MATH40006: An Introduction To Computation
Course notes, Section 7

These notes, together with all the other resources you'll need, are available on Blackboard, at

`https://bb.imperial.ac.uk/`

7   The `numpy` module

You've already met the NumPy module, but it's time we took a deeper dive into it. NumPy is, of course, short for Numerical Python, and is the key module for scientific computing in Python. So useful is it that we'll often want to import it in its entirety, and there's a convention that says that we do that like this:

`import numpy as np`

We can then put the shortened prefix `np`, instead of `numpy`, in front of any functions or constants we use.

7.1   Arrays

At the heart of NumPy is the specialised data structure called an **array**. The simplest way to make one is from a list or tuple, using the constructor function `array`:

```
arr1 = np.array([1, 4, 7])
arr2 = np.array([5, -6, 2])
```

Arrays look superficially like lists or tuples, but they behave very differently, as you'll see. Two major differences are

- a list or tuple can contain a variety of different types of data, whereas an array must consist of data of just one type;

- arrays behave radically differently from lists or tuples when added, or when multiplied by a scalar.

To see the former property, try typing

```
mixed_list = [1, 2.0, 3, 4.0]
mixed_array = np.array(mixed_list)
print(mixed_list)
print(mixed_array)
```

```
[1, 2.0, 3, 4.0]
[1. 2. 3. 4.]
```

Notice that the elements of `mixed_array` aren't mixed at all; they're all floats! (Notice, too, in passing, that the `print` function displays the elements of an array without separating commas.)

To see the second property, contrast

```
[1, 4, 7] + [5, -6, 2]
```

```
[1, 4, 7, 5, -6, 2]
```

and

```
np.array([1, 4, 7]) + np.array([5, -6, 2])
```

```
array([6, -2, 9])
```

Contrast, too,

```
3 * [1, 4, 7]
```

```
[1, 4, 7, 1, 4, 7]
```

and

```
3 * np.array([1, 4, 7])
a3 = 1
b3 = 2
tol3 = 1e-7
```

```
array([3, 12, 21])
```

NumPy arrays don't concatenate like lists and tuples; instead, addition, or scalar multiplication, works as if the arrays were **vectors**. (This is no accident, as we'll see.)

If you base your array on a **list of lists**, it will end up **two-dimensional**, like a matrix (indeed, *very* like a matrix, as we'll explore later):

```
array2d = np.array([[1, 3, 5], [2, 4, 6], [1,-1,1]])
print(array2d)
```

```
[[ 1  3  5]
 [ 2  4  6]
 [ 1 -1  1]]
```

These behave in a similar way to 1D arrays when added, or multiplied by a scalar:

```
print(array2d + array2d)
```

```
[[ 2  6 10]
 [ 4  8 12]
 [ 2 -2  2]]
```

```
print(3.1 * array2d)
```

```
[[ 3.1  9.3 15.5]
 [ 6.2 12.4 18.6]
 [ 3.1 -3.1  3.1]]
```

It's even possible to create 3-dimensional, or 4-dimensional, or 5-dimensional arrays, or whatever, via lists of lists of lists of . . . :

```
array3d = np.array([[[1, 3, 5],[2, 4, 6], [1,-1,1]],
          [[6, 0, -1],[0,1, -6], [1, 2, 1]]])
print(array3d)
```

```
[[[ 1  3  5]
  [ 2  4  6]
  [ 1 -1  1]]

 [[ 6  0 -1]
  [ 0  1 -6]
  [ 1  2  1]]]
```

7.2   The `linspace`, `arange`, `zeros` and `ones` functions

There are other ways of creating NumPy arrays. It's often useful to have an array all of whose elements are equally spaced, such as values between 0 and 2π in steps of $\pi/6$. To create a *list* containing those values, we'd have to use a loop or, better, a comprehension:

```
from math import pi
x_list = [i * pi/6 for i in range(13)]
print(x_list)
```

```
[0.0, 0.5235987755982988, 1.0471975511965976, 1.5707963267948966,
2.0943951023931953, 2.6179938779914944, 3.141592653589793,
3.665191429188092, 4.1887902047863905, 4.71238898038469,
5.235987755982989, 5.759586531581287, 6.283185307179586]
```

