What is NumPy?

- Python's attractive features:
  - Easy to extend
  - Great syntax which encourages easy to write and maintain code
  - Incredibly large standard-library and third-party tools
- No built-in multi-dimensional array (but it supports the needed syntax for extracting elements from one)
- NumPy provides a **fast** built-in object (ndarray) which is a multi-dimensional array of a homogeneous data-type.

Overview

- NumPy and SciPy are open-source add-on modules to Python
- They provide common mathematical and numerical routines in pre-compiled, fast functions
- Provide functionality that meets, or perhaps exceeds, that associated with common commercial software like MatLab
- The NumPy (Numeric Python) package provides basic routines for manipulating large arrays and matrices of numeric data
- The SciPy (Scientific Python) package extends the functionality of NumPy with a substantial collection of useful algorithms, like minimization, Fourier transformation, regression, and other applied mathematical techniques

NumPy

- Offers Matlab-ish capabilities within Python
- NumPy replaces Numeric and Numarray
- Initially developed by Travis Oliphant (building on the work of dozens of others)
- Over 30 sub-version “committers” to the project
- NumPy 1.0 released October, 2006
- ~25K downloads/month from Sourceforge.
- *Included with some Python distributions like Anaconda*

NumPy

- The NumPy and SciPy development community maintains an extensive online documentation system, including user guides and tutorials, at: https://docs.scipy.org/doc/

N-D ARRAY (NDARRAY)

- N-dimensional array of rectangular data
- Element of the array can be C-structure or simple data-type.
- Fast algorithms on machine data-types (int, float, etc.)

UNIVERSAL FUNCTIONS

- Functions that operate element-by-element and return result
- Fast-loops registered for each fundamental data-type
- $\sin(\mathbf{x}) = [\sin(x_i)\ i=0..N]$
- $\mathbf{x}+\mathbf{y} = [x_i + y_i\ i=0..N]$

Importing the NumPy module

- The standard approach is to use a simple import statement:

    >>> import numpy

- However, for large amounts of calls to NumPy functions, it can become tedious to write numpy.X over and over again. Instead, it is common to import under the briefer name np:

    >>> import numpy as np

- This statement will allow us to access NumPy objects using np.X instead of numpy.X

Importing the NumPy module

- It is also possible to import NumPy directly into the current namespace
  - We don't have to use dot notation at all, but rather simply call the functions as if they were built-in

```
>>> from numpy import *
```

- This strategy is usually frowned upon in Python programming because it starts to remove some of the nice organization that modules provide.

NumPy Array

- The central feature of NumPy is the *array* object class
- Arrays are similar to lists in Python, except that every element of an array must be of the same type, typically a numeric type like float or int
- A NumPy array is an N-dimensional homogeneous collection of “items” of the same “kind”
    - The kind can be any arbitrary structure and is specified using the data-type

NumPy Array

- Arrays make operations with large amounts of numeric data very fast and are generally much more efficient than lists
- An array can be created from a list:

```
>>> a = np.array([1, 4, 5, 8], float)
>>> a
array([ 1.,  4.,  5.,  8.])
>>> type(a)
<type 'numpy.ndarray'>
```

- Here, the function array takes two arguments: the list to be converted into the array and the type of each member of the list
- Array elements are accessed, sliced, and manipulated just like lists:

```
>>> a[:2]
array([ 1.,  4.])
>>> a[3]
8.0
>>> a[0] = 5.
>>> a
array([ 5.,  4.,  5.,  8.])
```

