Numpy and Matplotlib

Here I have included the code from the Python Boot Camp 2012 lecture on numpy and matplotlib. You can follow along with the material on the slides without having to manually type the code itself. The lecture was designed to provide an introduction to the numpy module (how python handles data arrays) and the matplotlib module (a python plotting module).

First we want to import the appropriate modules into our name space (note this is done automatically with the "--pylab" flag.

In [1]:
```
import numpy as np
```

The primary building block of the numpy module is the class "ndarray". A ndarray object represents a multidimensional, homogeneous array of fixed-sized items. An associated date-type object describes the format of each element in the array. An ndarray object is (almost) never instantiated directly, but instead using a method that returns an instance of the class.

In [2]:
```
a = np.array([1, 2, 3])
```

In [3]:
```
a
```

Out[3]:
```
array([1, 2, 3])
```

The "ones" and "zeros" methods return an array object of the requested shape and type.

In [4]:
```
b = np.ones((3,2))
```

In [5]:
```
b
```

Out[5]:
```
array([[ 1.,  1.],
       [ 1.,  1.],
       [ 1.,  1.]])
```

In [6]:
```
b.shape
```

Out[6]:
```
(3, 2)
```

In [7]:
```
c = np.zeros((1,3), int)
```

In [8]:
```
c
```

Out[8]:
```
array([[0, 0, 0]])
```

In [9]:
```
type(c)
```

Out[9]:
```
numpy.ndarray
```

In [10]:
```
c.dtype
```

Out[10]:
```
dtype('int64')
```

"linspace" creates a one-dimensional array running from arg1 to arg2 (with length arg3).

In [11]:
```
d = np.linspace(1,5,11)
```

In [12]:
```
d
```

Out[12]:
```
array([ 1. ,  1.4,  1.8,  2.2,  2.6,  3. ,  3.4,  3.8,  4.2,  4.6,  5. ])
```

numpy provides a variety of methods to read and write data to disk (binary, ascii, fits, csv, etc.). These include "loadtxt" and "tofile". I haven't included these because of limitations with the Notebook.

ndarray objects can be indexed, sliced, and iterated over much like lists. The format for slicing is still "x1:x2:dx".

In [22]:
```
a = np.arange(10)
```

In [23]:
```
a
```

Out[23]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [24]:
```
a[2]
```

Out[24]:
```
2
```

In [25]:
```
a[2:5]
```

Out[25]:
```
array([2, 3, 4])
```

In [27]:
```
a[:6:2] = -1000
```

In [28]:
```
a
```

Out[28]:
```
array([-1000,     1, -1000,     3, -1000,     5,     6,     7,     8,     9])
```

In [29]:
```
a[::-1]
```

Out[29]:
```
array([    9,     8,     7,     6,     5, -1000,     3, -1000,     1, -1000])
```

In [30]:
```
a[2:-2]
```

Out[30]:
```
array([-1000,     3, -1000,     5,     6,     7])
```

Arrays can hold (almost) any type of data, as long as each individual element is identical (i.e., requires the same amount of memory). The format of the ndarray can be specified with the "dtype" attribute. Individual elements may be "named" in a structured array.

In [31]:
```
x = np.zeros((2,),dtype=('i4,f4,a10'))
```

In [32]:
```
x
```

Out[32]:
```
array([(0, 0.0, ''), (0, 0.0, '')],
      dtype=[('f0', '<i4'), ('f1', '<f4'), ('f2', '|S10')])
```

In [33]:
```
x['f1']
```

Out[33]:
```
array([ 0.,  0.], dtype=float32)
```

Note that the same issues of references and copies that apply to other variables also apply to array objects.

In [34]:
```
y = x['f1']
```

In [35]:
```
y
```

Out[35]:
```
array([ 0.,  0.], dtype=float32)
```

In [36]:
```
y += np.array([1.0, 1.0])
```

In [37]:
```
y
```

Out[37]:
```
array([ 1.,  1.], dtype=float32)
```

In [38]:
```
x
```

Out[38]:
```
array([(0, 1.0, ''), (0, 1.0, '')],
      dtype=[('f0', '<i4'), ('f1', '<f4'), ('f2', '|S10')])
```

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. Examples include add, subtract, multiply, exp, log, and power.

