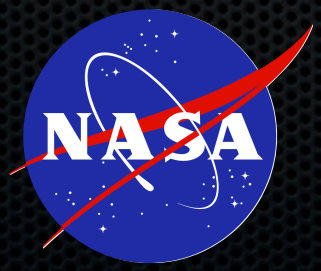
Numpy & Matplotlib



Daniel Kocevski

NASA Goddard Space Flight Center

Numpy & Matplotlib Overview

- NumPy
  - Fundamental package for scientific computing with Python
  - A powerful N-dimensional array object
  - Sophisticated (broadcasting) functions
  - Comparison testing, selection, and manipulation
  - Linear algebra, Fourier transform, and random number capabilities
  - Basic statistics
- Matplotlib
  - Basic 2D (MATLAB like) plotting capabilities

Basic Problem in Python

- What if we want to multiply the contents of a list by some value? Use list comprehension
- This is a very inefficient operation
- Not nearly as fast as compiled C code
- Numpy array fixes this problem

```
>>> list = [2,2]
>>> list * 2
[2, 2, 2, 2]

>>> [i*2 for i in list]
[4,4]
```

ndarray class

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

Instantiating ndarrays

- The array method takes a list and returns an ndarray
- Several methods to return filled arrays
  - ones(), zeros(), linspace(), arange()
- Object types are ‘numpy.ndarray’

```
>>> 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 properties

```
>>> a = np.array([1, 2, 3])
>>> a.dtype
dtype('float64')
>>> a * 2.0
array([ 2.,  4.,  6.])
>>> 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])
```

- Arrays have fixed-size items
  - All ‘float64’, ‘|S1’, ‘int32’, etc
- New data of a different type will get converted to match the array’s data type

Reading/Writing Arrays

- Arrays can be directly read from and written to files
  - Modules exist for csv, fits, jpg, etc

```
% 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")

% less data.out1
"data.out1" may be a binary file.  See
it anyway?
% less data.out2
5.000000,2.000000,3.000000,4.000000
data.out2 (END)
```

Manipulations, Slicing, 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[::2]
array([0, 2, 4, 6, 8])
>>> a[:6:2]
array([0, 2, 4])
>>> a[:6:2] = 0
>>> a
array([0, 1, 0, 3, 0, 5, 6, 7, 8, 9])
>>> a[-2]
8
>>> a[::-1]
array([9, 8, 7, 6, 5, 0, 3, 0, 1, 0])
>>> a[2:-2]
array([0, 1, 0, 3, 0, 5, 6, 7])
```

- Array objects can be indexed, sliced, and iterated over, much like lists
- ::n indexes every nth element

Universal Functions

- 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.
- 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]])
>>> a * b
array([[ 2,  6],[12, 20]])
>>> 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 Efficiency

- Universal functions run much faster than for loops
- Note the “timeit” function (as written) requires ipython

```
>>> 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
```

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 arrays of different shapes.
- The smaller array is broadcast across the larger array so that they have compatible shapes

Comparison Testing and Selection

- Arrays can be compared on an element-by-element basis
- Result is an array of bool (True/False) values

```
>>> 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)
```

Searching and Indexing

```
>>> a = np.array([1, 3, 0, -5, 0], float)
>>> np.where(a != 0)
(array([0, 1, 3]),)
>>> a[a != 0]
array([ 1.,  3., -5.])
>>> len(a[np.where(a > 0)])
2
>>> 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]))

>>> b = np.array([10,20,30,40,50]
>>> i = np.where(a > 0)
>>> b[i]
array([ 10., 20., 40.])
```

- where provides a fast way to search (and extract) individual elements of an ndarray
- See also nonzero

