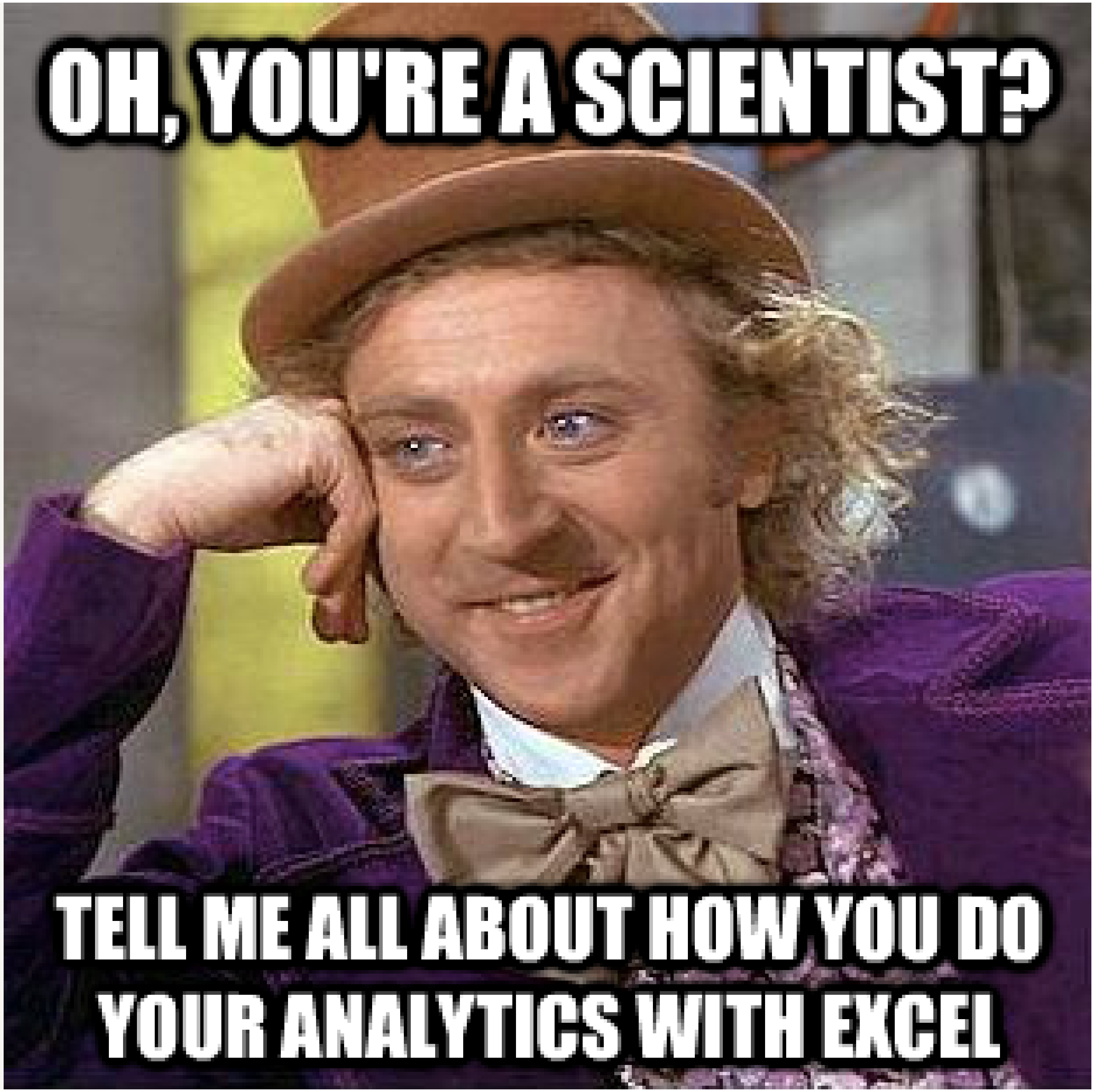
Numpy (Arrays) and Matplotlib (Plotting)

