# 6. Numpy

The standard Python data types are not very suited for mathematical operations. For example, suppose we have the list `a = [2, 3, 8]`. If we multiply this list by an integer, we get:

```
>>> a = [2, 3, 8]
>>> 2 * a
[2, 3, 8, 2, 3, 8]
```

And `float`'s are not even allowed:

```
>>> a = [2, 3, 8]
>>> 2 * a
>>> 2.1 * a
TypeError: can't multiply sequence by non-int of type 'float'
```

In order to solve this using Python lists, we would have to do something like:

```
values = [2, 3, 8]
result = []
for x in values:
    result.append(2.1 * x)
```

This is not very elegant, is it? This is because Python `list`'s are not designed as mathematical objects. Rather, they are purely a collection of items. In order to get a type of list which behaves like a mathematical array or matrix, we use Numpy.

```
>>> import numpy as np
>>> a = np.array([2, 3, 8])
>>> 2.1 * a
array([  4.2,   6.3,  16.8])
```

As we can see, this worked the way we expected it to. We note a couple of things: - We abbreviated numpy to np, this is conventional. - `np.array` takes a Python list as argument. - The list `[2, 3, 8]` contains `int`'s, yet the result contains `float`'s. This means numpy changed the data type automatically for us.

Now let's take it a step further and see what happens when we multiply together array's.

```
>>> import numpy as np
>>> a = np.array([2, 3, 8])
>>> a * a
array([ 4,  9, 64])
>>> a**2
array([ 4,  9, 64])
```

This has nicely squared the array element-wise.

Note

Those in the know might be a bit surprised by this. After all, if `a` is a vector, shouldn't `a**2` be the dot product of the two vectors, $\vec{a} \cdot \vec{a}$? Well, `numpy` arrays are not vectors in the algebraic sense. Arithmetic operations between arrays are performed element-wise, not on the arrays as a whole.

To tell `numpy` we want the dot product we simply use the `np.dot` function:

```
>>> a = np.array([2, 3, 8])
>>> np.dot(a,a)
77
```

Furthermore, if you pass 2D arrays to `np.dot` it will behave like matrix multiplication. Several other similar NumPy algebraic functions are available (like `np.cross`, `np.outer`, etc.)

**Bottom line**: when you want to treat `numpy` array operations as vector or matrix operations, make use of the specialized functions to this end.

## 6.1. Shape

One of the most important properties an array is its shape. We have already seen 1 dimensional (1D) arrays, but arrays can have any dimensions you like. Images for example, consist of a 2D array of pixels. But in color images every pixel is an RGB tuple: the intensity in red, green and blue. Every pixel itself is therefore an array as well. This makes a color image 3D overall.

To get the shape of an array, we use `shape`:

```
>>> import numpy as np
>>> a = np.array([2, 3, 8])
>>> a.shape
(3,)
```

Something slightly more interesting:

```
>>> b = np.array([
    [2, 3, 8],
    [4, 5, 6],
    ])
>>> b.shape
(2, 3)
```

## 6.2. Slicing

Just like with lists, we might want to select certain values from an array. For 1D arrays it works just like for normal python lists:

```
>>> a = np.array([2, 3, 8])
>>> a[2]
8
>>> a[1:]
np.array([3, 8])
```

However, when dealing with higher dimensional arrays something else happens:

```
>>> b = np.array([
    [2, 3, 8],
    [4, 5, 6],
    ])
>>> b[1]
array([4, 5, 6])
>>> b[1][2]
6
```

We see that using `b[1]` returns the 1th row along the first dimenion, which is still an array. After that, we can select individual items from that. This can be abbreviated to:

```
>>> b[1, 2]
6
```

But what if I wanted the 1th column instead of the first row? Then we use `:` to select all items along the first dimension, and then a `1`:

```
>>> b[:, 1]
array([3, 5])
```

By comparing with the definition of `b`, we see that this is the column we were looking for.

> **Note**
>
> Instead of first, I write 1th on purpose to signify the existence of a 0th element. Remember that in Python, as in any self-respecting programming language, we start counting at zero.

Find out more about advanced slicing at the Numpy indexing documentation page.

## 6.3. Masking

This is perhaps the single most powerful feature of Numpy. Suppose we have an array, and we want to throw away all values above a certain cutoff:

```
>>> a = np.array([230, 10, 284, 39, 76])
>>> cutoff = 200
>>> a > cutoff
np.array([True, False, True, False, False])
```

Simply using the larger than operator lets us know in which cases the test was positive. Now we set all the values above 200 to zero:

```
>>> a = np.array([230, 10, 284, 39, 76])
>>> cutoff = 200
>>> a[a > cutoff] = 0
>>> a
np.array([0, 10, 0, 39, 76])
```

The crucial line is `a[a > cutoff] = 0`. This selects all the points in the array where the test was positive and assigns 0 to that position. Without knowing this trick we would have had to loop over the array:

```
>>> a = np.array([230, 10, 284, 39, 76])
>>> cutoff = 200
>>> new_a = []
>>> for x in a:
>>>     if x > cutoff:
```

```
>>>         new_a.append(0)
>>>     else:
>>>         new_a.append(x)
>>> a = np.array(new_a)
```

Looks rather silly now, doesn't it? When working with images this becomes even more obvious, because there we might have to loop over three dimensions before we can use the if/else. Can you imagine the mess?

