2. Mathematics with arrays

One of the great advantages of Numpy arrays is that they allow one to very easily apply mathematical operations to entire arrays effortlessly. We are presenting here 3 ways in which this can be done.

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

2.1 Simple calculus

To illustrate how arrays are useful, let's first consider the following problem. You have a list:

In [2]:
```
mylist = [1,2,3,4,5]
```

And now you wish to add to each element of that list the value 3. If we write:

In [3]:
```
mylist + 3
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-3-ecae2962d7b1> in <module>
----> 1 mylist + 3

TypeError: can only concatenate list (not "int") to list
```

We receive an error because Python doesn't know how to combine a list with a simple integer. In this case we would have to use a for loop or a comprehension list, which is cumbersome.

In [4]:
```
[x + 3 for x in mylist]
```

Out[4]:
```
[4, 5, 6, 7, 8]
```

Let's see now how this works for an array:

In [5]:
```
myarray = np.array(mylist)
```

In [6]:
```
myarray + 3
```

Out[6]:
```
array([4, 5, 6, 7, 8])
```

Numpy understands without trouble that our goal is to add the value 3 to *each element* in our list. Naturally this is dimension independent e.g.:

In [7]:
```
my2d_array = np.ones((3,6))
my2d_array
```

Out[7]:
```
array([[1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1.]])
```

In [8]:
```
my2d_array + 3
```

Out[8]:
```
array([[4., 4., 4., 4., 4., 4.],
       [4., 4., 4., 4., 4., 4.],
       [4., 4., 4., 4., 4., 4.]])
```

Of course as long as we don't reassign this new state to our variable it remains unchanged:

In [9]:
```
my2d_array
```

Out[9]:
```
array([[1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1.]])
```

We have to write:

In [10]:
```
my2d_array = my2d_array + 3
```

In [11]:
```
my2d_array
```

Out[11]:
```
array([[4., 4., 4., 4., 4., 4.],
       [4., 4., 4., 4., 4., 4.],
       [4., 4., 4., 4., 4., 4.]])
```

Naturally all basic operations work:

In [12]:
```
my2d_array * 4
```

Out[12]:
```
array([[16., 16., 16., 16., 16., 16.],
       [16., 16., 16., 16., 16., 16.],
       [16., 16., 16., 16., 16., 16.]])
```

In [13]:
```
my2d_array / 5
```

Out[13]:
```
array([[0.8, 0.8, 0.8, 0.8, 0.8, 0.8],
       [0.8, 0.8, 0.8, 0.8, 0.8, 0.8],
       [0.8, 0.8, 0.8, 0.8, 0.8, 0.8]])
```

In [14]:
```
my2d_array ** 5
```

Out[14]:
```
array([[1024., 1024., 1024., 1024., 1024., 1024.],
       [1024., 1024., 1024., 1024., 1024., 1024.],
       [1024., 1024., 1024., 1024., 1024., 1024.]])
```

2.2 Mathematical functions

In addition to simple arithmetic, Numpy offers a vast choice of functions that can be directly applied to arrays. For example trigonometry:

In [15]:
```
np.cos(myarray)
```

Out[15]:
```
array([ 0.54030231, -0.41614684, -0.9899925 , -0.65364362,  0.28366219])
```

Exponentials and logs:

In [16]:
```
np.exp(myarray)
```

Out[16]:
```
array([  2.71828183,   7.3890561 ,  20.08553692,  54.59815003,
       148.4131591 ])
```

In [17]:
```
np.log10(myarray)
```

Out[17]:
```
array([0.        , 0.30103   , 0.47712125, 0.60205999, 0.69897   ])
```

2.3 Logical operations

If we use a logical comparison on a regular variable, the output is a *boolean* (True or False) that describes the outcome of the comparison:

In [18]:
```
a = 3
b = 2
a > 3
```

Out[18]:
```
False
```

We can do exactly the same thing with arrays. When we added 3 to an array, that value was automatically added to each element of the array. With logical operations, the comparison is also done for each element in the array resulting in a boolean array:

In [19]:
```
myarray = np.zeros((4,4))
myarray[2,3] = 1
myarray
```

Out[19]:
```
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 1.],
       [0., 0., 0., 0.]])
```

In [20]:
```
myarray > 0
```

Out[20]:
```
array([[False, False, False, False],
       [False, False, False, False],
       [False, False, False,  True],
       [False, False, False, False]])
```

Exactly as for simple variables, we can assign this boolean array to a new variable directly:

In [21]:
```
myboolean = myarray > 0
```

In [22]:
```
myboolean
```

Out[22]:
```
array([[False, False, False, False],
       [False, False, False, False],
       [False, False, False,  True],
       [False, False, False, False]])
```

2.4 Methods modifying array dimensions

The operations described above were applied *element-wise*. However sometimes we need to do operations either at the array level or some of its axes. For example, we need very commonly statistics on an array (mean, sum etc.)

In [23]:
```
nd_array = np.random.normal(10, 2, (3,4))
nd_array
```

Out[23]:
```
array([[ 8.22235922, 10.86316749,  8.97190654, 12.16211971],
       [11.31745909,  9.80774793, 11.2873836 ,  6.77945745],
       [10.20776894,  8.78011512,  6.96723135, 11.77819806]])
```

In [24]:
```
np.mean(nd_array)
```

Out[24]:
```
9.762076209457817
```

In [25]:
```
np.std(nd_array)
```

Out[25]:
```
1.747626512794281
```

Or the maximum value:

In [26]:
```
np.max(nd_array)
```

Out[26]:
```
12.162119714449235
```

Note that several of these functions can be called as array methods instead of numpy functions:

In [27]:
```
nd_array.mean()
```

Out[27]:
```
9.762076209457817
```

In [28]:
```
nd_array.max()
```

Out[28]:
```
12.162119714449235
```

Note that most functions can be applied to specific axes. Let's remember that our arrays is:

In [29]:
```
nd_array
```

Out[29]:
```
array([[ 8.22235922, 10.86316749,  8.97190654, 12.16211971],
       [11.31745909,  9.80774793, 11.2873836 ,  6.77945745],
       [10.20776894,  8.78011512,  6.96723135, 11.77819806]])
```

We can for example do a maximum projection along the first axis (rows): the maximum value of eadch column is kept:

In [30]:
```
proj0 = nd_array.max(axis=0)
proj0
```

Out[30]:
```
array([11.31745909, 10.86316749, 11.2873836 , 12.16211971])
```

In [31]:
```
proj0.shape
```

Out[31]:
```
(4,)
```

We can of course do the same operation for the second axis:

In [32]:
```
proj1 = nd_array.max(axis=1)
proj1
```

Out[32]:
```
array([12.16211971, 11.31745909, 11.77819806])
```

In [33]:
```
proj1.shape
```

Out[33]:
```
(3,)
```

There are of course more advanced functions. For example a cumulative sum:

In [34]:
```
np.cumsum(nd_array)
```

Out[34]:
```
array([  8.22235922,  19.08552671,  28.05743325,  40.21955296,
        51.53701205,  61.34475998,  72.63214358,  79.41160103,
        89.61936998,  98.3994851 , 105.36671645, 117.14491451])
```