Brad Cenko

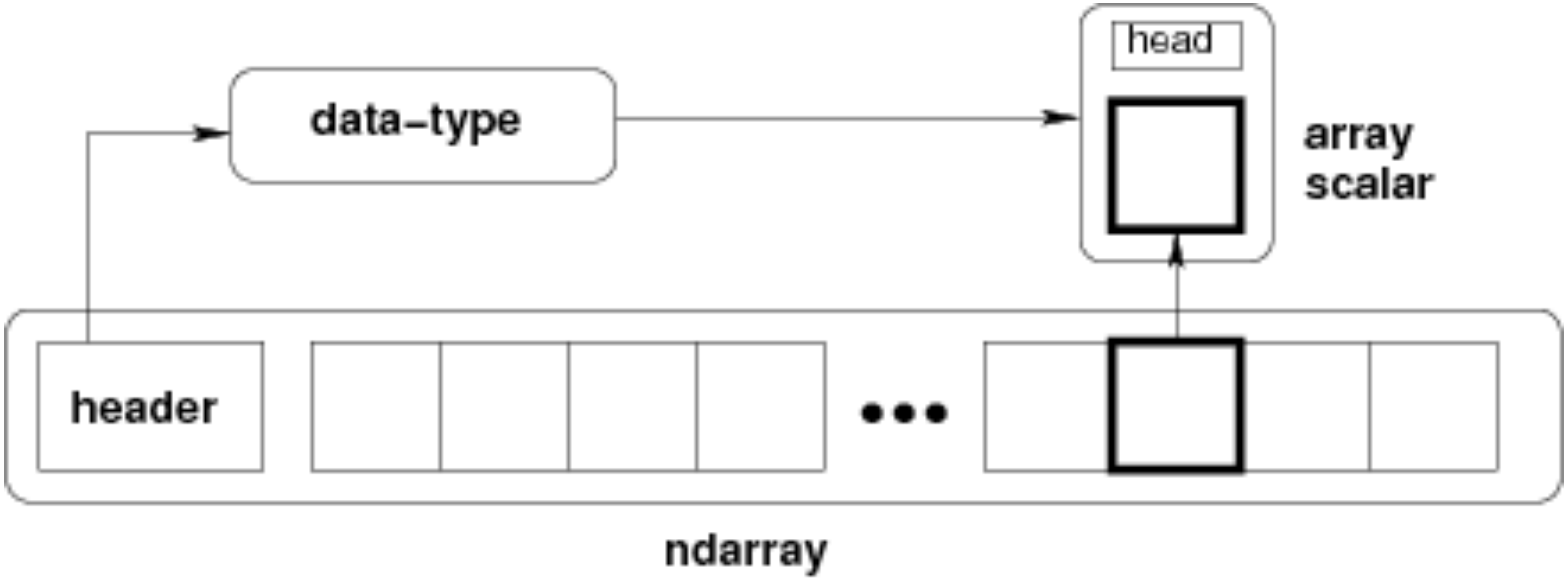
20 August 2012


OH, YOU'RE A SCIENTIST?
TELL ME ALL ABOUT HOW YOU DO YOUR ANALYTICS WITH EXCEL

Overview: *numpy* and *matplotlib*

- Array creation and basic operations
- Universal functions and broadcasting
- Comparison testing, selection, and manipulation
- Basic statistics
- Basic plotting capabilities

ndarray class



An array object represents a multidimensional, homogeneous array of fixed-size items. An associated data-type object describes the format of each element in the array (its byte-order, how many bytes it occupies in memory, whether it is an integer, a floating point number, or something else, etc.)

Instantiating *ndarrays*

```
>>> import numpy as np
>>> a = np.array([1, 2, 3])
>>> a
array([1, 2, 3])
>>> b = np.ones((3,2))
>>> b
array([[ 1.,  1.],
       [ 1.,  1.],
       [ 1.,  1.]])
>>> b.shape
(3,2)
>>> c = np.zeros((1,3), int)
>>> c
array([[0, 0, 0]])
>>> type(c)
<type 'numpy.ndarray'>
>>> c.dtype
dtype('int64')
>>> d = np.linspace(1,5,11)
>>> d
array([ 1. ,  1.4,  1.8,  2.2,  2.6,  3. ,  3.4,
3.8,  4.2,  4.6,  5. ])
```

ndarrays are (almost) never instantiated directly, but instead using a method that returns one

Instantiating *ndarrays*

```
>>> a = np.array([1, 2, 3.0])
>>> a.dtype
dtype('float64')
>>> a[0]
1.0
>>> b = np.array([1, 2, '3'])
>>> b
array(['1', '2', '3'],
      dtype='|S1')
>>> b[2] = 12.0
>>> b
array(['1', '2', '1'],
      dtype='|S1')
>>> c = np.array([1, 2, 3])
>>> c[0] = 1.5
>>> c
array([1, 2, 3])
>>> c.dtype='float64'
>>> c
array([  4.94065646e-324,
```

Instantiating *ndarrays*

```
[Xavi:~] cenko% less data.txt
1 2
3 4
data.txt (END)

>>> a = np.loadtxt("data.txt")
>>> a
array([[ 1.,  2.],
       [ 3.,  4.]])
>>> a.tofile("data.out1")
>>> a.tofile("data.out2", sep=",", format="%f")

[Xavi:~] cenko% less data.out1
"data.out1" may be a binary file.  See it anyway?
^@^@^@^@^@^@^T@^@^@^@^@^@^@@^@^@^@^@^@^@^H@^@^@^
@^@^@^@^P@
data.out1 (END)
[Xavi:~] cenko% less data.out2
5.000000,2.000000,3.000000,4.000000
data.out2 (END)
```

ndarrays can also be directly read from / written to files. There are modules for csv, fits, jpg,