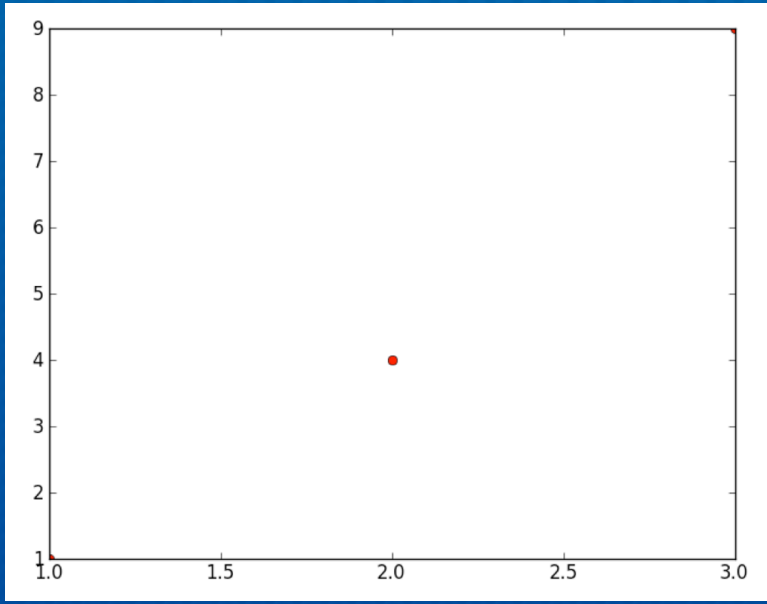
Basic Statistics

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

```
>>> 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
```

Matplotlib Basics - Points

```
>>> 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()
```

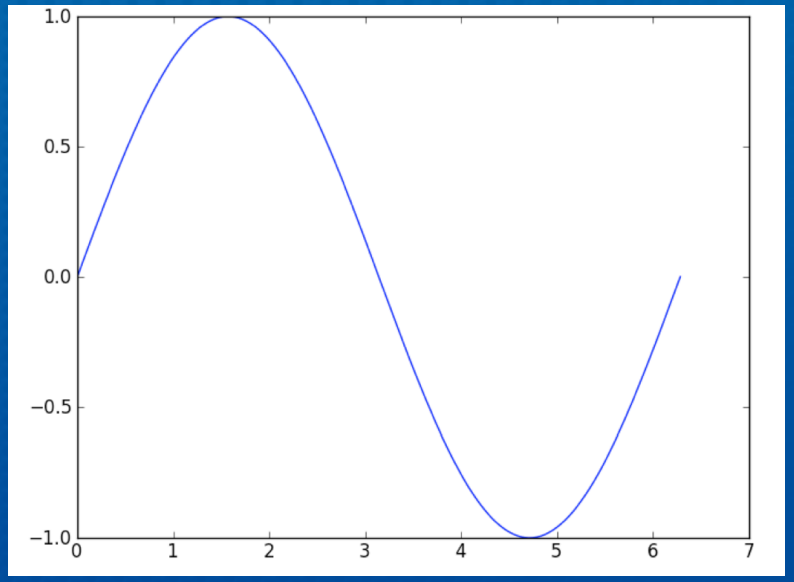


- The matplotlib module provides publication quality figures
- MATLAB-like syntax
- Specifying a plotting symbol will plot individual data points

matplotlib Basics - Lines

- Not specifying a plotting symbol will connect a line between the data points

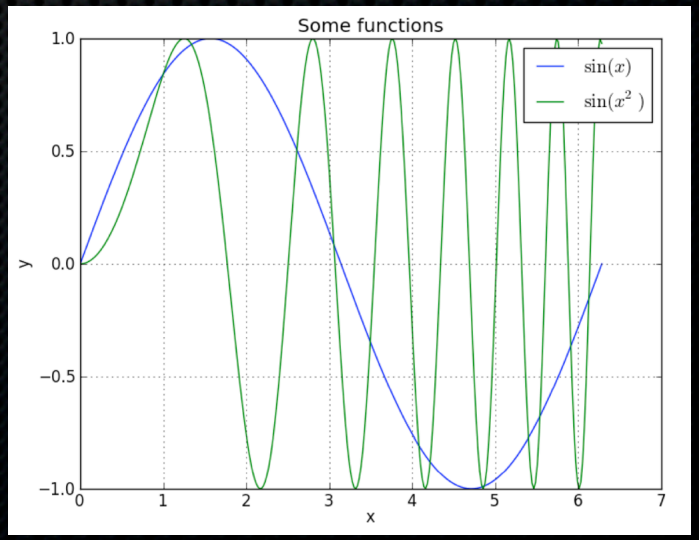
```
>>> import matplotlib.pylab as plt
>>> x = np.linspace(0, 2*np.pi, 300)
>>> y = np.sin(x)
>>> plt.plot(x, y)
[<matplotlib.lines.Line2D object at
0x1173aead0>]
>>> plt.show()
```



matplotlib Basics - Overplotting

```
>>> import matplotlib.pylab as plt
>>> 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()
```

