

Scientific Programming in Python  
Notes  
1 October 2009

## Arrays

Arrays are SciPy data structures for storing collections of numbers of the same type. Initially you might think of them as lists, but there are a few important differences between lists and arrays.

**Arrays are homogeneous; lists are heterogeneous** Recall that we could store a mixture of values of different data types in one list. An array can contain objects of the same type only.

```
>>> x = array([2.72, 3.14, 1])
>>> x
array([ 2.72,  3.14,  1.  ])
>>> x.dtype
dtype('float64')
```

The array in the example above contains floating point numbers. Note that the integer value (1) was also converted to a floating point number (1.) in order to fit in with the other values. We can check what type the values in an array are by reading the `dtype` attribute.

**Arrays have static size; lists have dynamic size** Once we have created an array, its size—i.e. the number of entries in the array—remains fixed. In order to change the size of an array we have to create a new array. This is different from lists, where we could append and delete values at any time, thus changing the length of the list.

**Arrays are faster than lists** One of the best reasons for using arrays is that they are significantly faster than lists when performing large numbers of calculations. The SciPy library has very optimised functions for the mathematical calculations that we are interested in. In general it is a good idea to use arrays if you want to perform the same calculation on a large number of values. The example below shows how we can use an array to compute

$$\sum_{i=1}^N i^2$$

using lists and using arrays.

```
from __future__ import division
from scipy import *

N = 1000
list_sum = sum([i**2 for i in range(1, N+1)])
array_sum = sum(arange(1, N+1)**2)
```

To compute the list sum we use list comprehension to generate a list containing the squared integers. The sum function adds all the elements in the list. To compute the array sum we use `arange` to generate an array with the first  $N$  positive integers, we then square them and sum them. On my computer the array method is approximately 23 times faster than the list method.

**Arrays can be multidimensional; lists are single dimensional** We can define arrays where we index values with more than one index. This is useful for representing not only vectors, but also matrices, tensors, and higher order objects. The following example creates a  $(3 \times 4)$  array filled with random numbers in the interval  $[0, 5)$ .

```
>>> random.uniform(0, 5, size=(3,4))
array([[ 1.38844461,  3.32587744,  4.08707978,  0.14804505],
       [ 4.2309029 ,  2.87627029,  3.47746552,  2.11885717],
       [ 1.37363963,  2.93885497,  0.8878225 ,  2.72939421]])
```

## Indexing and slicing

For 1-dimensional arrays indexing and slicing work in exactly the same way as for lists. For a multidimensional array we specify an index or slice for each dimension, separating each slice with a comma. For example, for the 2-dimensional array

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix},$$

```
>>> A[0,0]
1
>>> A[1,2]
6
>>> A[1:,:1]
array([[4, 5],
       [7, 8]])
```

An indexing feature for arrays that is not available for lists, is indexing using another array. For example, for

$$B = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 0 \end{bmatrix},$$

we can select all the rows and three of the columns as follows.

```
>>> B[:, [1, 3, 4]]
array([[2, 4, 5],
       [7, 9, 0]])
```

Note that it is not possible to make the same selection of rows and columns using slice notation.

## Operators

All of the standard mathematical operators in Python are defined to operate on arrays. To produce a result an operator is applied to each element of the input arrays.

```
>>> x = array([3, 14, 15])
>>> y = array([9, 26, 5])
>>> x * 2
array([ 6, 28, 30])
>>> x ** 2
array([ 9, 196, 225])
>>> x - y
array([ -6, -12, 10])
```

Conditional operators return an array with Boolean (`True` or `False`) entries. We continue from the previous example.

```
>>> y != 5
array([ True,  True, False])
>>> x < y
array([ True,  True, False])
```

Note that the condition is evaluated for each entry in the array independently.