Manipulations, Slicing, and Indexing

```
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a[2]
2
>>> a[2:5]
array([2, 3, 4])
>>> a[:6:2] = -1000
>>> a
array([-1000,     1, -1000,     3, -1000,     5,     6,
7,     8,     9])
>>> a[::-1]
array([    9,     8,     7,     6,     5, -1000,     3,
-1000,     1, -1000])
>>> a[2:-2]
array([-1000,     3, -1000,     5,     6,     7])
```

ndarray objects can be indexed, sliced, and iterated over much like lists

Structured Arrays

```
>>> x = np.zeros((2,),dtype=('i4,f4,a10'))
>>> x
array([(0, 0.0, ''), (0, 0.0, '')],
      dtype=[('f0', '<i4'), ('f1', '<f4'), ('f2', '|S10')])
>>> x['f1']
array([ 0.,  0.], dtype=float32)
>>> y = x['f1']
>>> y
array([ 0.,  0.], dtype=float32)
>>> y += np.array([1.0, 1.0])
>>> y
array([ 1.,  1.], dtype=float32)
>>> x
array([(0, 1.0, ''), (0, 1.0, '')],
      dtype=[('f0', '<i4'), ('f1', '<f4'), ('f2', '|S10')])
```

ndarrays can be composed of (almost) any data type. The data type is specified by the *dtype* attribute.

Universal Functions

A universal function (or *<u>ufunc</u>* for short) is a function that operates on **<u>ndarrays</u>** 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.*

Universal Functions

```
>>> a = np.array([[1, 2], [3, 4]])
>>> b = np.array([[2, 3], [4, 5]])
>>> a + b
array([[3, 5],
       [7, 9]])
>>> np.multiply(a, b)
array([[ 2,  6],
       [12, 20]])
>>> a ** b
array([[   1,    8],
       [  81, 1024]])
>>> np.dot(a,b)
array([[10, 13],
       [22, 29]])
```

Universal functions operate on an *element-by-element* basis.

Universal Functions

```
>>> a = np.random.random((500,500))
>>> b = np.random.random((500,500))
>>> def mult1(a,b):
...     return a*b
...
>>> 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
...
>>> timeit mult1(a,b)
100 loops, best of 3: 2.13 ms per loop
>>> timeit mult2(a,b)
1 loops, best of 3: 320 ms per loop
```

Universal functions run **much** faster than for loops (which should be avoided whenever possible)

Note the "timeit" function (as written) requires ipython

Broadcasting

```
>>> a=np.array([1,2,3.])
>>> a + 2
array([ 3.,  4.,  5.])
>>> b=np.array([10,20,30.,40])
>>> a*b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operands could not be broadcast
together with shapes (3) (4)
>>> a = a.reshape(3,1)
>>> a
array([[ 1.],
       [ 2.],
       [ 3.]])
>>> a*b
array([[  10.,   20.,   30.,   40.],
       [  20.,   40.,   60.,   80.],
       [  30.,   60.,   90.,  120.]])
```

numpy will intelligently deal with *ndarrays* of different shapes. The smaller array is *broadcast* across the larger array so that they have compatible shapes

Comparison Testing and Selection

```
>>> a = np.array([1, 3, 0], float)
>>> b = np.array([0, 3, 2], float)
>>> a > b
array([ True, False, False], dtype=bool)
>>> a == b
array([False,  True, False], dtype=bool)
>>> c = a <= b
>>> c
array([False,  True,  True], dtype=bool)
>>> np.logical_and(a > 0, a < 3)
array([ True, False, False], dtype=bool)
>>> np.logical_or(a,b)
array([ True,  True,  True], dtype=bool)
```

ndarrays can be compared on an element-by-element basis

Comparison Testing and Selection

```
>>> a = np.array([1, 3, 0, -5, 0], float)
>>> np.where(a != 0)
(array([0, 1, 3]),)
>>> a[a != 0]
array([ 1.,  3., -5.])
>>> np.where(a != 0.0, 1 / a, a)
array([ 1.        ,  0.33333333,  0.        , -0.2
0.        ])
>>> x = np.arange(9.).reshape(3, 3)
>>> x
array([[ 0.,  1.,  2.],
       [ 3.,  4.,  5.],
       [ 6.,  7.,  8.]])
>>> np.where( x > 5 )
(array([2, 2, 2]), array([0, 1, 2]))
```

where provides a fast way to search (and , extract) individual elements of an *ndarray* (see also *nonzero*).