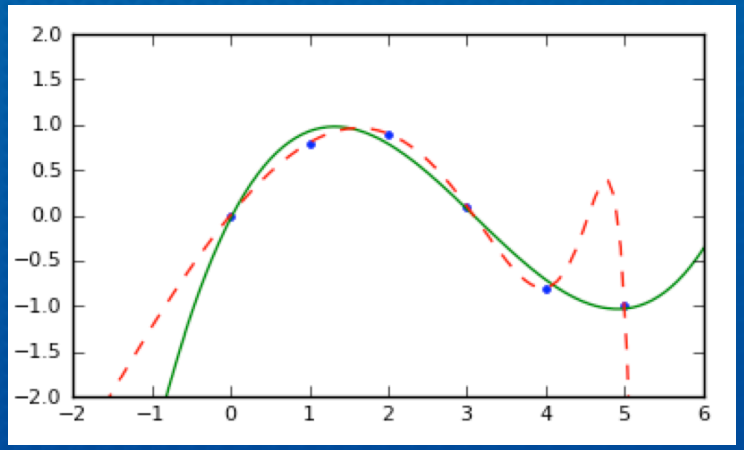
- Issuing the plot routine more than once will overplot data
- Host of other methods to control labels, grid, legend, etc...



Polynomial Fitting

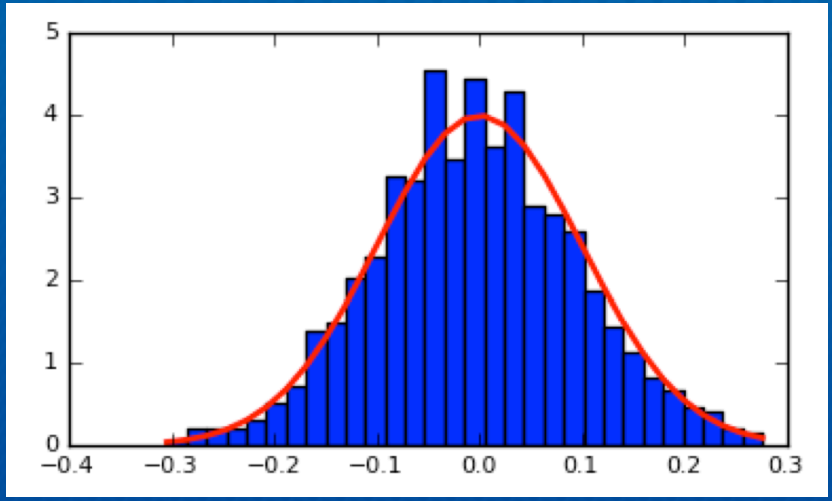
- Basic (least squares) polynomial fitting can be performed using the polyfit routine
- More complicated fitting tasks require scipy

```
>>> 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()
```



Random Sampling

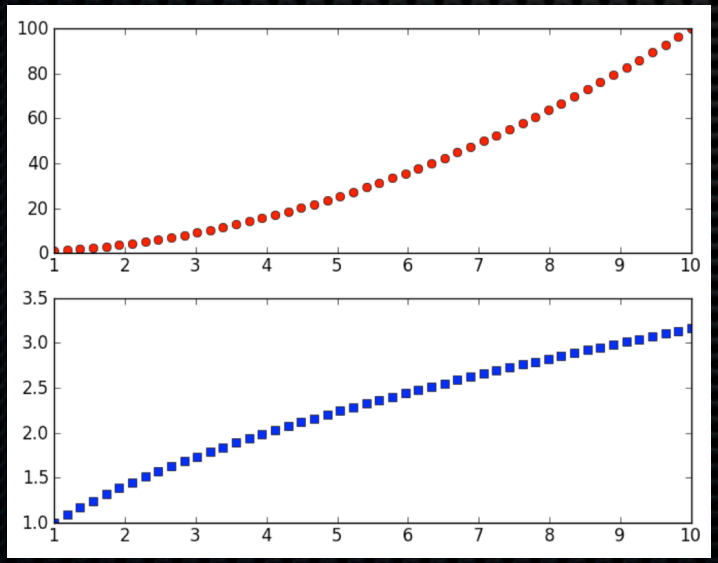
```
>>> mu, sigma = 0, 0.1
>>> s = np.random.normal(mu, sigma, 1000)
>>> count, bins, ignored = plt.hist(s, 30,
normed=True)
>>> plt.plot(bins, 1/(sigma * np.sqrt(2 *
np.pi)) * np.exp( - (bins - mu)**2 /(2 *
sigma**2) ), color='r')
>>> plt.show()
```



