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 [15]: import numpy as np
```

1 Simple calculus

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

```
In [16]: 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 [17]: mylist + 3
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[17], line 1
----> 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 [18]: [x + 3 for x in mylist]
Out[18]: [4, 5, 6, 7, 8]
```

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

```
In [19]: myarray = np.array(mylist)
```

```
In [20]: myarray + 3
Out[20]: 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 [21]: my2d_array = np.ones((3,6))
         my2d_array
```

```
Out[21]: array([[1., 1., 1., 1., 1., 1.],
                [1., 1., 1., 1., 1., 1.],
                [1., 1., 1., 1., 1., 1.]])
```

```
In [22]: my2d_array + 3
```

```
Out[22]: 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 [23]: my2d_array
```

```
Out[23]: array([[1., 1., 1., 1., 1., 1.],
                [1., 1., 1., 1., 1., 1.],
                [1., 1., 1., 1., 1., 1.]])
```

We have to write:

```
In [24]: my2d_array = my2d_array + 3
```

```
In [25]: my2d_array
```

```
Out[25]: 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 [26]: my2d_array * 4
```

```
Out[26]: array([[16., 16., 16., 16., 16., 16.],
                [16., 16., 16., 16., 16., 16.],
                [16., 16., 16., 16., 16., 16.]])
```

```
In [27]: my2d_array / 5
```

```
Out[27]: 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 [28]: my2d_array ** 5
```

```
Out[28]: array([[1024., 1024., 1024., 1024., 1024., 1024.],
                [1024., 1024., 1024., 1024., 1024., 1024.],
                [1024., 1024., 1024., 1024., 1024., 1024.]])
```

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 [29]: np.cos(myarray)
```

```
Out[29]: array([ 0.54030231, -0.41614684, -0.9899925 , -0.65364362,  0.28366219])
```

Exponentials and logs:

```
In [30]: np.exp(myarray)
```

```
Out[30]: array([   2.71828183,    7.3890561 ,   20.08553692,   54.59815003,
                 148.4131591 ])
```

```
In [31]: np.log10(myarray)
```

```
Out[31]: array([0.        , 0.30103   , 0.47712125, 0.60205999, 0.69897   ])
```

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 [32]: a = 3
         b = 2
         a > 3
```

```
Out[32]: 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 [33]: myarray = np.zeros((4,4))
         myarray[2,3] = 1
         myarray
```

```
Out[33]: array([[0., 0., 0., 0.],
                [0., 0., 0., 0.],
                [0., 0., 0., 1.],
                [0., 0., 0., 0.]])
```

```
In [34]: myarray > 0
```

```
Out[34]: 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 [35]: myboolean = myarray > 0
```

```
In [36]: myboolean
```

```
Out[36]: array([[False, False, False, False],
                [False, False, False, False],
                [False, False, False,  True],
                [False, False, False, False]])
```

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

```
Out[37]: array([[10.91240747,  8.4170081 ,  9.53223542, 12.23862268],
                [13.46374315, 11.37366878,  9.15293105, 11.25124047],
                [12.38752687, 10.1078357 , 13.66292386,  9.47170314]])
```

```
In [38]: np.mean(nd_array)
```

```
Out[38]: 10.99765389271875
```

```
In [39]: np.std(nd_array)
```

```
Out[39]: 1.6381000029622923
```

Or the maximum value:

```
In [40]: np.max(nd_array)
```

```
Out[40]: 13.66292386459667
```

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

```
In [41]: nd_array.mean()
```

```
Out[41]: 10.99765389271875
```

```
In [42]: nd_array.max()
```

```
Out[42]: 13.66292386459667
```

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

```
In [43]: nd_array
```

```
Out[43]: array([[10.91240747,  8.4170081 ,  9.53223542, 12.23862268],
                [13.46374315, 11.37366878,  9.15293105, 11.25124047],
                [12.38752687, 10.1078357 , 13.66292386,  9.47170314]])
```

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

```
In [44]: proj0 = nd_array.max(axis=0)
         proj0
```

```
Out[44]: array([13.46374315, 11.37366878, 13.66292386, 12.23862268])
```

```
In [45]: proj0.shape
```

```
Out[45]: (4,)
```

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

```
In [46]: proj1 = nd_array.max(axis=1)
         proj1
```

```
Out[46]: array([12.23862268, 13.46374315, 13.66292386])
```

```
In [47]: proj1.shape
```

```
Out[47]: (3,)
```

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

```
In [48]: np.cumsum(nd_array)
```

```
Out[48]: array([ 10.91240747,  19.32941557,  28.861651  ,  41.10027368,
                 54.56401683,  65.93768561,  75.09061666,  86.34185713,
                 98.729384  , 108.8372197 , 122.50014357, 131.97184671])
```