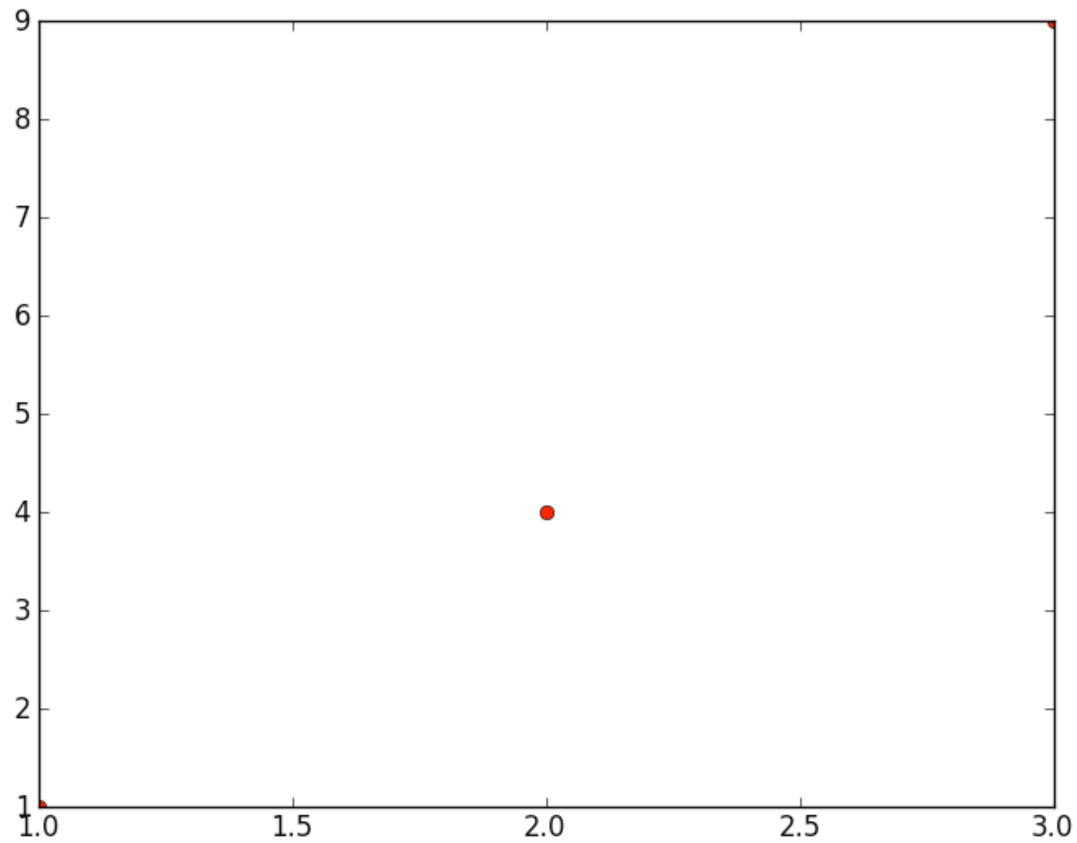
Basic Statistics

```
>>> a = np.array([[1, 2], [3, 4]])
>>> np.mean(a)
2.5
>>> np.mean(a, axis=0)
array([ 2.,  3.])
>>> np.mean(a, axis=1)
array([ 1.5,  3.5])
>>> np.std(a)
1.1180339887498949
>>> np.average(range(1,11), weights=range(10,0,-1))
4.0
>>> np.random.rand(5)
array([ 0.69759058,  0.90690445,  0.73032438,
0.58342295,  0.85800379])
>>> np.random.randint(5, 10)
8
>>> np.random.normal(1.5, 4.0)
0.3285939517604457
```

Basic statistics can be calculated with built-in *numpy* routines. More complicated tasks require *scipy*.