Finally, it is particularly import to note that multiplying two 2-dimensional arrays is *not* the same as matrix multiplication. As in the previous examples, multiplication is performed element by element.

```
>>> x = array([[1, 2], [3, 4]])
>>> y = array([[5, 6], [7, 8]])
>>> x * y
array([[ 5, 12],
       [21, 32]])
```

## Functions

### Functions for creating arrays

The `array` function creates an array from a list. It must be possible to fit the list into multi-dimensional array.

```
>>> x = [1.5, 2, 3.1]
>>> array(x)
array([ 1.5,  2. ,  3.1])
```

The `arange` function works much like `range` except that it (a) can take floating point numbers as arguments and (b) returns an array rather than a list.

```
>>> arange(2, 10, 2)
array([2, 4, 6, 8])
>>> arange(1.5, 3, 0.3)
array([ 1.5,  1.8,  2.1,  2.4,  2.7])
```

The `linspace` function creates an array of linearly spaced values—i.e. the difference between consecutive values is constant in the array. The first two function arguments define the first and final values in the resulting list. The third function argument specifies the number of entries in the array.

```
>>> linspace(0, 1, 5)
array([ 0. ,  0.25,  0.5 ,  0.75,  1.  ])
```

The `logspace` function creates an array of exponentially spaced values—i.e. the ratio between consecutive values is constant in the array. Note that the values range from  $10^a$  to  $10^b$  where  $a$  and  $b$  are the first two arguments of the `logspace` function.

```
>>> linspace(0, 1, 5)
array([ 1. ,  1.7783,  3.1623,  5.6234,  10.  ])
```

The functions `zeros`, `ones`, `empty` each create an array with a particular size and which is filled with 0s, or 1s, or nothing. We can specify the data type with the `dtype` keyword argument.

```
>>> ones(10, dtype=int)
array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1])
>>> zeros([3,4])
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]])
```

## Mathematical functions

The mathematical functions `sin`, `cos`, `tan`, `sqrt`, `log2`, `log10`, `log`, etc. are all defined for arrays. Each function acts on each element of an input array independently.

```
>>> x = array([[1,2,3], [4,5,6], [7,8,9]])
>>> x
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
>>> sqrt(x)
array([[ 1.          ,  1.41421356,  1.73205081],
       [ 2.          ,  2.23606798,  2.44948974],
       [ 2.64575131,  2.82842712,  3.          ]])
>>> sin(x)**2 + cos(x)**2
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
```

## Aggregating functions

These functions combine the values of an array to produce a result.

- `sum` returns the sum,
- `prod` returns the product,
- `mean` returns the average,
- `median` returns the median value,
- `var` returns the variance,
- `std` returns the standard deviation,
- `min` and `max` return the minimum and maximum value respectively,
- `argmin` and `argmax` return the index of the minimum and maximum value respectively.

## Conditional functions

When evaluating a conditional operator on an array, the result is a Boolean array. The `any` function will return `True` if at least one of the values in its input array is `True`. The `all` function will return `True` if all of the values in its input array are `True`.

```
>>> x = array([1, 4, 10, 7])
>>> y = array([2, 4, 12, 14])
>>> x == y
array([False,  True, False, False])
>>> any(x == y)
True
>>> all(x == y)
False
>>> x <= y
array([ True,  True,  True,  True])
>>> any(x <= y)
True
>>> all(x <= y)
True
```

The `where` function is used to select values from two arrays to form a new array. Using the form

```
where(condition, array1, array2)
```

the result will contain values from `array1` everywhere that the `condition` array is `True` and values from `array2` where `condition` is `False`. Note that `condition`, `array1`, and `array2` must have the same size. The example below computes the following for each element in an array, `x`.

$$y_i = \begin{cases} -x_i & \text{if } x_i < 0 \\ x_i^2 & \text{if } x_i \geq 0 \end{cases}$$

```
>>> x = arange(-3, 4)
>>> y = where(x < 0, -x, x**2)
>>> x
array([-3, -2, -1,  0,  1,  2,  3])
>>> y
array([3, 2, 1, 0, 1, 4, 9])
```

## Structural functions

These functions can determine and change the shape of an array. The number of elements in an array needs to stay constant when we change the shape of the array. This means that a  $3 \times 4$  array can be changed into a  $2 \times 6$  array or a length 12 array since all of these have 12 elements. The `shape` function returns the shape of an array.

```
>>> x = array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
>>> shape(x)
(3,4)
```

The `reshape` function returns the same array but with a new shape. The number of elements must remain constant.

```
>>> reshape(x, (2,6))
array([[ 1,  2,  3,  4,  5,  6],
       [ 7,  8,  9, 10, 11, 12]])
>>> reshape(x, (2,2,3))
array([[[ 1,  2,  3],
        [ 4,  5,  6]],

       [[ 7,  8,  9],
        [10, 11, 12]]])
```

The `ravel` function returns a 1-dimensional array containing all the elements of the input array.

```
>>> ravel(x)
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12])
```