Most array operations thus occur on an element-by-element basis:

In [39]:
```
a = np.array([[1, 2], [3, 4]])
```

In [40]:
```
b = np.array([[2, 3], [4, 5]])
```

In [41]:
```
a + b
```

```
Out[41]: array([[3, 5],
                [7, 9]])
```

```
In [42]: np.multiply(a, b)
```

```
Out[42]: array([[ 2,  6],
                [12, 20]])
```

```
In [43]: a ** b
```

```
Out[43]: array([[   1,    8],
                [  81, 1024]])
```

Standard linear algebra (i.e., matrix) operations are also available. Many are stored in the linalg module.

```
In [44]: np.dot(a,b)
```

```
Out[44]: array([[10, 13],
                [22, 29]])
```

Universal functions run *much* faster than for loops, which should be avoided whenever possible.

```
In [45]: a = np.random.random((500,500))
```

```
In [46]: b = np.random.random((500,500))
```

```
In [48]: def mult1(a,b):
             return a * b
```

```
In [50]: def mult2(a,b):
             c = np.empty(a.shape)
             for i in range(a.shape[0]):
                 for j in range(a.shape[1]):
                     c[i,j] = a[i,j] * b[i,j]
             return c
```

```
In [51]: timeit mult1(a,b)
```

```
100 loops, best of 3: 2.12 ms per loop
```

```
In [52]: timeit mult2(a,b)
```

```
1 loops, best of 3: 328 ms per loop
```

numpy will (usually) intelligently deal with arrays of different sizes. The smaller array is *broadcast* across the larger array so that they have compatible shapes. Note that the rules for broadcasting are not always intuitive, so be careful!

In [53]:
```
a=np.array([1,2,3.])
```

In [54]:
```
a + 2
```

Out[54]:
```
array([ 3.,  4.,  5.])
```

In [55]:
```
b=np.array([10,20,30.,40])
```

In [60]:
```
a * b
```

Out[60]:
```
array([[  10.,   20.,   30.,   40.],
       [  20.,   40.,   60.,   80.],
       [  30.,   60.,   90.,  120.]])
```

In [57]:
```
a = a.reshape(3,1)
```

In [58]:
```
a
```

Out[58]:
```
array([[ 1.],
       [ 2.],
       [ 3.]])
```

In [59]:
```
a * b
```

Out[59]:
```
array([[  10.,   20.,   30.,   40.],
       [  20.,   40.,   60.,   80.],
       [  30.,   60.,   90.,  120.]])
```

Universal functions make it nearly trivial to compare arrays on an element-by-element basis.

In [61]:
```
a = np.array([1, 3, 0], float)
```

In [62]:
```
b = np.array([0, 3, 2], float)
```

In [63]:
```
a > b
```

Out[63]:
```
array([ True, False, False], dtype=bool)
```

In [64]:
```
a == b
```

Out[64]:
```
array([False,  True, False], dtype=bool)
```

In [65]:
```
c = a <= b
```

In [66]:
```
c
```

Out[66]:
```
array([False,  True,  True], dtype=bool)
```

In [67]:
```
np.logical_and(a > 0, a < 3)
```

Out[67]:
```
array([ True, False, False], dtype=bool)
```

In [68]:
```
np.logical_or(a,b)
```

Out[68]:
```
array([ True,  True,  True], dtype=bool)
```

The *where* method provides a fast way to search (and extract) individual elements of an array. When called with a single (conditional) argument, the method returns an array of indicies where the conditional is met. If two additional arguments are added, more complex returns are possible.

In [69]:
```
a = np.array([1, 3, 0, -5, 0], float)
```

In [70]:
```
np.where(a != 0)
```

Out[70]:
```
(array([0, 1, 3]),)
```

In [71]:
```
a[a != 0]
```

Out[71]:
```
array([ 1.,  3., -5.])
```

In [73]:
```
np.where(a != 0.0, 1 / a, a)
```

Out[73]:
```
array([ 1.        ,  0.33333333,  0.        , -0.2       ,  0.        ])
```

In [74]:
```
x = np.arange(9.).reshape(3, 3)
```

In [75]:
```
x
```

Out[75]:
```
array([[ 0.,  1.,  2.],
       [ 3.,  4.,  5.],
       [ 6.,  7.,  8.]])
```

In [76]:
```
np.where( x > 5 )
```

Out[76]:
```
(array([2, 2, 2]), array([0, 1, 2]))
```

ndarray objects provide basic statistical methods (mean, median, standard deviation, etc.).

In [77]:
```
a = np.array([[1, 2], [3, 4]])
```

In [78]:
```
np.mean(a)
```

Out[78]:
```
2.5
```

In [79]:
```
np.mean(a, axis=0)
```

In [79]:

Out[79]: array([2., 3.])

In [80]:
```
np.mean(a, axis=1)
```

Out[80]: array([1.5, 3.5])

In [81]:
```
np.std(a)
```

Out[81]: 1.1180339887498949

In [82]:
```
np.average(range(1,11), weights=range(10,0,-1))
```

Out[82]: 4.0

The *random* module contains basic random number generation, as well as a few common probability distribution functions. Many more (complex) pdfs are available within scipy.