matplotlib Basics

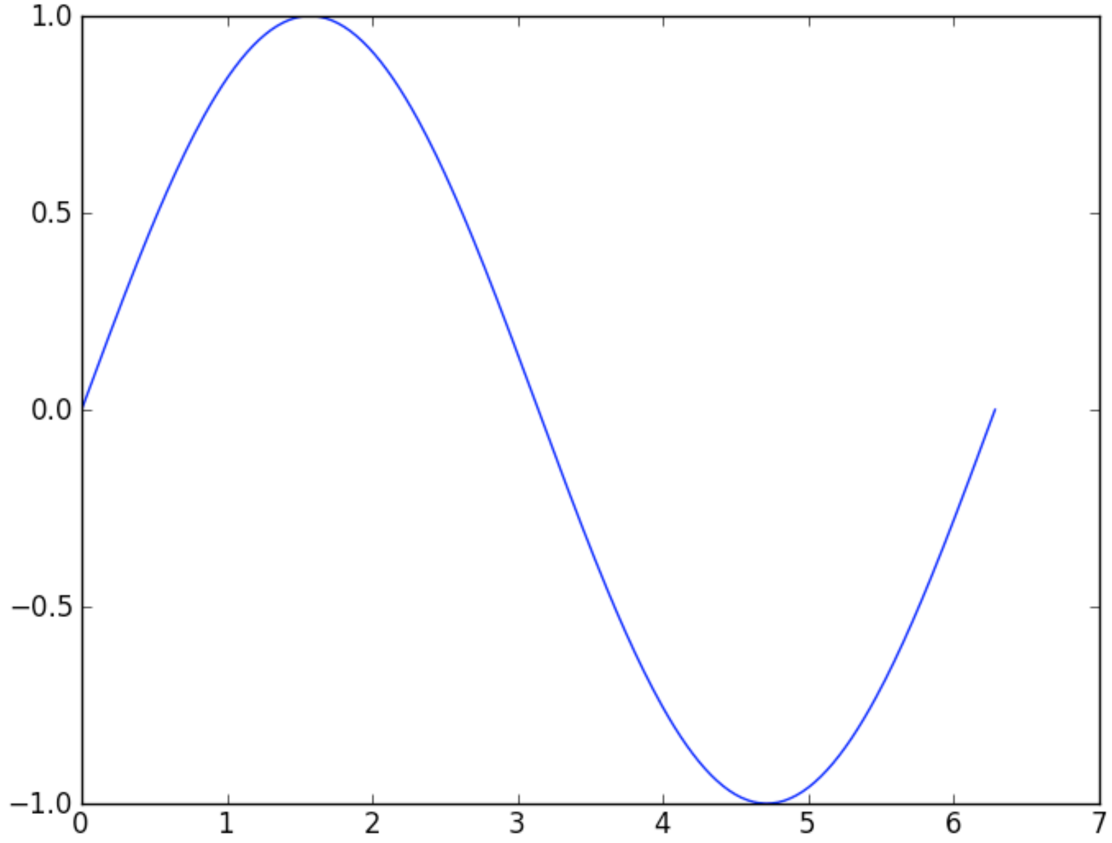
```
>>> import matplotlib.pylab as plt
>>> x = np.array([1,2,3])
>>> y = x**2
>>> plt.plot(x, y, "ro")
[<matplotlib.lines.Line2D object at 0x1032bb1d0>]
>>> plt.show()
```



The *matplotlib* module provides publication quality figures with a MATLAB-like syntax

matplotlib Basics

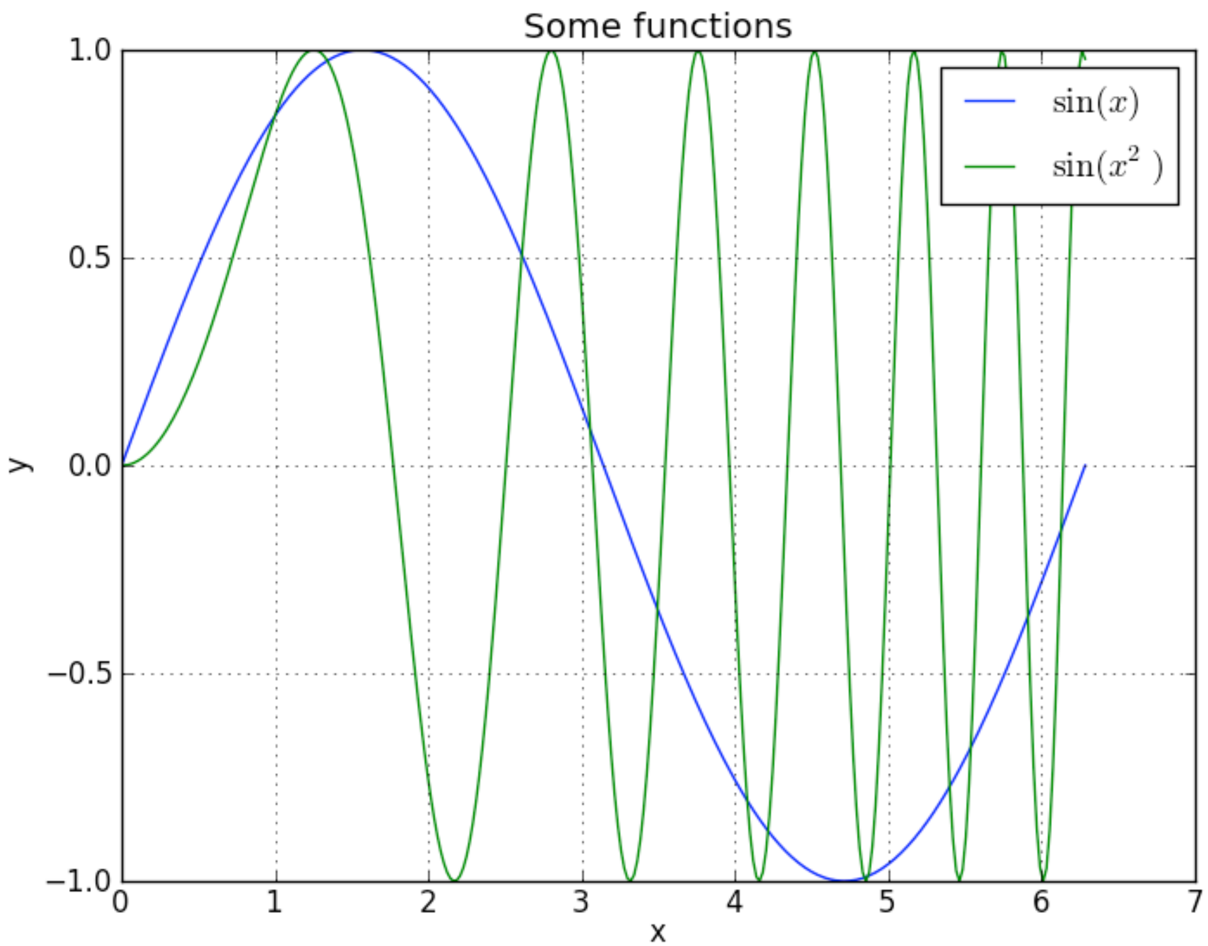
```
>>> x = np.linspace(0, 2*np.pi, 300)
>>> y = np.sin(x)
>>> plt.plot(x, y)
[<matplotlib.lines.Line2D object at 0x1173aead0>]
>>> plt.show()
```



