This statement will allow us to access NumPy objects using `np.X` instead of `numpy.X`. It is also possible to import NumPy directly into the current namespace so that 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 *
```

However, this strategy is usually frowned upon in Python programming because it starts to remove some of the nice organization that modules provide. For the remainder of this tutorial, we will assume that the `import numpy as np` has been used.

## Arrays

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`. 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.])
```

Arrays can be multidimensional. Unlike lists, different axes are accessed using commas inside bracket notation. 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.]])
```

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
```

The `in` statement 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. In the following example, we turn a ten-element one-dimensional array into a two-dimensional one whose first axis has five elements and whose second axis has two 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)
```

Notice that the `reshape` function creates a new array and does not itself modify the original array.

Keep in mind that 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]
```

One 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.])
```

One 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.])
```

```
>>> a = np.array([1, 2, 3], float)
>>> a
array([1., 2., 3.])
>>> a[:,np.newaxis]
array([[ 1.],
       [ 2.],
       [ 3.]])
>>> a[:,np.newaxis].shape
(3,1)
>>> b[np.newaxis,:]
array([[ 1.,  2.,  3.]])
>>> b[np.newaxis,:].shape
(1,3)
```

Notice here that in each case the new array has two dimensions; the one created by `newaxis` has a length of one. The `newaxis` approach is convenient for generating the proper-dimensioned arrays for vector and matrix mathematics.

## 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.]])
```

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. This means that the arrays should be the same size during addition, subtraction, etc.:

```
>>> 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.])
```

For two-dimensional arrays, multiplication remains elementwise and does *not* correspond to matrix multiplication. There are special functions for matrix math that we will cover later.

```
>>> 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
```

However, arrays that do not match in the number of dimensions will be *broadcasted* by Python to perform mathematical operations. This often means that the smaller array will be repeated as necessary to perform the operation indicated. Consider the following:

```
>>> 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.]])
```

Here, the one-dimensional array `b` was broadcasted to a two-dimensional array that matched the size of `a`. In essence, `b` was repeated for each item in `a`, as if it were given by

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

Python automatically broadcasts arrays in this manner. Sometimes, however, how we should broadcast is ambiguous. In these cases, we can use the `newaxis` constant to specify how we want to broadcast:

```
>>> 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.]])
```

In addition to the standard operators, NumPy offers a large library of common mathematical functions that can be applied elementwise to arrays. Among these are the functions: `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.])
```

Also included in the NumPy module are two important mathematical constants:

```
>>> np.pi
3.1415926535897931
>>> np.e
2.7182818284590451
```

## Array iteration

It is possible to iterate over arrays in a manner similar to that of lists:

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

For multidimensional arrays, iteration proceeds over the first axis such that each loop returns a subsection of the array:

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

Multiple assignment can also be used with array iteration:

```
>>> 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 array operations

Many functions exist for extracting whole-array properties. The items in an array can be summed or multiplied:

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

In this example, member functions of the arrays were used. Alternatively, standalone functions in the NumPy module can be accessed:

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

For most of the routines described below, both standalone and member functions are available.

A number of routines enable computation of statistical quantities in array datasets, such as the mean (average), variance, and standard deviation:

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

It's also possible to find the minimum and maximum element 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
```

For multidimensional arrays, each of the functions thus far described can take an optional argument `axis` that will perform an operation along only the specified axis, placing the results 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.])
```

Like lists, 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 prespecified range. This is the same as applying `min(max(x, minval), maxval)` to each element `x` in an array.

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

Unique elements can be extracted from an array:

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

For two dimensional arrays, the diagonal can be extracted:

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

## Comparison operators and value testing

Boolean comparisons can be used to compare members elementwise on arrays of equal size. The return value is an array of Boolean `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)
>>> 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)
```

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)
```

The `where` function forms a new array from two arrays of equivalent size using a Boolean filter to choose between elements of the two. Its basic syntax is `where(boolarray, truearray, falsearray)`:

```
>>> a = np.array([1, 3, 0], float)
>>> np.where(a != 0, 1 / a, a)
array([ 1.        ,  0.33333333,  0.        ])
```

Broadcasting can also be used with the where function:

```
>>> np.where(a > 0, 3, 2)
array([3, 3, 2])
```

A number of functions allow testing of the values in an array. 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.

## Array item selection and manipulation

We have already seen that, like lists, individual elements and slices of arrays can be selected using bracket notation. Unlike lists, however, arrays also permit selection using other arrays. That is, we can use *array selectors* to filter for specific subsets of elements of other arrays.

Boolean arrays can be used as array selectors:

```
>>> a = np.array([[6, 4], [5, 9]], float)
>>> a >= 6
array([[ True, False],
       [False,  True]], dtype=bool)
>>> a[a >= 6]
array([ 6.,  9.])
```

Notice that sending the Boolean array given by `a>=6` to the bracket selection for `a`, an array with only the `True` elements is returned. We could have also stored the selector array in a variable:

```
>>> a = np.array([[6, 4], [5, 9]], float)
>>> sel = (a >= 6)
>>> a[sel]
array([ 6.,  9.])
```

More complicated selections can be achieved using Boolean expressions:

```
>>> a[np.logical_and(a > 5, a < 9)]
>>> array([ 6.])
```

In addition to Boolean selection, it is possible to select using integer arrays. Here, the integer arrays contain the *indices* of the elements to be taken from an array. Consider the following one-dimensional example:

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

In other words, we take the 0th, 0th, 1st, 3rd, 2nd, and 1st elements of `a`, in that order, when we use `b` to select elements from `a`. Lists can also be used as selection arrays:

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

For multidimensional arrays, we have to send multiple one-dimensional integer arrays to the selection bracket, one for each axis. Then, each of these selection arrays is traversed in sequence: the first element taken has a first axis index taken from the first member of the first selection array, a second index from the first member of the second selection array, and so on. An example:

```
>>> a = np.array([[1, 4], [9, 16]], float)
>>> b = np.array([0, 0, 1, 1, 0], int)
>>> c = np.array([0, 1, 1, 1, 1], int)
>>> a[b,c]
array([  1.,   4.,  16.,  16.,   4.])
```

A special function `take` is also available to perform selection with integer arrays. This works in an identical manner as bracket selection:

```
>>> a = np.array([2, 4, 6, 8], float)
>>> b = np.array([0, 0, 1, 3, 2, 1], int)
>>> a.take(b)
array([ 2.,  2.,  4.,  8.,  6.,  4.])
```

`take` also provides an axis argument, such that subsections of an multi-dimensional array can be taken across a given dimension.

```
>>> a = np.array([[0, 1], [2, 3]], float)
>>> b = np.array([0, 0, 1], int)
>>> a.take(b, axis=0)
array([[ 0.,  1.],
       [ 0.,  1.],
       [ 2.,  3.]])
>>> a.take(b, axis=1)
array([[ 0.,  0.,  1.],
       [ 2.,  2.,  3.]])
```