NumPy Array

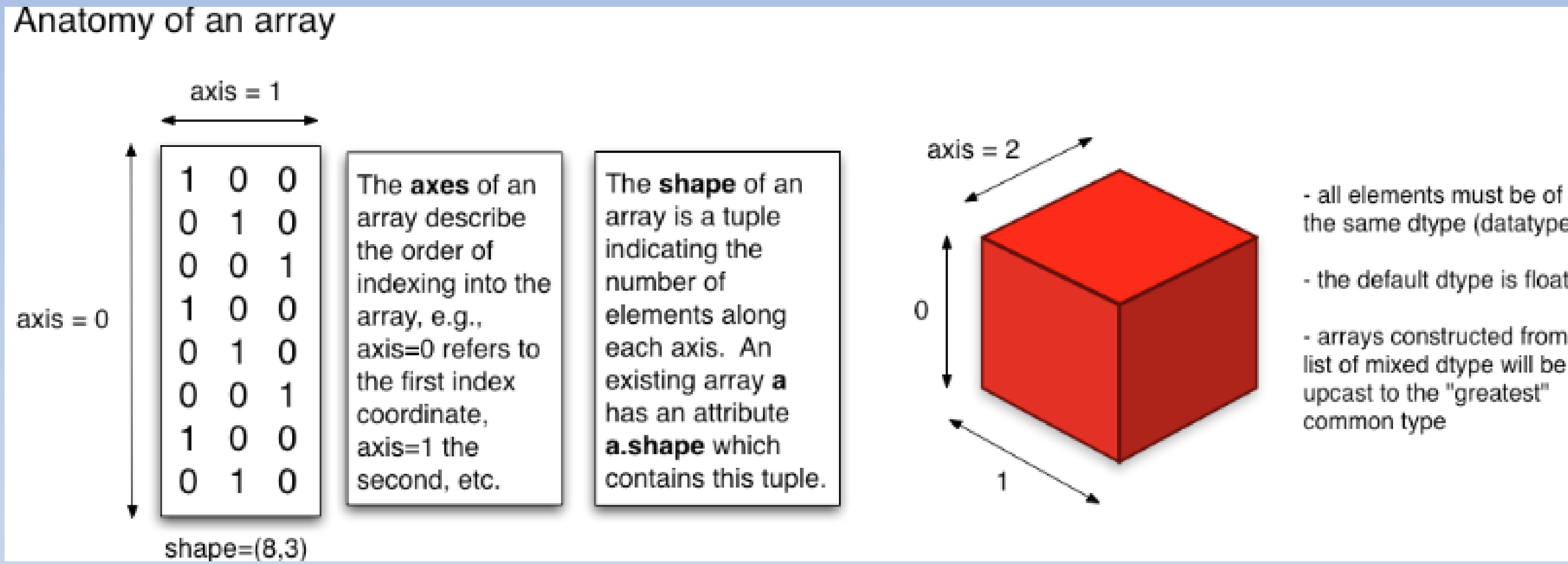
- Arrays can be multidimensional
- Here is an example with a two-dimensional array (e.g., a matrix):

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]], float)
>>> a
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])
>>> a[0,0]
1.0
>>> a[0,1]
2.0
```

- Array slicing works with multiple dimensions in the same way as usual, applying each slice specification as a filter to a specified dimension. Use of a single ":" in a dimension indicates the use of everything along that dimension

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]], float)
>>> a[1,:]
array([ 4.,  5.,  6.])
>>> a[:,2]
array([ 3.,  6.])
>>> a[-1:,-2:]
array([[ 5.,  6.]])
```

NumPy Array


Anatomy of an array
axis = 1
axis = 0
1 0 0
0 1 0
0 0 1
1 0 0
0 1 0
0 0 1
1 0 0
0 1 0
shape=(8,3)
The **axes** of an array describe the order of indexing into the array, e.g., axis=0 refers to the first index coordinate, axis=1 the second, etc.
The **shape** of an array is a tuple indicating the number of elements along each axis. An existing array **a** has an attribute **a.shape** which contains this tuple.
axis = 2
0
1
- all elements must be of the same dtype (datatype)
- the default dtype is float
- arrays constructed from list of mixed dtype will be upcast to the "greatest" common type

NumPy Array

- The shape property of an array returns a tuple with the size of each array dimension:

```
>>> a.shape
(2, 3)
```

- The dtype property tells you what type of values are stored by the array:

```
>>> a.dtype
dtype('float64')
```

- Here, float64 is a numeric type that NumPy uses to store double-precision (8-byte) real numbers, similar to the float type in Python.
- When used with an array, the len function returns the length of the first axis:

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]], float)
>>> len(a)
2
```

NumPy Array

- The **in** operator can be used to test if values are present in an array:

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]], float)
>>> 2 in a
True
>>> 0 in a
False
```

- Arrays can be reshaped using tuples that specify new dimensions (creates a new array, does not change the original one)
  - In the following example, we turn a ten-element one-dimensional array into a two-dimensional one whose first axis has five elements

```
>>> a = np.array(range(10), float)
>>> a
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.])
>>> a = a.reshape((5, 2))
>>> a
array([[ 0.,  1.],
       [ 2.,  3.],
       [ 4.,  5.],
       [ 6.,  7.],
       [ 8.,  9.]])
>>> a.shape
(5, 2)
```

NumPy Array

- Python's name-binding approach still applies to arrays
  - The copy function can be used to create a new, separate copy of an array in memory if needed:

```
>>> a = np.array([1, 2, 3], float)
>>> b = a
>>> c = a.copy()
>>> a[0] = 0
>>> a
array([0., 2., 3.])
>>> b
array([0., 2., 3.])
>>> c
array([1., 2., 3.])
```

- Lists can also be created from arrays:

```
>>> a = np.array([1, 2, 3], float)
>>> a.tolist()
[1.0, 2.0, 3.0]
>>> list(a)
[1.0, 2.0, 3.0]
```

NumPy Array

- You can convert the raw data in an array to a binary string (i.e., not in human-readable form) using the tostring() function
- The fromstring() function then allows an array to be created from this data later on
- These routines are sometimes convenient for saving large amount of array data in files that can be read later on:

```
>>> a = array([1, 2, 3], float)
>>> s = a.tostring()
>>> s
'\x00\x00\x00\x00\x00\x00\xf0?\x00\x00\x00\x00\x00\x00\x00@\x00\x00\x00\x00
\x00\x00\x08@'
>>> np.fromstring(s)
array([ 1.,  2.,  3.])
```

NumPy Array

- You can fill an array with a single value:

```
>>> a = array([1, 2, 3], float)
>>> a
array([ 1.,  2.,  3.])
>>> a.fill(0)
>>> a
array([ 0.,  0.,  0.])
```

- Transposed versions of arrays can also be generated, which will create a new array with the final two axes switched:

```
>>> a = np.array(range(6), float).reshape((2, 3))
>>> a
array([[ 0.,  1.,  2.],
       [ 3.,  4.,  5.]])
>>> a.transpose()
array([[ 0.,  3.],
       [ 1.,  4.],
       [ 2.,  5.]])
```

NumPy Array

- One dimensional versions of multi-dimensional arrays can be created using *flatten*()

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]], float)
>>> a
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])
>>> a.flatten()
array([ 1.,  2.,  3.,  4.,  5.,  6.])
```

- Two or more arrays can be concatenated together using the *concatenate* function with a tuple of the arrays to be joined:

```
>>> a = np.array([1,2], float)
>>> b = np.array([3,4,5,6], float)
>>> c = np.array([7,8,9], float)
>>> np.concatenate((a, b, c))
array([1., 2., 3., 4., 5., 6., 7., 8., 9.])
```

NumPy Array

- If an array has more than one dimension, it is possible to specify the axis along which multiple arrays are concatenated
- By default (without specifying the axis), NumPy concatenates along the first dimension:

```
>>> a = np.array([[1, 2], [3, 4]], float)
>>> b = np.array([[5, 6], [7,8]], float)
>>> np.concatenate((a,b))
array([[ 1.,  2.],
       [ 3.,  4.],
       [ 5.,  6.],
       [ 7.,  8.]])
>>> np.concatenate((a,b), axis=0)
array([[ 1.,  2.],
       [ 3.,  4.],
       [ 5.,  6.],
       [ 7.,  8.]])
>>> np.concatenate((a,b), axis=1)
array([[ 1.,  2.,  5.,  6.],
       [ 3.,  4.,  7.,  8.]])
```

Creating Arrays

- Other ways to create arrays
  - The *arange* function is similar to the range function but returns an array:

```
>>> np.arange(5, dtype=float)
array([ 0.,  1.,  2.,  3.,  4.])
>>> np.arange(1, 6, 2, dtype=int)
array([1, 3, 5])
```

  - The functions *zeros* and *ones* create new arrays of specified dimensions filled with these values. These are perhaps the most commonly used functions to create new arrays:

```
>>> np.ones((2,3), dtype=float)
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
>>> np.zeros(7, dtype=int)
array([0, 0, 0, 0, 0, 0, 0])
```

  - The *zeros_like* and *ones_like* functions create a new array with the same dimensions and type of an existing one:

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]], float)
>>> np.zeros_like(a)
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
>>> np.ones_like(a)
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
```

Creating Arrays

- There are also a number of functions for creating special matrices (2D arrays). To create an identity matrix of a given size:

```
>>> np.identity(4, dtype=float)
```

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

- The eye function returns matrices with ones along the kth diagonal:

```
>>> np.eye(4, k=1, dtype=float)
array([[ 0.,  1.,  0.,  0.],
       [ 0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  1.],
       [ 0.,  0.,  0.,  0.]])
```

Array Mathematics

- When standard mathematical operations are used with arrays, they are applied on an element-by-element basis (array sizes must be the same):

```
>>> a = np.array([1,2,3], float)
>>> b = np.array([5,2,6], float)
>>> a + b
array([6., 4., 9.])
>>> a – b
array([-4., 0., -3.])
>>> a * b
array([5., 4., 18.])
>>> b / a
array([5., 1., 2.])
>>> a % b
array([1., 0., 3.])
>>> b**a
array([5., 4., 216.])
```

Array Mathematics

- For two-dimensional arrays, multiplication remains elementwise and does *not* correspond to matrix multiplication (special methods for matrix operations):

```
>>> a = np.array([[1,2], [3,4]], float)
>>> b = np.array([[2,0], [1,3]], float)
>>> a * b
array([[2., 0.], [3., 12.]])
```

- Errors are thrown if arrays do not match in size:

```
>>> a = np.array([1,2,3], float)
>>> b = np.array([4,5], float)
>>> a + b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: shape mismatch: objects cannot be broadcast to a single shape
```

Array Broadcasting

- Arrays that do not match in the number of dimensions will be *broadcasted* by Python to perform mathematical operations
- Often the smaller array will be repeated as necessary to perform the operation indicated

```
>>> a = np.array([[1, 2], [3, 4], [5, 6]], float)
>>> b = np.array([-1, 3], float)
>>> a
array([[ 1.,  2.],
       [ 3.,  4.],
       [ 5.,  6.]])
>>> b
array([-1.,  3.])
>>> a + b
array([[ 0.,  5.],
       [ 2.,  7.],
       [ 4.,  9.]])
```

- b was repeated as if it were:

```
array([[-1.,  3.],
       [-1.,  3.],
       [-1.,  3.]])
```

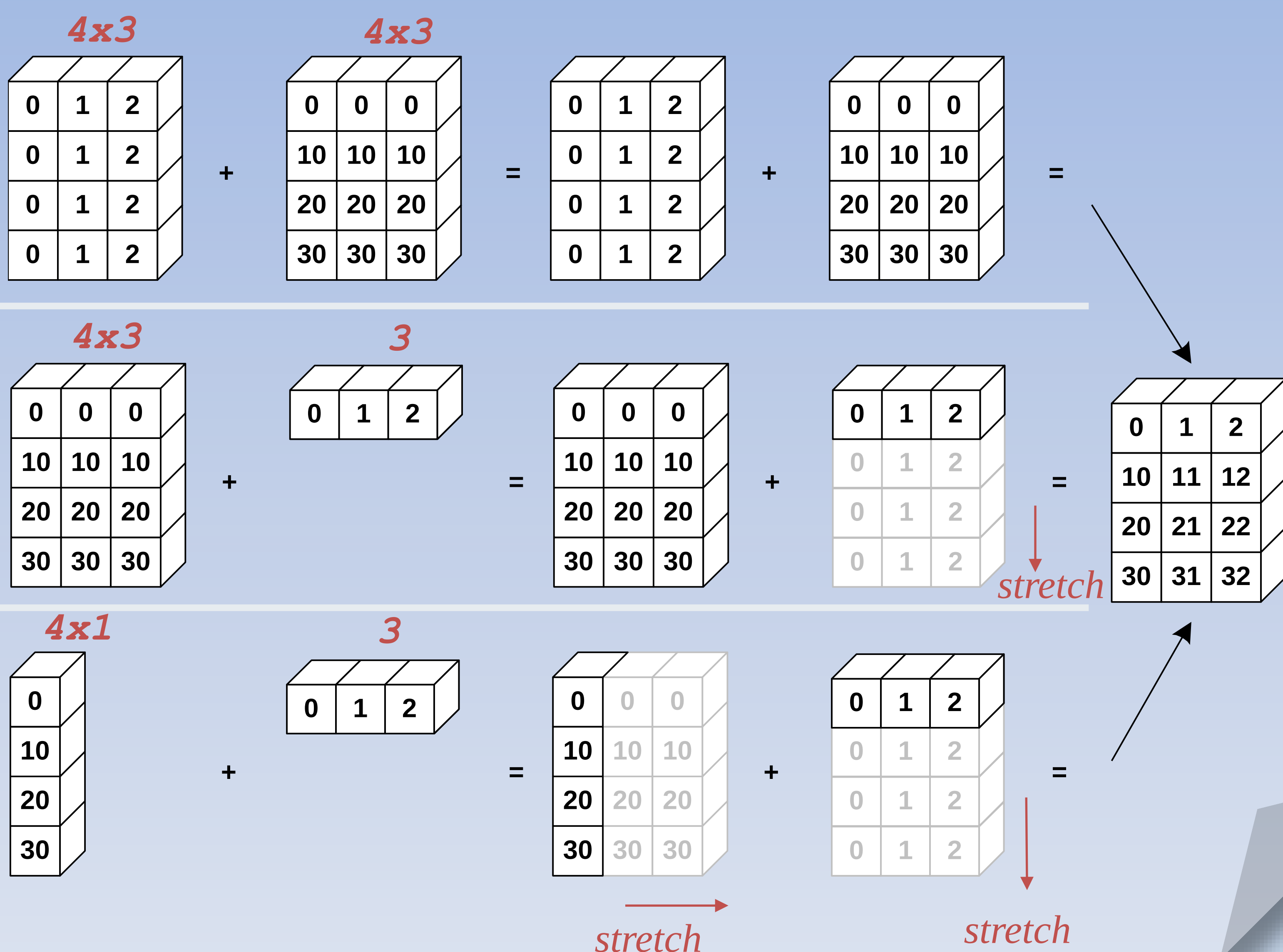
Array Broadcasting

- If you want to remove the ambiguity, specify the *newaxis* constant

```
>>> a = np.zeros((2,2), float)
>>> b = np.array([-1., 3.], float)
>>> a
array([[ 0.,  0.],
       [ 0.,  0.]])
>>> b
array([-1., 3.])
>>> a + b
array([[-1.,  3.],
       [-1.,  3.]])
>>> a + b[np.newaxis,:]
array([[-1.,  3.],
       [-1.,  3.]])
>>> a + b[:,np.newaxis]
array([[-1., -1.],
       [ 3.,  3.]])
```

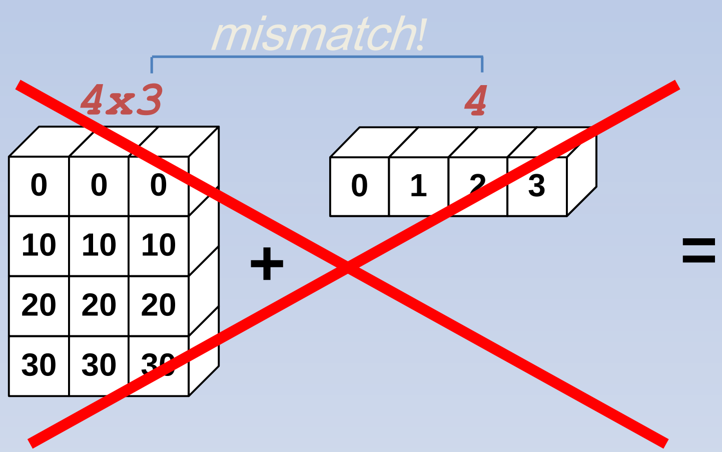
```
b=np.array([-1,3])

b
array([-1,  3])

b[np.newaxis,:]
array([[-1,  3]])

b[:,np.newaxis]
array([[-1],
       [ 3]])
```

Array Broadcasting



Broadcasting Rules

- The *trailing* axes of both arrays must either be 1 or have the same size for broadcasting to occur. Otherwise, a "ValueError: frames are not aligned" exception is thrown.



Math Functions

- NumPy offers a large library of common mathematical functions that can be applied elementwise to arrays
- Some of them are:

  abs, sign, sqrt, log, log10, exp, sin, cos, tan, arcsin, arccos, arctan, sinh, cosh, tanh, arcsinh, arccosh, and arctanh

```
>>> a = np.array([1, 4, 9], float)
```

```
>>> np.sqrt(a)
array([ 1.,  2.,  3.])
```

- The functions floor, ceil, and rint give the lower, upper, or nearest (rounded) integer:

```
>>> a = np.array([1.1, 1.5, 1.9], float)
>>> np.floor(a)
array([ 1.,  1.,  1.])
>>> np.ceil(a)
array([ 2.,  2.,  2.])
>>> np.rint(a)
array([ 1.,  2.,  2.])
```

Array Iteration

```
>>> a = np.array([1, 4, 5], int)
>>> for x in a:
...   print x
... <hit return>
1
4
5
```

```
>>> a = np.array([[1, 2], [3, 4], [5, 6]], float)
>>> for x in a:
...   print x
... <hit return>
[ 1.  2.]
[ 3.  4.]
[ 5.  6.]
```

```
>>> a = np.array([[1, 2], [3, 4], [5, 6]], float)
>>> for (x, y) in a:
...   print x * y
...  <hit return>
2.0
12.0
30.0
```

Basic *whole* array operations

- Sum and Product using member functions of the array:

```
>>> a = np.array([2, 4, 3], float)
>>> a.sum()
9.0
>>> a.prod()
24.0
```

- Alternatively, standalone functions in the NumPy module can be accessed:

```
>>> np.sum(a)
9.0
>>> np.prod(a)
24.0
```

- Computation of statistical quantities in datasets:
  Mean, Variance, Standard Deviation

```
>>> a = np.array([2, 1, 9], float)
>>> a.mean()
4.0
>>> a.var()
12.666666666666666
>>> a.std()
3.5590260840104371
```

Basic *whole* array operations

- Min and Max values:

```
>>> a = np.array([2, 1, 9], float)
>>> a.min()
1.0
>>> a.max()
9.0
```

- The argmin and argmax functions return the array indices of the minimum and maximum values:

```
>>> a = np.array([2, 1, 9], float)
>>> a.argmin()
1
>>> a.argmax()
2
```