The *matplotlib* module provides publication quality figures with a MATLAB-like syntax

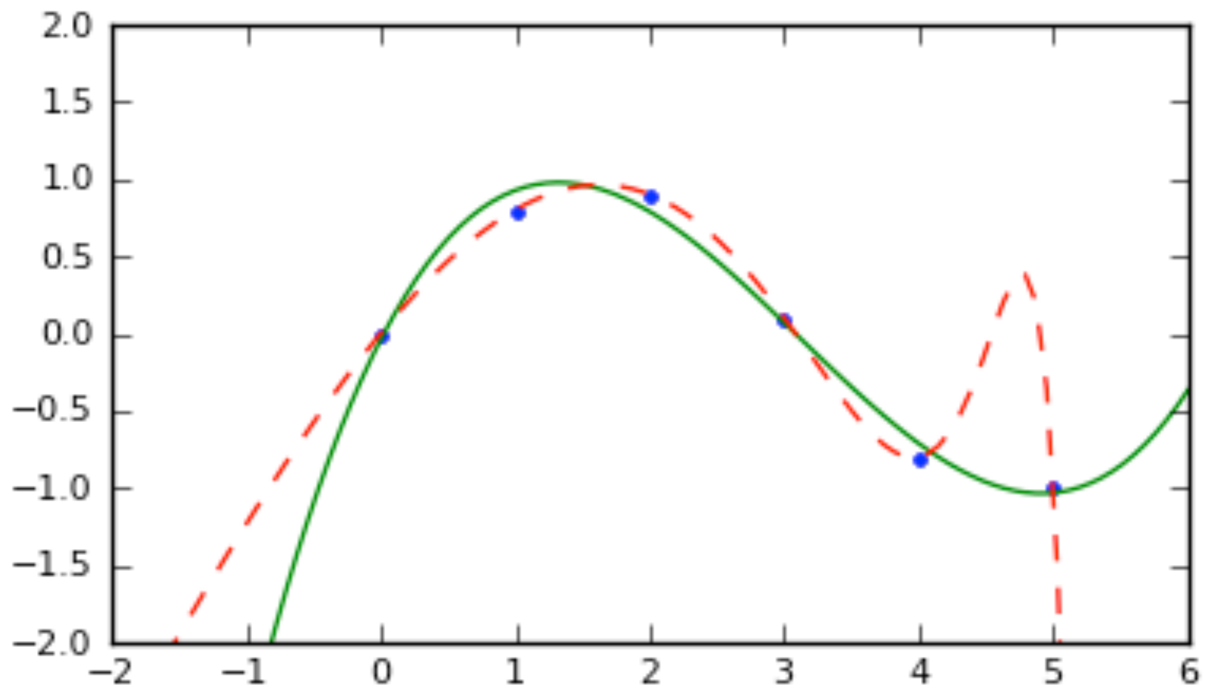
A more realistic plot

```
>>> x = np.linspace(0, 2*np.pi, 300)
>>> y = np.sin(x)
>>> y2 = np.sin(x**2)
>>> plt.plot(x, y, label=r'$\sin(x)$')
[<matplotlib.lines.Line2D object at
0x117572390>]
>>> plt.plot(x, y2, label=r'$\sin(x^2)$')
[<matplotlib.lines.Line2D object at
0x1173b9750>]
>>> plt.title('Some functions')
<matplotlib.text.Text object at 0x103298f50>
>>> plt.xlabel('x')
<matplotlib.text.Text object at 0x1032b00d0>
>>> plt.ylabel('y')
<matplotlib.text.Text object at 0x117573e50>
>>> plt.grid()
>>> plt.legend()
<matplotlib.legend.Legend object at
0x1173bb750>
>>> plt.show()
```



