The NumPy library (written by Malte M. Max)

Introduction

NumPy provides Python users with a flexible object to store data in, *ndarray*s (n-dimensional arrays), and includes many useful and powerful mathematical functions that can be performed with these *ndarray*s. Many other Python libraries have been built around the core of NumPy: later in this course, you will learn that much of pandas' capabilities is closely connected to the functionalities of NumPy.

In some respects *ndarray*s are comparable to standard Python lists, but in other respects they are very different. Key differences are:

- *ndarray*s are homogenous, that is, they can only store one data type
- *ndarray*s enable complex mathematical calculations on large amounts of data with (much) less code and faster execution due to a process called vectorization

This document deals with the basics of NumPy as we need it for the purpose of this course. We will be scratching the surface of what is possible with NumPy, and keep background information to a minimum. For more information and further reading, please visit the NumPy user guide (https://docs.scipy.org/doc/numpy/user/index.html).

Note also, we will refer to NumPy's *ndarray*s simply as arrays.

We start by importing NumPy:

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

Importing NumPy as above enables us to use the `np` shortcut when calling NumPy functions below. It is common practice to import NumPy like this.

Creating arrays

Let's start by creating an *ndarray*. There are many ways to do so. You could create a Python list and then pass that list to NumPy's `array` function:

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

```
In [3]: np.array(a)
Out[3]: array([1, 2, 3, 4, 5])
```

Or we use NumPy's `arange` function directly and save the output to `a`:

```
In [4]: a = np.arange(5)
        a
Out[4]: array([0, 1, 2, 3, 4])
```

The above command has created a NumPy array with values running from 0 to 4. We can check its shape and dimensions by:

```
In [5]: a.shape
Out[5]: (5,)

In [6]: a.ndim
Out[6]: 1
```

As you can see, `a` has 5 elements (5,) along one dimension (dimensions are also called axes).

*ndarray*s can also have more than one axis (i.e., they can be more than one-dimensional):

```
In [7]: a = np.array([[0, 1, 2, 3], [4, 5, 6, 7]])
        a
Out[7]: array([[0, 1, 2, 3],
               [4, 5, 6, 7]])
```

Note that we have passed a nested list (i.e., a list - actually, two lists in this case - in a list) to `np.array()`. (You can see that by noting the double brackets at the start (`[[`) and at the end (`]]`) of the input to `np.array()`.)

Checking `a`'s shape yields:

```
In [8]: a.shape
Out[8]: (2, 4)

In [9]: a.ndim
Out[9]: 2
```

You can see above that `a` has 2 axes, where the first axis has 2 elements and the second axis has 4 elements. To make understanding easier, I like to think of `a`'s shape as consisting of 2 rows and 4 columns.

As noted in the beginning, *ndarrray*s are n-dimensional, so we can also create even more complex arrays if we need them:

In [10]:
```
a = np.array([[[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]],
              [[12, 13, 14, 15], [16, 17, 18, 19], [20, 21, 22, 2
3]]])
a
```

Out[10]:
```
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]],

       [[12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]]])
```

In [11]:
```
a.shape
```

Out[11]:
```
(2, 3, 4)
```

In [12]:
```
a.ndim
```

Out[12]:
```
3
```

As you can see, `a` has 3 axes with 2, 3, and 4 elements along the first, second, and third axis, respectively.

You can also use `arange()` to create arrays by passing more arguments to it:

In [13]:
```
np.arange(0, 5, 0.5)
```

Out[13]:
```
array([0. , 0.5, 1. , 1.5, 2. , 2.5, 3. , 3.5, 4. , 4.5])
```

The code above creates an array with data running from 0 to 5 (excluding the 5 itself) and intermediate steps of width 0.5.

We can also let NumPy create an array with evenly spaced distances between each element, using `linspace()`:

In [14]:
```
np.linspace(1, 15, 5)
```

Out[14]:
```
array([ 1. ,  4.5,  8. , 11.5, 15. ])
```

The function above will return an array with 5 evenly spaced distances between 1 and 15.

We can also create arrays with pre-filled values:

```
In [15]: np.zeros(15)
```

```
Out[15]: array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
         0.])
```

`np.zeros()` creates an array of the desired length filled with zeros. Equivalently, `np.ones()` creates an array of ones:

```
In [16]: np.ones(10)
```

```
Out[16]: array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
```

And `np.full()` creates an array of desired size (5), filled with desired values (2):

```
In [17]: np.full(5, 2)
```

```
Out[17]: array([2, 2, 2, 2, 2])
```

NumPy data types

It can be important to know the type of data contained in a *ndarray*. We can obtain this information with NumPy's `dtype` :

```
In [18]: a = np.array([[0, 1, 2, 3], [4, 5, 6, 7]])
         a
```

```
Out[18]: array([[0, 1, 2, 3],
                [4, 5, 6, 7]])
```

Using `dtype` shows that it holds integer values:

```
In [19]: a.dtype
```

```
Out[19]: dtype('int32')
```

Let's create another array:

```
In [20]: b = np.arange(0, 5, 0.5)
         b
```

```
Out[20]: array([0. , 0.5, 1. , 1.5, 2. , 2.5, 3. , 3.5, 4. , 4.5])
```

In [21]:
```
b.dtype
```

Out[21]:
```
dtype('float64')
```

As you can see, the type of the data in `b` is double precision float. *ndarrray*s can also hold other data types, such as strings, Booleans, and many more (check here (https://docs.scipy.org/doc/numpy/user/basics.types.html)).

Reshaping arrays

We can easily reshape arrays using `reshape()`. Let's create an array with values from 0 to 9, with one dimension:

In [22]:
```
a = np.arange(10)
a
```

Out[22]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

By applying `reshape()`, we can cast it into a different shape:

In [23]:
```
a.reshape(2, 5)
```

Out[23]:
```
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
```

Be aware though to assign the reshaped array to an object. If we just execute the above code and then check `a`'s shape, we notice:

In [24]:
```
a.shape
```

Out[24]:
```
(10,)
```

If we want the `reshape()` operation to be saved to an object, we have to assign it to an object. Let's overwrite `a`, and then check its shape:

In [25]:
```
a = a.reshape(2, 5)
a.shape
```

Out[25]:
```
(2, 5)
```

Now you see that `a` has 2 elements on its first axis and 5 on its second axis. It therefore has two dimensions:

In [26]:
```
a.ndim
```

Out[26]:
```
2
```

Another way to achieve that a reshaped object "keeps" the reshaped shape is by using `resize()`. `resize()` will modify the array on which it is applied without the need to overwrite that specific array:

```
In [27]: a = np.arange(10)
         a

Out[27]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
In [28]: a.resize((2, 5))
         a

Out[28]: array([[0, 1, 2, 3, 4],
                [5, 6, 7, 8, 9]])
```

(Note: we do not have to explicitly write `a = a.resize((2, 5))`; `a.resize((2, 5))` suffices.)

Let's take another arrray with 15 numbers:

```
In [29]: a = np.arange(0, 15)
         a

Out[29]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 1
         4])
```

If we try to reshape it into a form with 4 elements on its first axis, and 3 on the second axis, we get an error:

```
In [30]: a.reshape(4, 3)

         ---------------------------------------------------------------
         ---------
         ValueError                                Traceback (most recent c
         all last)
         <ipython-input-30-d86b48f8f711> in <module>
         ----> 1 a.reshape(4, 3)

         ValueError: cannot reshape array of size 15 into shape (4,3)
```

This happens because 4x3 does not work with an array with 15 elements along its axis (a 4x3 array would have 12 elements in total, which is less than the 15 elements of the input array). When we apply:

```
In [31]: a.reshape(5, 3)

Out[31]: array([[ 0,  1,  2],
                [ 3,  4,  5],
                [ 6,  7,  8],
                [ 9, 10, 11],
                [12, 13, 14]])
```

we see that it works because a 5x3 array can fit the 15 elements of the input array.