But with NumPy arrays, it's far easier. We can either use the `linspace` function, which allows us to specify the first value, the last value and the number of values . . .

```
x_arr = np.linspace(0, 2*np.pi, 13)
print(x_arr)
```

```
[0.         0.52359878 1.04719755 1.57079633 2.0943951  2.61799388
 3.14159265 3.66519143 4.1887902  4.71238898 5.23598776 5.75958653
 6.28318531]
```

... or the `arange` function, which allows us to set up a range of values—a little like the `range` function, except that the step size is allowed to be a float.

```
x_arr2 = np.arange(0, 2*np.pi+0.01, np.pi/6)
print(x_arr2)
```

```
[0.         0.52359878 1.04719755 1.57079633 2.0943951  2.61799388
 3.14159265 3.66519143 4.1887902  4.71238898 5.23598776 5.75958653
 6.28318531]
```

Notice that `arange` obeys the standard Python system: it delivers those numbers, in steps of $\pi/6$, that are greater than or equal to zero but **strictly less than** $2\pi + 0.01$. For this reason,

```
x_arr2 = np.arange(0, 2*np.pi, np.pi/6)
print(x_arr2)
```

wouldn't have worked.

It may not be immediately obvious why, but it turns out to be really useful to be able to create NumPy arrays consisting entirely of 0s or 1s.

```
print(np.zeros(5))
print(np.ones(5))
print(np.zeros([3,4]))
print(np.ones([2,6]))
```

```
[0. 0. 0. 0. 0.]
[1. 1. 1. 1. 1.]
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
[[1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1.]]
```

7.3 Array operators

We've already seen that if two arrays are the same shape, it's possible to add them, and they'll add component-by-component. Well, it turns out it's also possible to use all the other arithmetic operators on arrays as well: `-`, `*`, `/`, `//` and `**`.

```
arr1 = np.array([[1, 2, 5], [0, 4, 3]])
arr2 = np.array([[5, 3, 2], [1, 3, 2]])
print(arr1 + arr2)
print(arr1 - arr2)
print(arr1 * arr2)
print(arr1 / arr2)
print(arr1 // arr2)
print(arr1 ** arr2)
```

```
[[6 5 7]
 [1 7 5]]
[[-4 -1  3]
 [-1  1  1]]
[[ 5  6 10]
 [ 0 12  6]]
[[0.2        0.66666667 2.5        ]
 [0.         1.33333333 1.5        ]]
[[0 0 2]
 [0 1 1]]
[[ 1  8 25]
 [ 0 64  9]]
```

Again, the operations are all carried out component-by-component.

It's also possible to use all the arithmetical operations on a NumPy array and a scalar:

```
arr1 = np.array([[1, 2, 5], [0, 4, 3]])
print(arr1 + 2)
print(arr1 - 2)
print(arr1 * 2)
print(arr1 / 2)
print(arr1 // 2)
print(arr1 ** 2)
```

```
[[3 4 7]
 [2 6 5]]
[[-1  0  3]
 [-2  2  1]]
[[ 2  4 10]
 [ 0  8  6]]
[[0.5 1.  2.5]
 [0.  2.  1.5]]
[[0 1 2]
 [0 2 1]]
```

```
[[ 1  4 25]
 [ 0 16  9]]
```

So: you can add (or multiply, or subtract, etc) two arrays of exactly the same shape, and you can add (or multiply, or subtract, etc) an array and a scalar. Can you do anything else? Well, yes, actually. The following works, for example:

```
arr1 = np.array([[1, 2, 5], [0, 4, 3]])
arr2 = np.array([[5, 3, 2]])
print(arr1 + arr2)
print(arr1 - arr2)
print(arr1 * arr2)
print(arr1 / arr2)
print(arr1 // arr2)
print(arr1 ** arr2)
```

```
[[6 5 7]
 [5 7 5]]
[[-4 -1  3]
 [-5  1  1]]
[[ 5  6 10]
 [ 0 12  6]]
[[0.2        0.66666667 2.5       ]
 [0.        1.33333333 1.5       ]]
[[0 0 2]
 [0 1 1]]
[[ 1  8 25]
 [ 0 64  9]]
```

On the other hand, the following doesn't:

```
arr1 = np.array([[1, 2, 5], [0, 4, 3], [-2, 2, -3]])
arr2 = np.array([[5, 3, 2], [1, 3, 2]])
print(arr1 + arr2)
print(arr1 - arr2)
print(arr1 * arr2)
print(arr1 / arr2)
print(arr1 // arr2)
print(arr1 ** arr2)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-20-b161a505ab0d> in <module>()
      1 arr1 = np.array([[1, 2, 5], [0, 4, 3], [-2, 2, -3]])
```

```
      2 arr2 = np.array([[5, 3, 2], [1, 3, 2]])
----> 3 print(arr1 + arr2)
      4 print(arr1 - arr2)
      5 print(arr1 * arr2)
```

`ValueError: operands could not be broadcast together with shapes (3,3) (2,3)`

More about what exactly the rules are, what exactly happens, and what exactly Python means by "broadcast", in the exercises.