In [83]:
```
np.random.rand(5)
```

Out[83]: array([0.30690353, 0.30895097, 0.21526229, 0.3493788 , 0.16837581])

In [84]:
```
np.random.randint(5, 10)
```

Out[84]: 9

In [85]:
```
np.random.normal(1.5, 4.0)
```

Out[85]: 3.0703190817944908

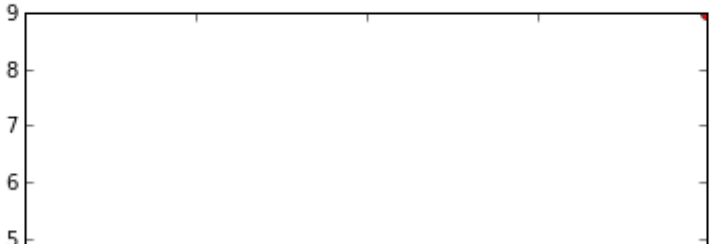
Plotting is done with the *matplotlib* module. If you have used MATLAB before, the syntax should look very familiar.

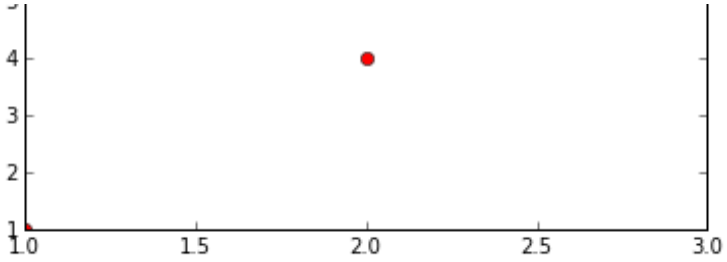
In [86]:
```
import matplotlib.pylab as plt
```

The simplest (but still quite powerful) method is *plot*.

In [87]:
```
x = np.array([1,2,3])
y = x**2
plt.plot(x, y, "ro")
```

Out[87]: [<matplotlib.lines.Line2D at 0x10523ba90>]

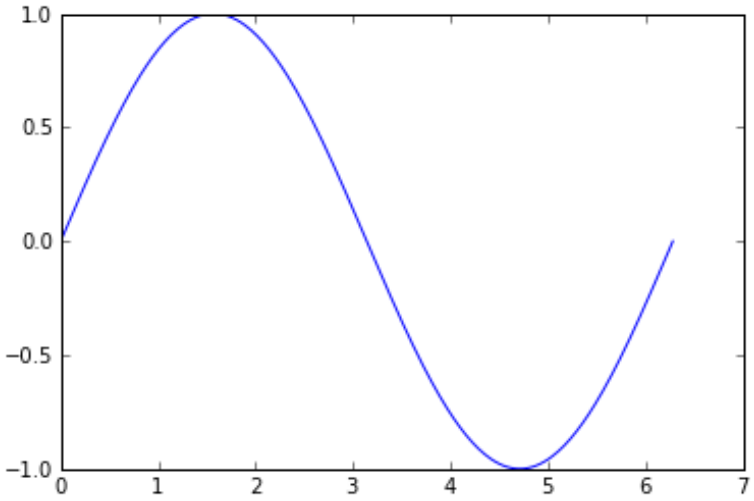




Another basic example.

In [88]:
```
x = np.linspace(0, 2*np.pi, 300)
y = np.sin(x)
plt.plot(x, y)
```

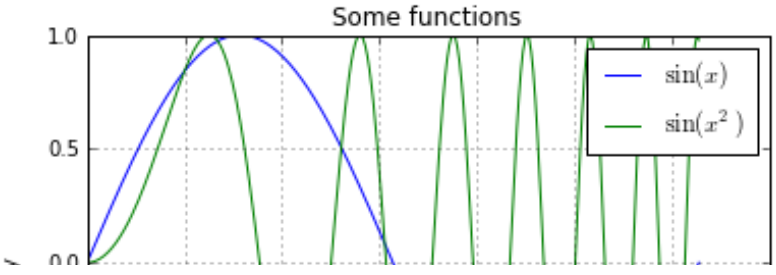
Out[88]: [<matplotlib.lines.Line2D at 0x1057620d0>]

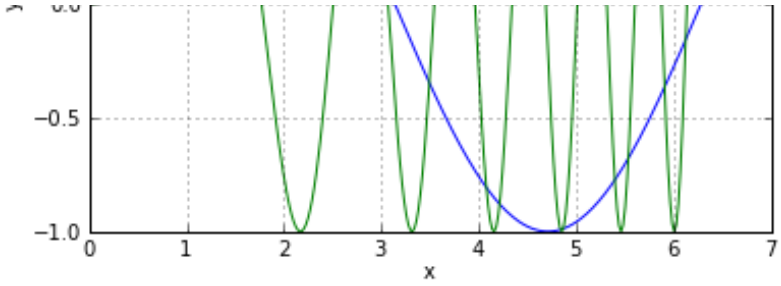


Here's a more realistic plot, including how to modify axis labels, create a legend, etc.