## 6.4. Broadcasting

Another powerful feature of Numpy is broadcasting. Broadcasting takes place when you perform operations between arrays of different shapes. For instance

```
>>> a = np.array([
    [0, 1],
    [2, 3],
    [4, 5],
    ])
>>> b = np.array([10, 100])
>>> a * b
array([[  0, 100],
       [ 20, 300],
       [ 40, 500]])
```

The shapes of `a` and `b` don't match. In order to proceed, Numpy will stretch `b` into a second dimension, as if it were stacked three times upon itself. The operation then takes place element-wise.

One of the rules of broadcasting is that only dimensions of size 1 can be stretched (if an array only has one dimension, all other dimensions are considered for broadcasting purposes to have size 1). In the example above `b` is 1D, and has shape (2,). For broadcasting with `a`, which has two dimensions, Numpy adds another dimension of size 1 to `b`. `b` now has shape (1, 2). This new dimension can now be stretched three times so that `b`'s shape matches `a`'s shape of (3, 2).

The other rule is that dimensions are compared from the last to the first. Any dimensions that do not match must be stretched to become equally sized. However, according to the previous rule, only dimensions of size 1 can stretch. This means that some shapes cannot broadcast and Numpy will give you an error:

```
>>> c = np.array([
    [0, 1, 2],
    [3, 4, 5],
    ])
```

```
>>> b = np.array([10, 100])
>>> c * b
ValueError: operands could not be broadcast together with shapes (2,3) (2,)
```

What happens here is that Numpy, again, adds a dimension to `b`, making it of shape (1, 2). The sizes of the last dimensions of `b` and `c` (2 and 3, respectively) are then compared and found to differ. Since none of these dimensions is of size 1 (therefore, unstretchable) Numpy gives up and produces an error.

The solution to multiplying `c` and `b` above is to specifically tell Numpy that it must add that extra dimension as the second dimension of `b`. This is done by using `None` to index that second dimension. The shape of `b` then becomes (2, 1), which is compatible for broadcasting with `c`:

```
>>> c = np.array([
    [0, 1, 2],
    [3, 4, 5],
    ])
>>> b = np.array([10, 100])
>>> c * b[:, None]
array([[  0,  10,  20],
       [300, 400, 500]])
```

A good visual description of these rules, together with some advanced broadcasting applications can be found in this tutorial of Numpy broadcasting rules.

## 6.5. dtype

A commonly used term in working with numpy is `dtype` - short for data type. This is typically int or float, followed by some number, e.g. `int8`. This means the value is integer with a size of 8 bits. As an example, let's discuss the properties of an `int8`.

Each bit is either 0 or 1. With 8 of them, we have $2^8 = 256$ possible values. Since we also have to count zero itself, the largest possible value is 255. The data type we have now described is called `uint8`, where the u stands for unsigned: only positive values are allowed. If we want to allow negative numbers we use `int8`. The range then shifts to -128 to +127.

The same holds for bigger numbers. An `int64` for example is a 64 bit unsigned integer with a range of -9223372036854775808 to 9223372036854775807. It is also the standard type on a 64 bits machine. You might think bigger is better. You'd be wrong. If you know the elements of your array are never going to be bigger than 100,

why waste all the memory space? You might be better off setting your array to `uint8` to conserve memory. In general however, the default setting is fine. Only when you run into memory related problems should you remember this comment.

What happens when you set numbers bigger than the maximum value of your dtype?

```
>>> import numpy as np
>>> a = np.array([200], dtype='uint8')
>>> a + a
array([144], dtype=uint8)
```

That doesn't seem right, does it? If you add two `uint8`, the result of 200 + 200 cannot be 400, because that doesn't fit in a `uint8`. In standard Python, Python does a lot of magic in the background to make sure the result is the 400 you would expect. But numpy doesn't, and will return 144. Why 144 is left as an exercise. To fix this, you should make sure that your numbers where not stored as `uint8`, but as something larger; `uint16` for example. That way the resulting 400 will fit.

```
>>> import numpy as np
>>> a = np.array([200], dtype='uint16')
>>> a + a
array([400], dtype=uint16)
```

By now you must be thinking: so bigger is better after all! Just use the biggest possible int all the time, and you'll be fine! Apart from the fact that there is no biggest int, there is a bigger problem. If you work with images, each pixel from that image is stored as an RGB tuple: the intensity in red, green and blue. Each of these is a `uint8` value for most standard formats such as .jpg and .png. For example, (0, 0, 0) will be black, and (255, 0, 0) is red. This means that when you load an image from your hard drive this dtype is selected for you, and if you are not aware of this, what will happen when you add an image to itself? (In other words, place two copies on top of each other) You might expect that everything will become more dense. Instead, you'll get noise because of the effect we just talked about.

## 6.6. Changing dtype

To change the dtype of an existing array, you can use the `astype` method:

```
>>> import numpy as np
>>> a = np.array([200], dtype='uint8')
>>> a.astype('uint64')
```

## 6.7. Advanced Usage

Numpy has vast capabilities. It has way too many options to discuss here. More information can be found in

1. the Quickstart Numpy Tutorial;
2. the Numpy indexing documentation (for advanced slicing and indexing);
3. and the Numpy broadcasting rules (for what happens when performing operations between arrays of different shapes and sizes).

## 6.8. Excercises

1. Make an array with `dtype = uint8` and elements of your choosing. Keep adding to it until (one of) the items go over 255. What happens? Hint: make an array, and just add a constant to it. The constant will be added to all the items of the array element-wise.
2. Use a mask to multiply all values below 100 in the following list by 2:

   ```
   >>> a = np.array([230, 10, 284, 39, 76])
   ```

   Repeat this until all values are above 100. (Not manually, but by looping)

   Then, select all values between $150 < a < 200$.