Basic *whole* array operations

- For multidimensional arrays:
  - An optional argument axis that will perform an operation along only the specified axis (i.e. look at the axis specified and apply the function)
  - Results are placed in a return array

```
>>> a = np.array([[0, 2], [3, -1], [3, 5]], float)
>>> a.mean(axis=0)
array([ 2.,  2.])
>>> a.mean(axis=1)
array([ 1.,  1.,  4.])
>>> a.min(axis=1)
array([ 0., -1.,  3.])
>>> a.max(axis=0)
array([ 3.,  5.])
```

```
[[ 0.  2.]
 [ 3. -1.]
 [ 3.  5.]]
```

Basic *whole* array operations

- Arrays can be sorted:

```
>>> a = np.array([6, 2, 5, -1, 0], float)
>>> sorted(a)
[-1.0, 0.0, 2.0, 5.0, 6.0]
>>> a.sort()
>>> a
array([-1.,  0.,  2.,  5.,  6.])
```

- Values in an array can be "clipped" to be within a pre-specified range *(clip(min,max))*

```
>>> a = np.array([6, 2, 5, -1, 0], float)
>>> a.clip(0, 5)
array([ 5.,  2.,  5.,  0.,  0.])
```

- Unique elements can be extracted:

```
>>> a = np.array([1, 1, 4, 5, 5, 5, 7], float)
>>> np.unique(a)
array([ 1.,  4.,  5.,  7.])
```

- For 2-D arrays, the diagonal can be extracted:

```
>>> a = np.array([[1, 2], [3, 4]], float)
>>> a.diagonal()
array([ 1.,  4.])
```

Comparison Operators

- Boolean comparisons can be used to compare members elementwise on arrays of equal size.
- The return value is an array of Boolean 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)
>>> a <= b
array([False,  True,  True], dtype=bool)
```

- The results of a Boolean comparison can be stored in an array:

```
>>> c = a > b
>>> c
array([ True, False, False], dtype=bool)
```

Comparison Operators

- Arrays can be compared to single values using broadcasting:

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

- The any and all operators can be used to determine whether or not any or all elements of a Boolean array are true:

```
>>> c = np.array([ True, False, False], bool)
>>> any(c)
True
>>> all(c)
False
```

- Compound Boolean expressions can be applied to arrays on an element-by-element basis using special functions logical_and, logical_or, and logical_not:

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

Value Testing

- The nonzero function gives a tuple of indices of the nonzero values in an array
- The number of items in the tuple equals the number of axes of the array:

```
>>> a = np.array([[0, 1], [3, 0]], float)
>>> a.nonzero()
(array([0, 1]), array([1, 0]))
```

- It is also possible to test whether or not values are NaN ("not a number") or finite:

```
>>> a = np.array([1, np.NaN, np.Inf], float)
>>> a
array([  1.,   NaN,   Inf])
>>> np.isnan(a)
array([False,  True, False], dtype=bool)
>>> np.isfinite(a)
array([ True, False, False], dtype=bool)
```

- *Although here we used NumPy constants to add the NaN and infinite values, these can result from standard mathematical operations*

Vectors and Matrices

- To performing standard vector and matrix multiplication, use the dot product function *dot*
- With vectors:

```
>>> a = np.array([1, 2, 3], float)
>>> b = np.array([0, 1, 1], float)
>>> np.dot(a, b)
5.0
```

- You can use the same function for matrix multiplication:

```
>>> a = np.array([[0, 1], [2, 3]], float)
>>> b = np.array([2, 3], float)
>>> c = np.array([[1, 1], [4, 0]], float)
>>> a
array([[ 0.,  1.],
       [ 2.,  3.]])
>>> np.dot(b, a)
array([  6.,  11.])
>>> np.dot(a, b)
array([  3.,  13.])
>>> np.dot(a, c)
array([[  4.,   0.],
       [ 14.,   2.]])
>>> np.dot(c, a)
array([[ 2.,  4.],
       [ 0.,  4.]])
```