Especially for larger arrays that you want to reshape, it can be hard to calculate the missing dimension. For this, NumPy can conveniently calculate the missing dimension for us if we supply a `-1` as follows:

In [32]:
```
np.arange(45).reshape(3, -1)
```

Out[32]:
```
array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 1
4],
       [15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 2
9],
       [30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 4
4]])
```

Or the other way around:

In [33]:
```
np.arange(45).reshape(-1, 3)
```

Out[33]:
```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11],
       [12, 13, 14],
       [15, 16, 17],
       [18, 19, 20],
       [21, 22, 23],
       [24, 25, 26],
       [27, 28, 29],
       [30, 31, 32],
       [33, 34, 35],
       [36, 37, 38],
       [39, 40, 41],
       [42, 43, 44]])
```

In case we want to transpose an arrary, we can use either `T` or `transpose()` (both achieve the same here):

In [34]:
```
a = np.arange(10).reshape(2, -1)
a
```

Out[34]:
```
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
```

In [35]:
```
a.T
```

Out[35]:
```
array([[0, 5],
       [1, 6],
       [2, 7],
       [3, 8],
       [4, 9]])
```

In [36]:
```
a.transpose()
```

Out[36]:
```
array([[0, 5],
       [1, 6],
       [2, 7],
       [3, 8],
       [4, 9]])
```

In case we want to convert a n-dimensional array to a one-dimensional array, we can use `ravel()` to flatten the array:

In [37]:
```
a.ravel()
```

Out[37]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [38]:
```
a.ravel().ndim
```

Out[38]:
```
1
```

Integer and slice indexing

Imagine we do not want to work with an entire array, but with parts of it. With NumPy, selecting only parts of an array works similar to indexing/slicing of regular Python lists. Using a two-dimensional array:

In [39]:
```
a = np.arange(20).reshape(4, -1)
a
```

Out[39]:
```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19]])
```

Suppose we need only the first row. We can use integer indexing:

In [40]:
```
a[0]
```

Out[40]:
```
array([0, 1, 2, 3, 4])
```

Or, if we want to select the first element of the second row, again with integer indexing:

In [41]:
```
a[1, 0]
```

Out[41]:
```
5
```

We can also get "slices" of the array with slice indexing:

```
In [42]: a[:2]
```

```
Out[42]: array([[0, 1, 2, 3, 4],
                [5, 6, 7, 8, 9]])
```

The command above returned the first two rows of the array. The colon indicates slice indexing. Note that the code above is equivalent to `a[0:2]`, that is, the `0` before the `:` can be left out.

We can also combine integer and slice indexing:

```
In [43]: a[:, 2]
```

```
Out[43]: array([ 2,  7, 12, 17])
```

The code above returns the second column of `a`. Note that `:` means that all elements along the respective axis are considered.

You can also use negative indices. `-1` returns the first element from the end:

```
In [44]: a[-1]
```

```
Out[44]: array([15, 16, 17, 18, 19])
```

With the code above, we obtained the last row of `a`.

Equivalenty, with the code below, we get the last column of `a`, with all elements along the second axis:

```
In [45]: a[:, -1]
```

```
Out[45]: array([ 4,  9, 14, 19])
```

Boolean indexing

Whereas above we explicitly tell NumPy which elements to select, we can also use Boolean indexing (Boolean values are logical: true or false) to obtain parts of an array that fulfill certain criteria. This is useful when we want to obtain a slice of an arrray based on certain criteria but do not know where in the array that slice is located.

```
In [46]: a = np.arange(0, 40, 2).reshape(4, -1)
         a
```

```
Out[46]: array([[ 0,  2,  4,  6,  8],
                [10, 12, 14, 16, 18],
                [20, 22, 24, 26, 28],
                [30, 32, 34, 36, 38]])
```

We could check which of the values is larger than 2:

```
In [47]: a > 2
```

```
Out[47]: array([[False, False,  True,  True,  True],
                [ True,  True,  True,  True,  True],
                [ True,  True,  True,  True,  True],
                [ True,  True,  True,  True,  True]])
```

You can see that the values that are smaller than 2 receive a `False`; all other values match the logical expression and receive a `True`. You can pass this expression to `a` directly:

```
In [48]: a[a > 2]
```

```
Out[48]: array([ 4,  6,  8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32,
                34, 36,
                38])
```

We could also wish to replace all values that are lower/equal to 14 by 0, and use a slightly more verbose approach:

```
In [49]: b = a <= 14
         b
```

```
Out[49]: array([[ True,  True,  True,  True,  True],
                [ True,  True,  True, False, False],
                [False, False, False, False, False],
                [False, False, False, False, False]])
```

```
In [50]: a[b] = 0
         a
```

```
Out[50]: array([[ 0,  0,  0,  0,  0],
                [ 0,  0,  0, 16, 18],
                [20, 22, 24, 26, 28],
                [30, 32, 34, 36, 38]])
```

In the above cells, we have first created a Boolean array, `b`, containing `True` and `False` for each element of `a` (i.e., a `True` for each element of `a` that is $<= 14$, and a `False` for each element of `a` that is > 14). We then supplied that `b` to `a`, and have assigned all values of `a` that are `True` in `b` the value `0`.

As before, a shorter version of what we have done above would be to supply the conditional statement `a <= 14` directly to `a`, without saving `a <= 14` to the object `b` first (note the second line):

In [51]:
```
a = np.arange(0, 40, 2).reshape(4, -1)
a[a <= 14] = 0
a
```

Out[51]:
```
array([[ 0,  0,  0,  0,  0],
       [ 0,  0,  0, 16, 18],
       [20, 22, 24, 26, 28],
       [30, 32, 34, 36, 38]])
```

Stacking, concatenating and splitting

We can stack separate arrays with `vstack` (vertical stack) or `hstack` (horizontal stack) as follows:

In [52]:
```
a = np.arange(4).reshape(2, 2)
a
```

Out[52]:
```
array([[0, 1],
       [2, 3]])
```

In [53]:
```
b = np.arange(4, 8).reshape(2, 2)
b
```

Out[53]:
```
array([[4, 5],
       [6, 7]])
```

In [54]:
```
np.vstack((a, b))
```

Out[54]:
```
array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7]])
```

In [55]:
```
np.hstack((a, b))
```

Out[55]:
```
array([[0, 1, 4, 5],
       [2, 3, 6, 7]])
```

With `concatenate()`, we can achieve the same results. We only have to specify the axis along which to concatenate: `axis = 0` (along "rows") or `axis = 1` (along "columns"):

In [56]:
```
np.concatenate((a, b), axis = 0)
```

Out[56]:
```
array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7]])
```

In [57]:
```
np.concatenate((a, b), axis = 1)
```

Out[57]:
```
array([[0, 1, 4, 5],
       [2, 3, 6, 7]])
```

We can use `hsplit()` and `vsplit()` to split arrays into smaller parts. `hsplit()` splits along the horizontal axis, `vsplit()` along the vertical axis:

```
In [58]: a = np.arange(16).reshape(-1, 4)
         a
```

```
Out[58]: array([[ 0,  1,  2,  3],
                [ 4,  5,  6,  7],
                [ 8,  9, 10, 11],
                [12, 13, 14, 15]])
```

The code below will split `a` into two equally shaped arrays along the vertical axis:

```
In [59]: np.vsplit(a, 2)
```

```
Out[59]: [array([[0, 1, 2, 3],
                 [4, 5, 6, 7]]), array([[ 8,  9, 10, 11],
                 [12, 13, 14, 15]])]
```

The first resulting array is `[[0, 1, 2, 3], [4, 5, 6, 7]]`, and the second one is: `[[8, 9, 10, 11], [12, 13, 14, 15]]`.

`hsplit()` below achieves the same along the horizontal axis:

```
In [60]: np.hsplit(a, 2)
```

```
Out[60]: [array([[ 0,  1],
                 [ 4,  5],
                 [ 8,  9],
                 [12, 13]]), array([[ 2,  3],
                 [ 6,  7],
                 [10, 11],
                 [14, 15]])]
```

The first resulting array is `[[0, 1], [4, 5], [8, 9], [12, 13]]`, and the second one is: `[[2, 3], [6, 7], [10, 11], [14, 15]]`.

You can alternatively also specify where NumPy should split along the selected axis. The code below will split `a` after the first element on the horizontal axis (i.e., after the first "column"), and then return the rest of `a` as the second array:

In [61]:
```
np.hsplit(a, (1,))
```

Out[61]:
```
[array([[ 0],
        [ 4],
        [ 8],
        [12]]), array([[ 1,  2,  3],
        [ 5,  6,  7],
        [ 9, 10, 11],
        [13, 14, 15]])]
```

First array: `[[0], [4], [8], [12]]`, second array: `[[1, 2, 3], [5, 6, 7], [9, 10, 11], [13, 14, 15]]`.

