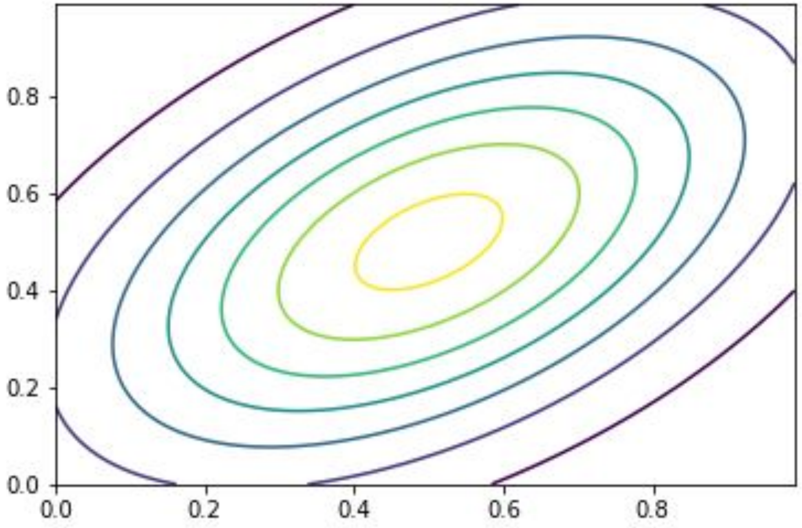
Drawing contours

```
mu=np.array([0.5,0.5])
Sigma=np.array([[0.1,0.05],[0.05,0.1]])
mvn1 = scipy.stats.multivariate_normal(mu,Sigma)

x, y = np.mgrid[0:1:.01, 0:1:.01]
pos = np.empty(x.shape + (2,))
pos[:, :, 0] = x; pos[:, :, 1] = y

_ = plt.contour(x, y, mvn1.pdf(pos))
```

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



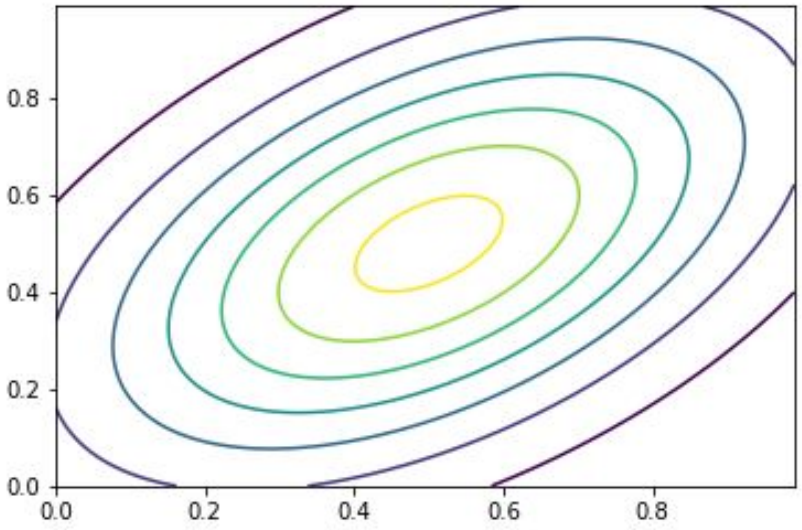
Drawing contours

```
mu=np.array([0.5,0.5])
Sigma=np.array([[0.1,0.05],[0.05,0.1]])
mvn1 = scipy.stats.multivariate_normal(mu,Sigma)

x, y = np.mgrid[0:1:.01, 0:1:.01]
pos = np.empty(x.shape + (2,))
pos[:, :, 0] = x; pos[:, :, 1] = y

_ = 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.



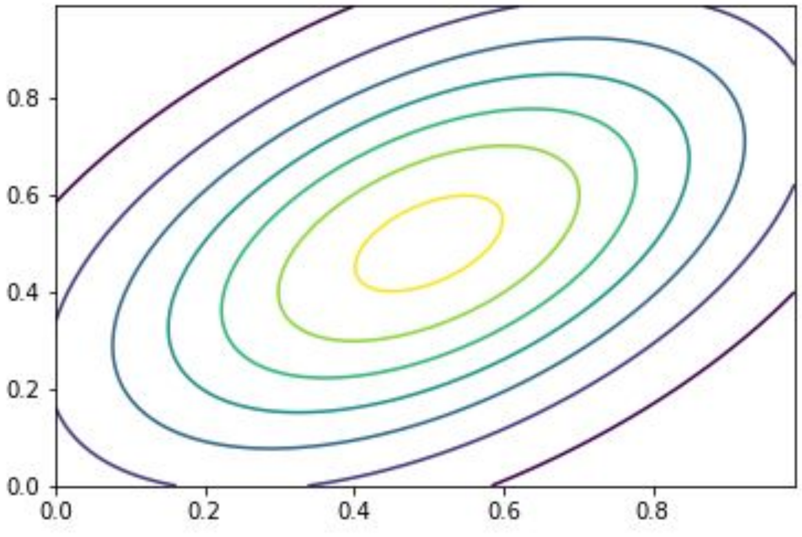
Drawing contours

```
mu=np.array([0.5,0.5])
Sigma=np.array([[0.1,0.05],[0.05,0.1]])
mvn1 = scipy.stats.multivariate_normal(mu,Sigma)

x, y = np.mgrid[0:1:.01, 0:1:.01]
pos = np.empty(x.shape + (2,))
pos[:, :, 0] = x; pos[:, :, 1] = y

_ = 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`.



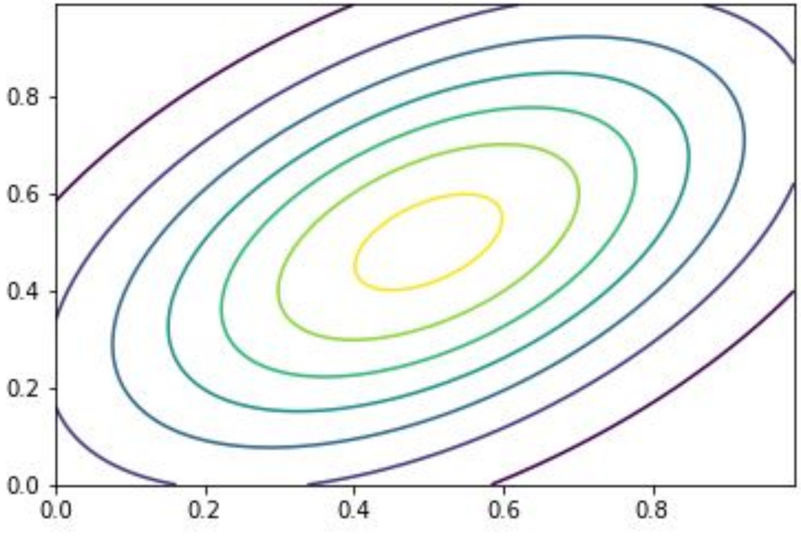
Drawing contours

```
mu=np.array([0.5,0.5])
Sigma=np.array([[0.1,0.05],[0.05,0.1]])
mvn1 = scipy.stats.multivariate_normal(mu,Sigma)

x, y = np.mgrid[0:1:.01, 0:1:.01]
pos = np.empty(x.shape + (2,))
pos[:, :, 0] = x; pos[:, :, 1] = y

_ = 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

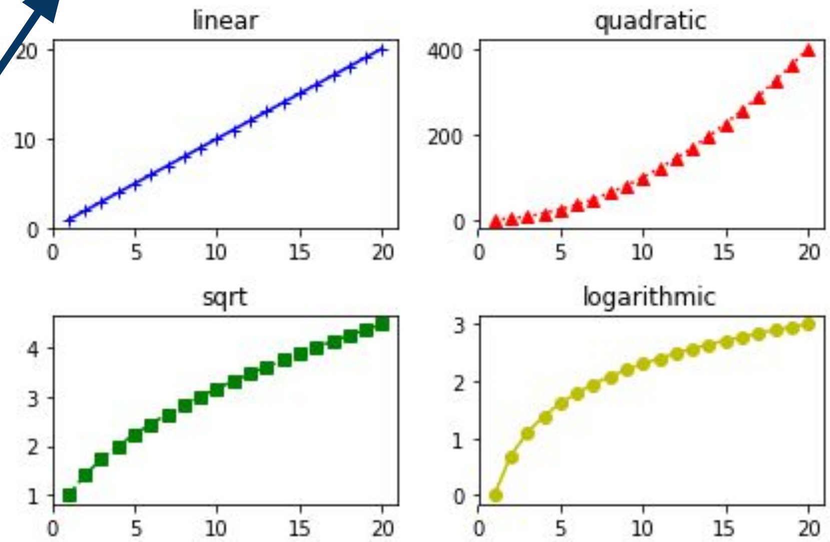
`subplot(nrows, ncols, plot_number)`

Shorthand: `subplot(XYZ)`

- Makes an `X`-by-`Y` plot
- Picks out the `Z`-th plot
- Counting in row-major order

`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()
```