- The numpy.random module contains the most common probability density distributions
- Also includes a random number generator

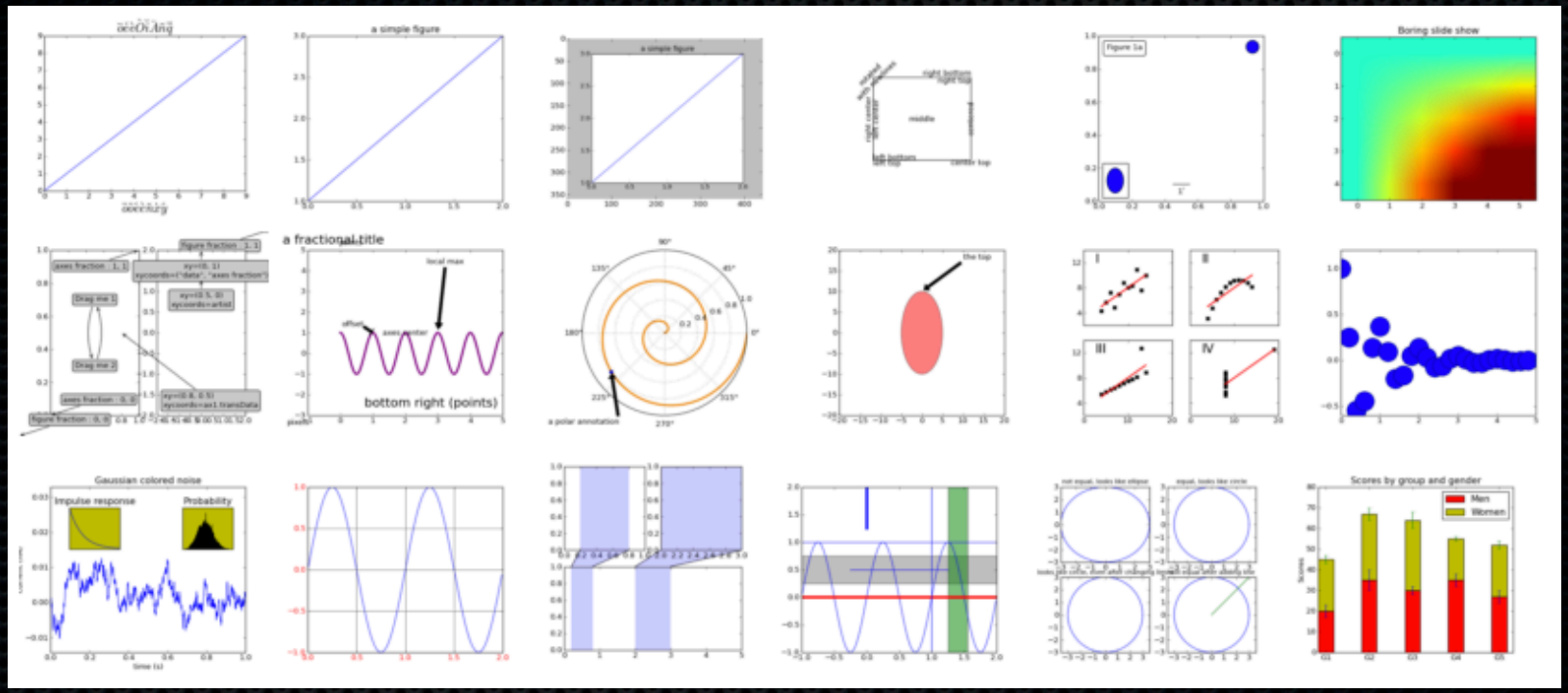
Subplots

- Subplot routine allows for multiple plots on a single canvas
- subplot(nrows, ncols, plot_number)




```
>>> 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()
```

Matplotlib Gallery



- Plots are easily scriptable
- Plenty of example scripts online to learn from

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.org/gallery.html