Polynomial Fitting

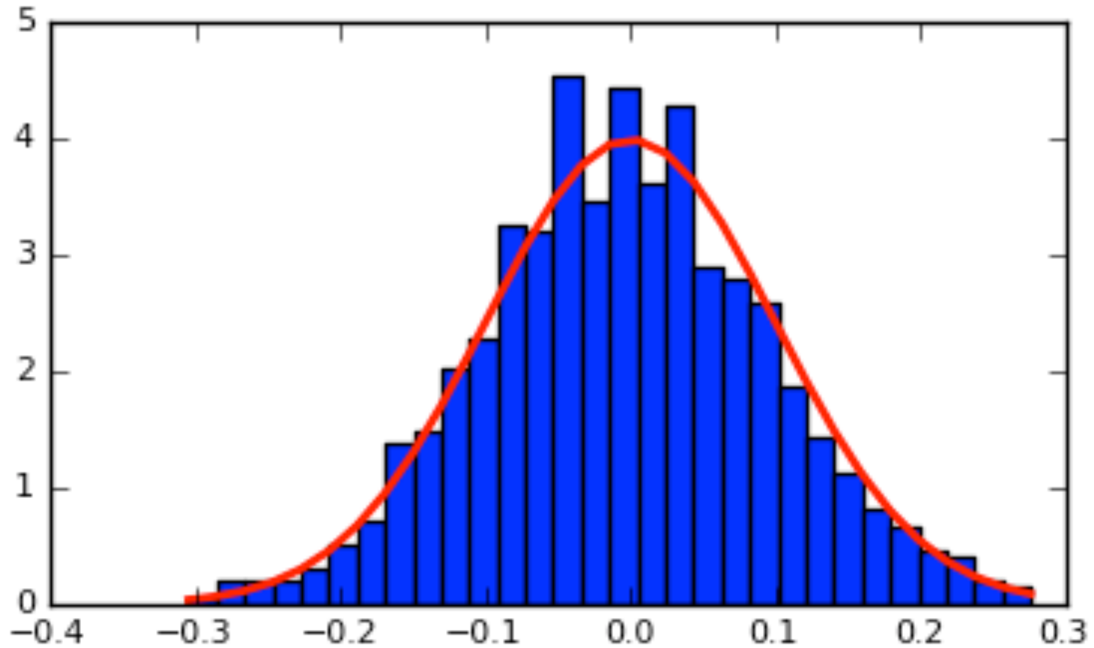
```
>>> x = np.array([0.0, 1.0, 2.0, 3.0,  4.0,  5.0])
>>> y = np.array([0.0, 0.8, 0.9, 0.1, -0.8, -1.0])
>>> z = np.polyfit(x, y, 3)
>>> p = np.poly1d(z)
>>> p30 = np.poly1d(np.polyfit(x, y, 30))
>>> xp = np.linspace(-2, 6, 100)
>>> plt.plot(x, y, '.', xp, p(xp), '-', xp, p30
(xp), '--')
>>> plt.ylim(-2,2)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



Basic (least squares) polynomial fitting can be performed using the *polyfit* routine. More complicated fitting tasks require *scipy*

Random Sampling

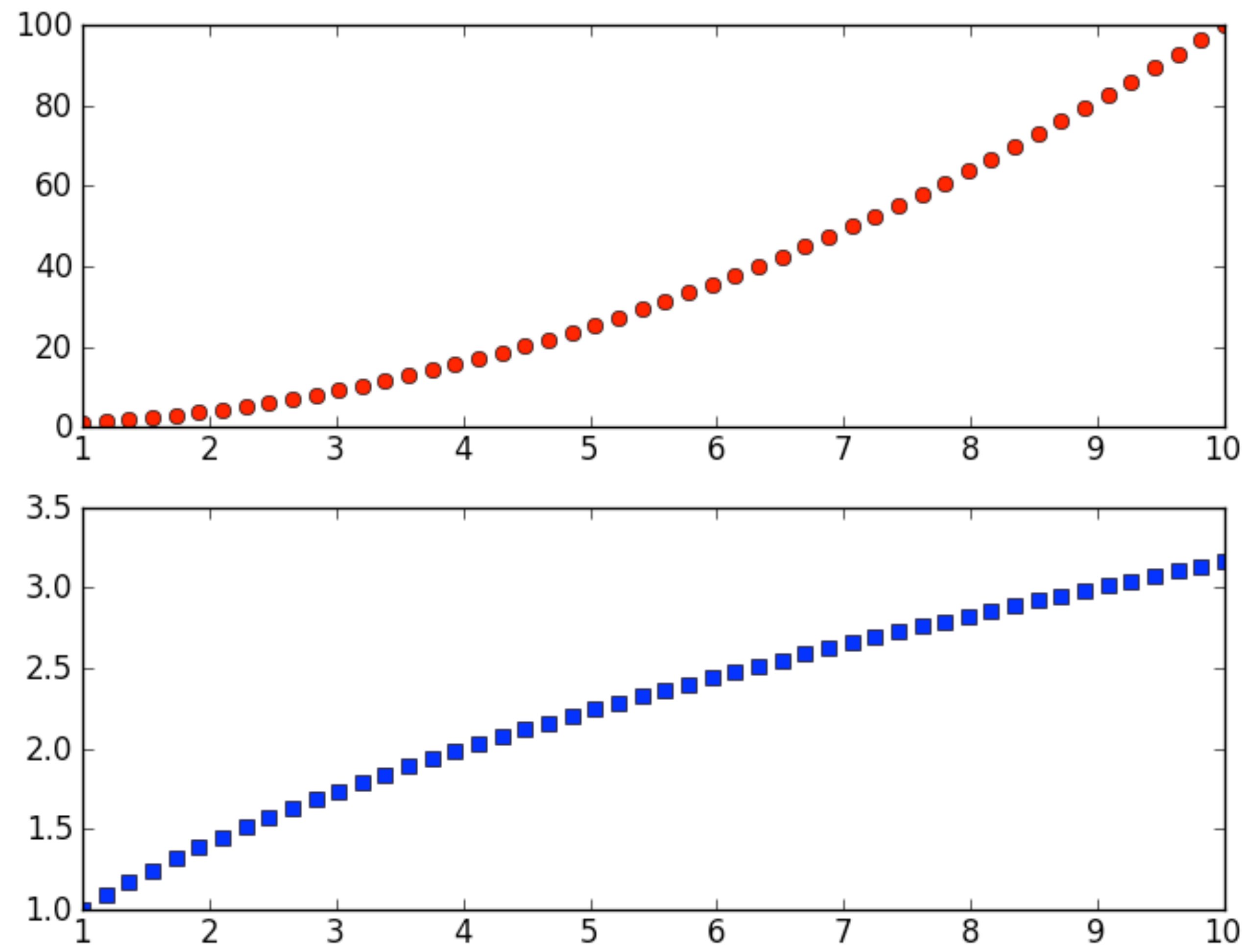
```
>>> mu, sigma = 0, 0.1
>>> s = np.random.normal(mu, sigma, 1000)
>>> count, bins, ignored = plt.hist(s, 30,
                                    normed=True)
>>> plt.plot(bins, 1/(sigma * np.sqrt(2 * np.pi))
             * np.exp( - (bins - mu)**2 /
             (2 * sigma**2) ), color='r')
>>> plt.show()
```



The *numpy.random* module contains the most common probability density distributions, as well as a random number generator

Subplots