Calculation methods

Let's use NumPy for what we might be using it for the most: any type of mathematical calculations!

Basic operations

Arithmetic operations are performed **element-wise**. Suppose we have two arrays (`a` and `b`):

In [62]:
```
a = np.array([[20, 30, 40, 50], [60, 70, 80, 90]])
a
```

Out[62]:
```
array([[20, 30, 40, 50],
       [60, 70, 80, 90]])
```

In [63]:
```
b = np.array([[25, 35, 45, 55], [65, 75, 85, 95]])
b
```

Out[63]:
```
array([[25, 35, 45, 55],
       [65, 75, 85, 95]])
```

Let's add `a` and `b` :

In [64]:
```
a + b
```

Out[64]:
```
array([[ 45,  65,  85, 105],
       [125, 145, 165, 185]])
```

Or, divide them:

In [65]:
```
a / b
```

Out[65]:
```
array([[0.8       , 0.85714286, 0.88888889, 0.90909091],
       [0.92307692, 0.93333333, 0.94117647, 0.94736842]])
```

Multiply them:

```
In [66]: a * b
```

```
Out[66]: array([[ 500, 1050, 1800, 2750],
                [3900, 5250, 6800, 8550]])
```

Squared the values in `a` :

```
In [67]: a ** 2
```

```
Out[67]: array([[ 400,  900, 1600, 2500],
                [3600, 4900, 6400, 8100]], dtype=int32)
```

We can also perform calculations on selected parts of an array. If we take `a` from above:

```
In [68]: a
```

```
Out[68]: array([[20, 30, 40, 50],
                [60, 70, 80, 90]])
```

Suppose we want the sum of all elements of the first row:

```
In [69]: a[0, :]
```

```
Out[69]: array([20, 30, 40, 50])
```

```
In [70]: a[0, :].sum()
```

```
Out[70]: 140
```

Or the sum of the second row, but then only of the last two elements of that row:

```
In [71]: a[1, 2:4]
```

```
Out[71]: array([80, 90])
```

```
In [72]: a[1, 2:4].sum()
```

```
Out[72]: 170
```

Universal functions

NumPy also provides many functions out of the box (check here (https://docs.scipy.org/doc/numpy/reference/ufuncs.html) for more information):

For example, if instead of element-wise multiplication, we need the matrix product, we can use `np.dot()`:

In [73]:
```
a = np.arange(20, 100, 10).reshape(2, -1)
a
```

Out[73]:
```
array([[20, 30, 40, 50],
       [60, 70, 80, 90]])
```

In [74]:
```
b = np.arange(25, 105, 10).reshape(-1, 2)
b
```

Out[74]:
```
array([[25, 35],
       [45, 55],
       [65, 75],
       [85, 95]])
```

In [75]:
```
a.dot(b)
```

Out[75]:
```
array([[ 8700, 10100],
       [17500, 20500]])
```

Other examples of NumPy's universal functions are:

In [76]:
```
a = np.arange(-5, 15).reshape(5, -1)
a
```

Out[76]:
```
array([[-5, -4, -3, -2],
       [-1,  0,  1,  2],
       [ 3,  4,  5,  6],
       [ 7,  8,  9, 10],
       [11, 12, 13, 14]])
```

In [77]:
```
np.mean(a)
```

Out[77]:
```
4.5
```

As you can see above, the result is a scalar. `np.mean()` has returned the average of all values in the array. If we instead want to obtain averages along a certain axis, we can specify that axis. For averages along the first axis (along the "rows"):

In [78]:
```
np.mean(a, axis = 0)
```

Out[78]:
```
array([3., 4., 5., 6.])
```

For averages along the second axis (along the "columns"):

In [79]:
```
np.mean(a, axis = 1)
```

Out[79]:
```
array([-3.5,  0.5,  4.5,  8.5, 12.5])
```

You can also call `mean()` like this:

```
In [80]: a.mean(axis = 1)
```

```
Out[80]: array([-3.5,  0.5,  4.5,  8.5, 12.5])
```

There are many, many more functions. Here are some more examples:

Sum:

```
In [81]: np.sum(a)
```

```
Out[81]: 90
```

Multiplication (`a` times `a` is equivalent to taking all elements in `a` to the second power; see `np.power()` below):

```
In [82]: np.multiply(a, a)
```

```
Out[82]: array([[ 25,  16,   9,   4],
                [  1,   0,   1,   4],
                [  9,  16,  25,  36],
                [ 49,  64,  81, 100],
                [121, 144, 169, 196]])
```

Standard deviation (here: along the "columns"):

```
In [83]: np.std(a, axis = 1)
```

```
Out[83]: array([1.11803399, 1.11803399, 1.11803399, 1.11803399, 1.1180339
         9])
```

Minimum (here: along the "rows"):

```
In [84]: np.min(a, axis = 0)
```

```
Out[84]: array([-5, -4, -3, -2])
```

Take all values to the second power (note that we supply `2` to the `np.power()` function below; we could supply any number based on the power to which we want to raise the numbers):

```
In [85]: np.power(a, 2)
```

```
Out[85]: array([[ 25,  16,   9,   4],
                [  1,   0,   1,   4],
                [  9,  16,  25,  36],
                [ 49,  64,  81, 100],
                [121, 144, 169, 196]], dtype=int32)
```

Get the absolute values:

In [86]:
```
np.absolute(a)
```

Out[86]:
```
array([[ 5,  4,  3,  2],
       [ 1,  0,  1,  2],
       [ 3,  4,  5,  6],
       [ 7,  8,  9, 10],
       [11, 12, 13, 14]])
```

Get the natural logarithm of the absolute values $+1$ (Note: `log1p()` takes the input array $+1$. We take this function here because `np.absolute(a)` from above contains a zero. Otherwise, we could just use `np.log()`):

In [87]:
```
np.log1p(np.absolute(a))
```

Out[87]:
```
array([[1.79175947, 1.60943791, 1.38629436, 1.09861229],
       [0.69314718, 0.        , 0.69314718, 1.09861229],
       [1.38629436, 1.60943791, 1.79175947, 1.94591015],
       [2.07944154, 2.19722458, 2.30258509, 2.39789527],
       [2.48490665, 2.56494936, 2.63905733, 2.7080502 ]])
```

Views or copies

In order to ensure fast execution of code, most NumPy operations create **views** of the original array, **and not copies**. For example:

In [88]:
```
a = np.arange(0, 20, 0.5).reshape(-1, 5)
a
```

Out[88]:
```
array([[ 0. ,  0.5,  1. ,  1.5,  2. ],
       [ 2.5,  3. ,  3.5,  4. ,  4.5],
       [ 5. ,  5.5,  6. ,  6.5,  7. ],
       [ 7.5,  8. ,  8.5,  9. ,  9.5],
       [10. , 10.5, 11. , 11.5, 12. ],
       [12.5, 13. , 13.5, 14. , 14.5],
       [15. , 15.5, 16. , 16.5, 17. ],
       [17.5, 18. , 18.5, 19. , 19.5]])
```

Let's take a slice from `a`, store it as `b`, and then make some changes to it:

In [89]:
```
b = a[:, 2]
b
```

Out[89]:
```
array([ 1. ,  3.5,  6. ,  8.5, 11. , 13.5, 16. , 18.5])
```

In [90]:
```
b[b <= 6] = 0
b
```

Out[90]:
```
array([ 0. ,  0. ,  0. ,  8.5, 11. , 13.5, 16. , 18.5])
```

When we call `a` (the array from which we have taken a slice and stored it to `b`), we note that the changes are also reflected there. This happens because `a` and `b` point to **the same object**.

In [91]:
```
a
```

Out[91]:
```
array([[ 0. ,  0.5,  0. ,  1.5,  2. ],
       [ 2.5,  3. ,  0. ,  4. ,  4.5],
       [ 5. ,  5.5,  0. ,  6.5,  7. ],
       [ 7.5,  8. ,  8.5,  9. ,  9.5],
       [10. , 10.5, 11. , 11.5, 12. ],
       [12.5, 13. , 13.5, 14. , 14.5],
       [15. , 15.5, 16. , 16.5, 17. ],
       [17.5, 18. , 18.5, 19. , 19.5]])
```

This is important to keep in mind. To create an actual copy, you can use `np.copy()`.