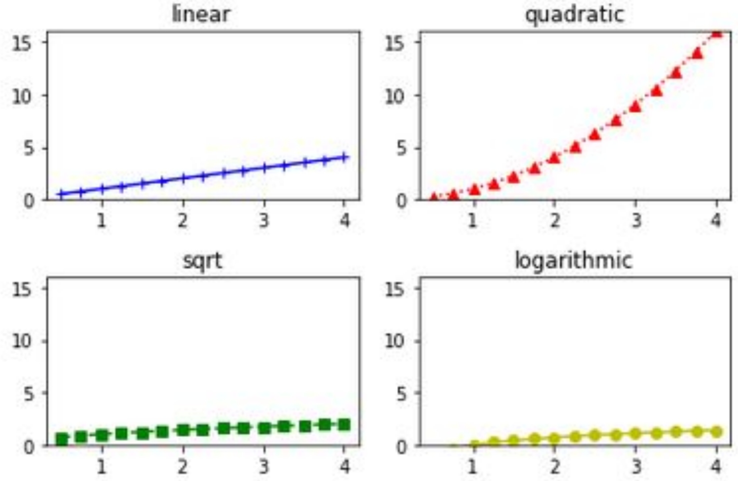
Specifying axis ranges

`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 axes

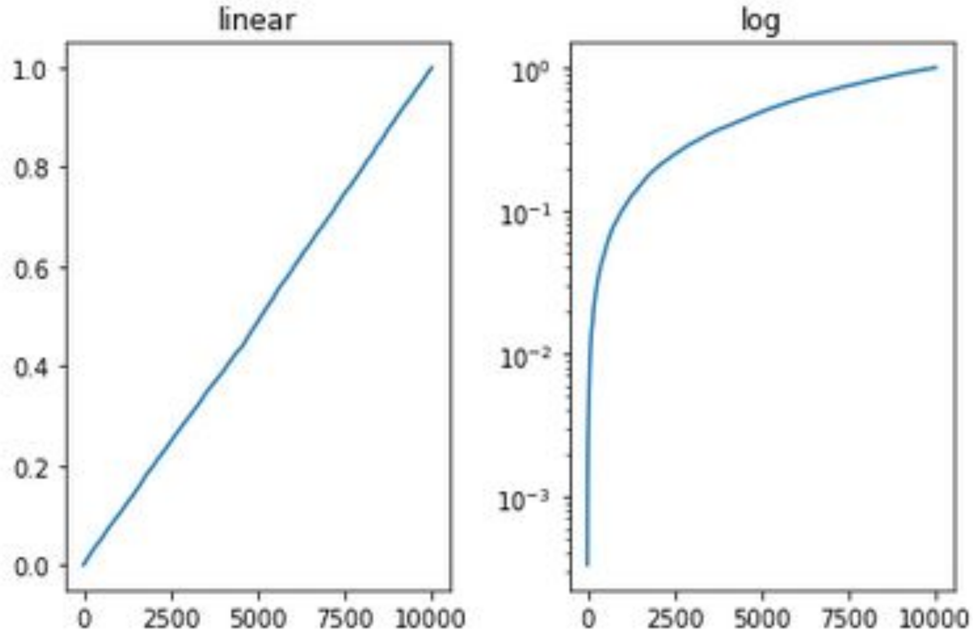
Scale the axes with `plt.xscale` and `plt.yscale`

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

```
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 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

```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

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