In [91]:
```
x = np.linspace(0, 2*np.pi, 300)
y = np.sin(x)
y2 = np.sin(x**2)
plt.plot(x, y, label=r'$\sin(x)$')
plt.plot(x, y2, label=r'$\sin(x^2)$')
plt.title('Some functions')
plt.xlabel('x')
plt.ylabel('y')
plt.grid()
plt.legend()
```

Out[91]: <matplotlib.legend.Legend at 0x105938bd0>



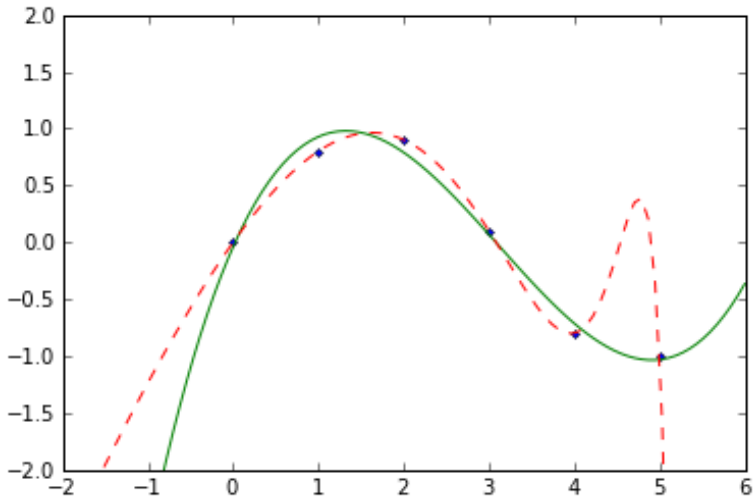


numpy also provides capabilities for (least squares) polynomial fitting.

In [92]:
```
x = np.array([0.0, 1.0, 2.0, 3.0,  4.0,  5.0])
y = np.array([0.0, 0.8, 0.9, 0.1, -0.8, -1.0])
z = np.polyfit(x, y, 3)
p = np.poly1d(z)
p30 = np.poly1d(np.polyfit(x, y, 30))
xp = np.linspace(-2, 6, 100)
plt.plot(x, y, '.', xp, p(xp), '-', xp, p30(xp), '--')
plt.ylim(-2,2)
```

```
/opt/local/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/site-packages/numpy/lib/polynomial.py:560: RankWarning: Polyfit may be poorly conditioned
  warnings.warn(msg, RankWarning)
```

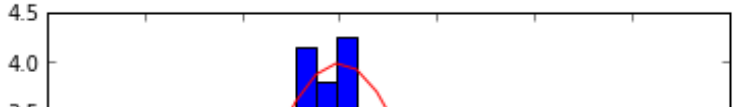
Out[92]: (-2, 2)

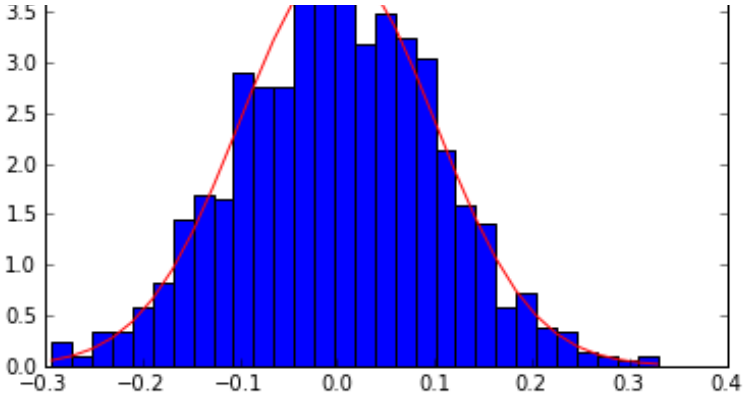


Here's an example of the histogram plotting method (as well as the random number generation).

In [93]:
```
mu, sigma = 0, 0.1
s = np.random.normal(mu, sigma, 1000)
count, bins, ignored = plt.hist(s, 30, normed=True)
plt.plot(bins, 1/(sigma * np.sqrt(2 * np.pi)) * np.exp( - (bins - mu)**2 / (2
```

Out[93]: [<matplotlib.lines.Line2D at 0x105900c50>]





Probably the best way to learn about making new plots is looking at the Matplotlib Gallery.