7.4 Mathematical functions

The NumPy module comes with a complete set of mathematical functions, largely paralleling those in the `math` and `cmath` modules:

```
print(np.sin(np.pi/6))
print(np.cos(np.pi/6))
print(np.exp(np.log(2)))
```

```
0.49999999999999994
0.8660254037844387
2.0
```

However, there's a difference, which is that NumPy's functions map automatically across arrays:

```
angles = np.arange(0,np.pi/2+0.01,np.pi/6)
sines = np.sin(angles)
print(sines)
```

```
[0.        0.5       0.8660254 1.        ]
```

This is fantastically useful. Consider, for example, the problem of plotting $\cos x$ against x. Here's how we'd have to do that without NumPy.

```
from math import pi, cos
x_values = [i * pi/100 for i in range(201)]
y_values = [cos(x) for x in x_values]
plt.plot(x_values, y_values)
```

With NumPy, this is simply

```
import numpy as np
x_values = np.linspace(0, 2*np.pi, 201)
```

```
    y_values = np.cos(x_values)
    plt.plot(x_values, y_values)
```

No need to use comprehensions at all! I think this is simpler and easier. And, as we'll see later, this "vectorized" way of working can also be fundamentally much more *efficient* than using comprehensions or loops, meaning that NumPy can offer strategies for speeding up certain programs very dramatically.

7.5   Arrays as matrices and vectors

NumPy supports a comprehensive selection of linear algebra functions and methods. 1D arrays can be used to represent vectors . . .

```
vec1 = np.array([-1,2,2])
vec2 = np.array([2,-1,2])
print(np.dot(vec1, vec2))
print(np.cross(vec1, vec2))
```

```
0
[ 6  6 -3]
```

. . . and 2D arrays can ve used to represent matrices . . .

```
mat1 = np.array([[2, 3, -2], [1, -5, 0], [-2, 1, 2]])
mat2 = np.array([[1, 3], [-1, 0], [2, -1]])
print(np.dot(mat1, mat2))
```

```
[[-5  8]
 [ 6  3]
 [ 1 -8]]
```

Notice that the same function, `dot`, is used for the scalar product of two vectors and for matrix multiplication.

7.6   The `linalg` submodule

The basic linear algebra functions `dot` and `cross` live in the top level of NumPy; for anything even a bit more specialised, you need the `linalg` submodule. First let's import it; then we can use it to calculate the determinant, and the inverse, of our square matrix `mat1`.

```
import numpy.linalg
print(np.linalg.det(mat1))
print(np.linalg.inv(mat1))
```

```
-8.000000000000002
[[ 1.25   1.     1.25 ]
 [ 0.25  -0.     0.25 ]
 [ 1.125  1.     1.625]]
```

8

7.7 Polynomials

NumPy has a collection of functions for dealing with the algebra of polynomials, which are represented by a special kind of array called `poly1d`. These arrays consist of the coefficients in the polynomial, in descending power order. Here, for example, are the polynomials $x^2 - 3x + 2$ and $x^3 - 2x^2 - 5x + 6$:

```
poly1 = np.poly1d([1, -3, 2])
poly2 = np.poly1d([1, -2, -5, 6])
```

We can then add or multiply...

```
print(poly1 + poly2)
print(poly1 * poly2)
```

```
   3     2
1 x - 1 x - 8 x + 8
   5     4     3      2
1 x - 5 x + 3 x + 17 x - 28 x + 12
```

...calculate roots, and substitute in values...

```
print(np.roots(poly2))
print(np.polyval(poly1, [1, 3, 5, 7]))
```

```
[-2.  3.  1.]
[ 0  2 12 30]
```

...differentiate and integrate ...

```
print(np.polyder(poly2))
print(np.polyint(poly1))
```

```
    2
3 x - 4 x - 5
        3        2
0.3333 x - 1.5 x + 2 x
```

...and so on.