```
>>> x = np.linspace(1,10)
>>> y1 = x**2
>>> y2 = np.sqrt(x)
>>> plt.subplot(2,1,1)
<matplotlib.axes.AxesSubplot object at
0x101592a90>
>>> plt.plot(x, y1, "ro")
[<matplotlib.lines.Line2D object at
0x1032ad190>]
>>> plt.subplot(2,1,2)
<matplotlib.axes.AxesSubplot object at
0x10329ca90>
>>> plt.plot(x, y2, "bs")
[<matplotlib.lines.Line2D object at
0x10329c7d0>]
>>> plt.show()
```



Where to go for help

- NumPy Tutorial:
  - http://www.scipy.org/Tentative_NumPy_Tutorial
- NumPy / SciPy documentation:
  - http://docs.scipy.org/doc/
- Matplotlib Tutorial:
  - http://matplotlib.sourceforge.net/users/pyplot_tutorial.html
- Matplotlib Gallery:
  - http://matplotlib.sourceforge.net/gallery.html

Extra Slides

Masked Arrays

```
>>> x = np.array([1, 2, 3, -1, 5])
>>> mx = np.ma.masked_array(x, mask=[0, 0, 0, 1, 0])
>>> mx.data
array([ 1,  2,  3, -1,  5])
>>> mx.mask
array([False, False, False,  True, False], dtype=bool)
>>> mx.mean()
2.75
>>> x = np.ma.array([1, 2, 3])
>>> x[0] = np.ma.masked
>>> x
masked_array(data = [-- 2 3],
             mask = [ True False False],
       fill_value = 999999)
>>> x = np.ma.array([-1, 1, 0, 2, 3], mask=[0, 0, 0, 0, 1])
>>> np.log(x)
masked_array(data = [-- 0.0 -- 0.69314718056 --],
             mask = [ True False  True False  True],
       fill_value = 1e+20)
```

MaskedArrays are a subclass of *ndarray*. In addition to standard *ndarray* properties, they contain an additional Boolean *mask* to indicate invalid data.

Manipulations, Slicing, and Indexing

```
>>> a = arange(12).reshape(3,4)
>>> a
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> i = array( [ [0,1],
...              [1,2] ] )
>>> j = array( [ [2,1],
...              [3,3] ] )
>>> a[i,j]
array([[ 2,  5],
       [ 7, 11]])
>>>
>>> a[i,2]
array([[ 2,  6],
       [ 6, 10]])
>>>
>>> a[:,j]
array([[[ 2,  1],
        [ 3,  3]],
       [[ 6,  5],
        [ 7,  7]],
        [[10,  9],
        [11, 11]]])
```

We can also give indexes for more than one dimension. The arrays of indices for each dimension must have the same shape.