02-Numpy

Stephen Pascoe

March 16, 2014

1 An introduction to `numpy`

This material is based on a the SciTools Courses notebook available at https://github.com/SciTools/courses

While the Python language is an excellent tool for general-purpose programming, with a highly readable syntax, rich and powerful data types (strings, lists, sets, dictionaries, arbitrary length integers, etc) and a very comprehensive standard library, it was not designed specifically for mathematical and scientific computing. Neither the language nor its standard library have facilities for the efficient representation of multidimensional datasets, tools for linear algebra and general matrix manipulations (essential building blocks of virtually all technical computing), nor any data visualisation facilities.

In particular, Python lists are very flexible containers that can be nested arbitrarily deep and which can hold any Python object in them, but they are poorly suited to represent common mathematical constructs like vectors and matrices. It is for this reason that numpy exists. Typically numpy is imported as np:

In [1]:
```
import numpy as np
```

Numpy, at its core, provides a powerful array object. Let's start by exploring how the numpy array differs from Python lists. We start by creating a simple list and an array with the identical contents:

In [2]:
```
lst = [10, 20, 30, 40]
arr = np.array([10, 20, 30, 40])
print lst
print arr
```

```
[10, 20, 30, 40]
[10 20 30 40]
```

In [3]:
```
print lst[0], arr[0]
```

```
10 10
```

In [4]:
```
print lst[2:4], arr[2:4]
```

```
[30, 40] [30 40]
```

Data types

Numpy comes with a most of the common data types (and some uncommon ones too).

The most used (and portable) dtypes are:

- bool

- uint8
- int (machine dependent)
- int8
- int32
- int64
- float (machine dependent)
- float32
- float64

Full details can be found at http://docs.scipy.org/doc/numpy/user/basics.types.html.

In [5]:
```
# The information about the type of an array is contained in its dtype
arr.dtype
```

Out [5]:
```
dtype('int64')
```

Once an array has been created, its dtype is fixed (in this case to an 8 byte/64 bit signed integer) and it can only store elements of the same type. For this example where the dtype is integer, if we try storing a floating point number in the array it will be automatically converted into an integer:

In [6]:
```
arr[-1] = 1.234
arr
```

Out [6]:
```
array([10, 20, 30,  1])
```

In [7]:
```
arr.shape
```

Out [7]:
```
(4,)
```

Creating Arrays

Above we created an array from an existing list; now let us now see other ways in which we can create arrays, which we'll illustrate next. A common need is to have an array initialized with a constant value, and very often this value is 0 or 1 (suitable as starting value for additive and multiplicative loops respectively); `zeros` creates arrays of all zeros, with any desired dtype:

In [8]:
```
print '5 zeros:', np.zeros(5, dtype=np.int)
```

```
5 zeros: [0 0 0 0 0]
```

and similarly for `ones`:

In [9]:
```
print '5 ones:', np.ones(5, dtype=np.int)
```

```
5 ones: [1 1 1 1 1]
```

In [10]:
```
a = np.empty(4, dtype=np.float)
a.fill(5.5)
print a

# Alternatives such as
print np.ones(4) * 5.5
print np.zeros(4) + 5.5
```

```
[ 5.5  5.5  5.5  5.5]
[ 5.5  5.5  5.5  5.5]
[ 5.5  5.5  5.5  5.5]
```

Filling arrays with sequences

Numpy also offers the `arange` function, which works like the builtin `range` but returns an array instead of a list:

```
In [11]: np.arange(10, dtype=np.float64)
```

```
Out [11]: array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.])
```

```
In [12]: np.arange(5, 7, 0.1)
```

```
Out [12]: array([ 5. ,  5.1,  5.2,  5.3,  5.4,  5.5,  5.6,  5.7,  5.8,  5.9,
,
         6.1,  6.2,  6.3,  6.4,  6.5,  6.6,  6.7,  6.8,  6.9])
```

The `linspace` and `logspace` functions to create linearly and logarithmically-spaced grids respectively, with a fixed number of points and including both ends of the specified interval:

```
In [13]: print "A linear grid between 0 and 1:"
         print np.linspace(0, 1, 5)
```

```
A linear grid between 0 and 1:
[ 0.    0.25  0.5   0.75  1.  ]
```

1.1 Arrays with more than one dimension

Up until now all our examples have used one-dimensional arrays. But Numpy can create arrays of arbitrary dimensions, and all the methods illustrated in the previous section work with more than one dimension. For example, a list of lists can be used to initialize a two dimensional array:

```
In [14]: lst2 = [[1, 2, 3], [4, 5, 6]]
         arr2 = np.array([[1, 2, 3], [4, 5, 6]])
         print arr2
         print arr2.shape
```

```
[[1 2 3]
 [4 5 6]]
(2, 3)
```

With two-dimensional arrays we start seeing the power of numpy: while a nested list can be indexed using repeatedly the `[ ]` operator, multidimensional arrays support a much more natural indexing syntax with a single `[ ]` and a set of indices separated by commas:

```
In [15]: print lst2[0][1]
         print arr2[0, 1]
```

```
2
2
```

The array creation functions listed previously can be used with more than one dimension, for example:

In [16]:
```
np.zeros((2, 3))
```

Out [16]:
```
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
```

Slices

With multidimensional arrays, you can also use slices, and you can mix and match slices and single indices in the different dimensions:

In [17]:
```
arr = np.arange(8).reshape(2, 4)
print arr

print 'Second element from dimension 0, last 2 elements from dimension
print arr[1, 2:]

print 'All elements bar the last from dimension 0, third element from d
print arr[:-1, 2]

print 'Second element from dimension 0 (maintaining its dimension), all
print arr[1:2, :]
```

```
[[0 1 2 3]
 [4 5 6 7]]
Second element from dimension 0, last 2 elements from dimension one
[6 7]
All elements bar the last from dimension 0, third element from
dimension one: [2]
Second element from dimension 0 (maintaining its dimension), all
elements from dimension one: [[4 5 6 7]]
```

If you only provide one index, then the slice will be expanded to `":"` for all of the remaining dimensions (aka Ellipsis):

In [18]:
```
print 'First row:  ', arr[0], 'is equivalent to', arr[0, :]
print 'Second row: ', arr[1], 'is equivalent to', arr[1, :]
```

```
First row:   [0 1 2 3] is equivalent to [0 1 2 3]
Second row:  [4 5 6 7] is equivalent to [4 5 6 7]
```

1.2 Generating 2D coordinate arrays

A common task is to generate a pair of arrays which represent the coordinates of our data. When the orthogonal 1d coordinate arrays already exist, numpy's meshgrid function is very useful:

In [19]:
```
x_g = np.linspace(0, 9, 3)
y_g = np.linspace(-8, 4, 3)
x2d, y2d = np.meshgrid(x_g, y_g)
print x2d
print y2d
```

```
[[ 0.   4.5  9. ]
 [ 0.   4.5  9. ]
 [ 0.   4.5  9. ]]
[[-8. -8. -8.]
 [-2. -2. -2.]
 [ 4.  4.  4.]]
```

1.3 Operating with arrays

Arrays support all regular arithmetic operators, and the numpy library also contains a complete collection of basic mathematical functions that operate on arrays. It is important to remember that in general, all operations with arrays are applied *element-wise*, i.e., are applied to all the elements of the array at the same time. Consider for example:

```
In [20]: arr1 = np.arange(4)
         arr2 = np.arange(10, 14)
         print arr1, '+', arr2, '=', arr1 + arr2

         [0 1 2 3] + [10 11 12 13] = [10 12 14 16]
```

Importantly, even the multiplication operator is by default applied element-wise, it is *not* the matrix multiplication from linear algebra:

```
In [21]: print arr1, '*', arr2, '=', arr1 * arr2

         [0 1 2 3] * [10 11 12 13] = [ 0 11 24 39]
```

We may also multiply an array by a scalar:

```
In [22]: 1.5 * arr1

Out [22]: array([ 0. ,  1.5,  3. ,  4.5])
```

This is an example of **broadcasting**.Pictorially:

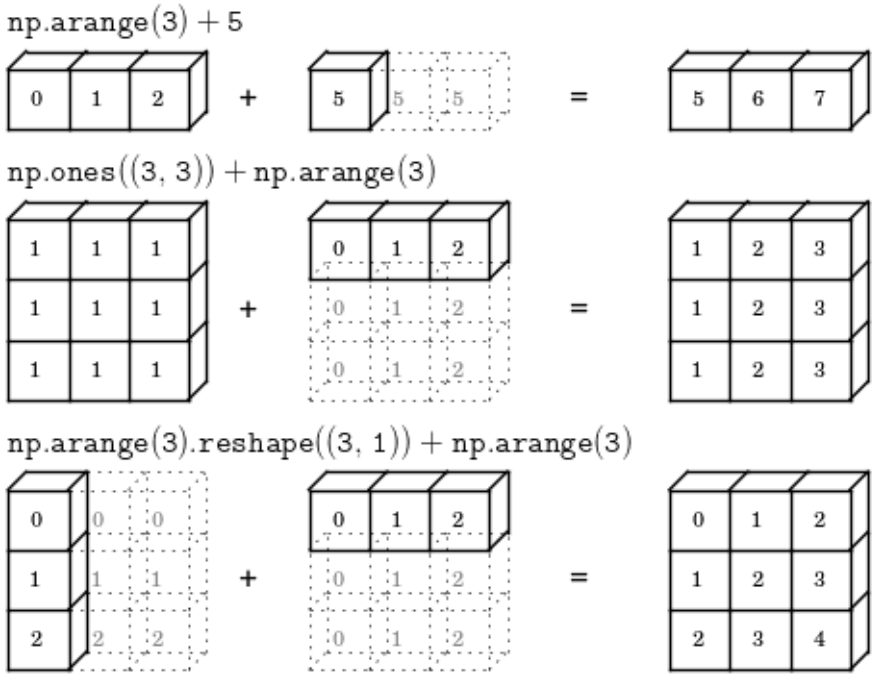

Figure 1: Illustration of broadcasting

(image source)

Exercise 3 : Numpy arrays

1.4 Further Reading

1. http://www.numpy.org
2. http://www.scipy.org

In []: