

# 1. Creating Numpy arrays

Numpy has many different types of data "containers": lists, dictionaries, tuples etc. However none of them allows for efficient numerical calculation, in particular not in multi-dimensional cases (think e.g. of operations on images). Numpy has been developed exactly to fill this gap. It provides a new data structure, the **numpy array**, and a large library of operations that allow to:

- generate such arrays
- combine arrays in different ways (concatenation, stacking etc.)
- modify such arrays (projection, extraction of sub-arrays etc.)
- apply mathematical operations on them

Numpy is the base of almost the entire Python scientific programming stack. Many libraries build on top of Numpy, either by providing specialized functions to operate on them (e.g. scikit-image for image processing) or by creating more complex data containers on top of it. The data science library Pandas that will also be presented in this course is a good example of this with its dataframe structures.

```
In [ ]: import numpy as np
        from svg import numpy_to_svg
```

## 1.1 What is an array ?

Let us create the simplest example of an array by transforming a regular Python list into an array (we will see more advanced ways of creating arrays in the next chapters):

```
In [ ]: mylist = [2,5,3,9,5,2]
```

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

```
Out[3]: [2, 5, 3, 9, 5, 2]
```

```
In [4]: myarray = np.array(mylist)
```

```
In [5]: myarray
```

```
Out[5]: array([2, 5, 3, 9, 5, 2])
```

```
In [6]: type(myarray)
```

```
Out[6]: numpy.ndarray
```

We see that `myarray` is a Numpy array thanks to the `array` specification in the output. The type also says that we have a `numpy ndarray` (n-dimensional). At this point we don't see a big difference with regular lists, but we'll see in the following sections all the operations we can do with these objects.

We can already see a difference with two basic attributes of arrays: their type and shape.

### 1.1.1 Array Type

Just like when we create regular variables in Python, arrays receive a type when created. Unlike regular list, **all** elements of an array always have the same type. The type of an array can be recovered through the `.dtype` method:

```
In [7]: myarray.dtype
```

```
Out[7]: dtype('int64')
```

Depending on the content of the list, the array will have different types. But the logic of "maximal complexity" is kept. For example if we mix integers and floats, we get a float array:

```
In [8]: myarray2 = np.array([1.2, 6, 7.6, 5])
myarray2
```

```
Out[8]: array([1.2, 6. , 7.6, 5. ])
```

```
In [9]: myarray2.dtype
```

```
Out[9]: dtype('float64')
```

In general, we have the possibility to assign a type to an array. This is true here, as well as later when we'll create more complex arrays, and is done via the `dtype` option:

```
In [10]: myarray2 = np.array([1.2, 6, 7.6, 500], dtype=np.uint8)
myarray2
```

```
Out[10]: array([ 1,  6,  7, 244], dtype=uint8)
```

The type of the array can also be changed after creation using the `.astype()` method:

```
In [11]: myfloat_array = np.array([1.2, 6, 7.6, 500], dtype=np.float)
myfloat_array.dtype
```

```
Out[11]: dtype('float64')
```

```
In [12]: myint_array = myfloat_array.astype(np.int8)
myint_array.dtype
```

```
Out[12]: dtype('int8')
```

### 1.1.2 Array shape

A very important property of an array is its **shape** or in other words the dimensions of each axis. That property can be accessed via the `.shape` property:

```
In [13]: myarray
```

```
Out[13]: array([2, 5, 3, 9, 5, 2])
```

```
In [14]: myarray.shape
```

```
Out[14]: (6,)
```

We see that our simple array has only one dimension of length 6. Now of course we can create more complex arrays. Let's create for example a *list of two lists*:

```
In [15]: my2d_list = [[1,2,3], [4,5,6]]  
  
         my2d_array = np.array(my2d_list)  
         my2d_array
```

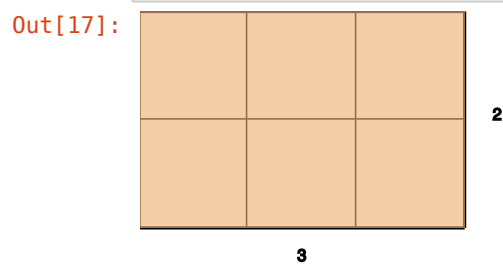
```
Out[15]: array([[1, 2, 3],  
               [4, 5, 6]])
```

```
In [16]: my2d_array.shape
```

```
Out[16]: (2, 3)
```

We see now that the shape of this array is *two-dimensional*. We also see that we have 2 lists of 3 elements. In fact at this point we should forget that we have a list of lists and simply consider this object as a *matrix* with *two rows and three columns*. We'll use the following graphical representation to clarify some concepts:

```
In [17]: numpy_to_svg(my2d_array)
```



## 1.2 Creating arrays

We have seen that we can turn regular lists into arrays. However this becomes quickly impractical for larger arrays. Numpy offers several functions to create particular arrays.

### 1.2.1 Common simple arrays

For example an array full of zeros or ones:

```
In [18]: one_array = np.ones((2,3))  
         one_array
```

```
Out[18]: array([[1., 1., 1.],  
               [1., 1., 1.]])
```

```
In [19]: zero_array = np.zeros((2,3))  
         zero_array
```

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

One can also create diagonal matrix:

```
In [20]: np.eye(3)
Out[20]: array([[1., 0., 0.],
               [0., 1., 0.],
               [0., 0., 1.]])
```

By default Numpy creates float arrays:

```
In [21]: one_array.dtype
Out[21]: dtype('float64')
```

However as mentioned before, one can impose a type using the `dtype` option:

```
In [22]: one_array_int = np.ones((2,3), dtype=np.int8)
          one_array_int
Out[22]: array([[1, 1, 1],
               [1, 1, 1]], dtype=int8)

In [23]: one_array_int.dtype
Out[23]: dtype('int8')
```

## 1.2.2 Copying the shape

Often one needs to create arrays of same shape. This can be done with "like-functions":

```
In [24]: same_shape_array = np.zeros_like(one_array)
          same_shape_array
Out[24]: array([[0., 0., 0.],
               [0., 0., 0.]])

In [25]: one_array.shape
Out[25]: (2, 3)

In [26]: same_shape_array.shape
Out[26]: (2, 3)

In [27]: np.ones_like(one_array)
Out[27]: array([[1., 1., 1.],
               [1., 1., 1.]])
```

## 1.2.3 Complex arrays

We are not limited to create arrays containing ones or zeros. Very common operations involve e.g. the creation of arrays containing regularly arranged numbers. For example a "from-to-by-step" list:

```
In [28]: np.arange(0, 10, 2)
Out[28]: array([0, 2, 4, 6, 8])
```

Or equidistant numbers between boundaries:

```
In [29]: np.linspace(0,1, 10)
Out[29]: array([0.          , 0.11111111, 0.22222222, 0.33333333, 0.44444444,
                0.55555556, 0.66666667, 0.77777778, 0.88888889, 1.          ])
```

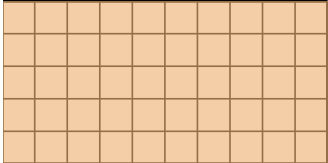
Numpy offers in particular a `random` submodule that allows one to create arrays containing values from a wide array of distributions. For example, normally distributed:

```
In [30]: normal_array = np.random.normal(loc=10, scale=2, size=(3,4))
         normal_array
Out[30]: array([[16.64156121, 13.38970093, 11.32772287,  7.93713055],
                [ 8.33365707, 11.27817138,  9.81766403, 11.11541451],
                [12.97743479,  7.1622948 , 12.02417108,  8.64402656]])

In [31]: np.random.poisson(lam=5, size=(3,4))
Out[31]: array([[4, 4, 2, 4],
                [3, 7, 6, 3],
                [6, 5, 5, 4]])
```

## 1.2.4 Higher dimensions

Until now we have almost only dealt with 1D or 2D arrays that look like a simple grid:

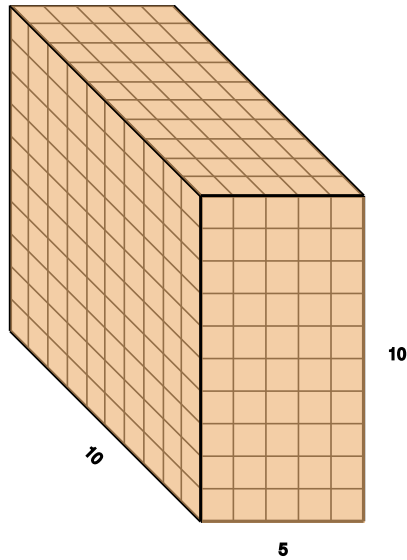
```
In [32]: myarray = np.ones((5,10))
         numpy_to_svg(myarray)
Out[32]: 
         10
```

We are not limited to create 1 or 2 dimensional arrays. We can basically create any-dimension array. For example in microscopy, images can be volumetric and thus they are 3D arrays in Numpy. For example if we acquired 5 planes of a 10px by 10px image, we would have something like:

```
In [33]: array3D = np.ones((10,10,5))
```

```
In [34]: numpy_to_svg(array3D)
```

```
Out[34]:
```



All the functions and properties that we have seen until now are N-dimensional, i.e. they work in the same way irrespective of the array size.

## 1.3 Importing arrays

We have seen until now multiple ways to create arrays. However, most of the time, you will *import* data from some source, either directly as arrays or as lists, and use these data in your analysis.

### 1.3.1 Loading and saving arrays

Numpy can efficiently save and load arrays in its own format `.npy`. Let's create an array and save it:

```
In [35]: array_to_save = np.random.normal(10, 2, (4,5))  
array_to_save
```

```
Out[35]: array([[ 5.41052227, 11.78370736,  9.22402365,  9.91645679,  9.48495895],  
                [10.10853493,  8.75839699,  8.26026504, 12.51736441,  9.80407577],  
                [10.09084097,  7.27962072, 11.05963249, 14.37978527,  9.00654627],  
                [ 6.01521954, 10.25115807, 10.28647927, 10.12389832,  8.91184397]])
```

```
In [36]: np.save('my_saved_array.npy', array_to_save)
```

In [37]: `ls`

```
01-DA_Numpy_arrays_creation.ipynb  98-DA_Numpy_Solutions.ipynb
02-DA_Numpy_array_maths.ipynb     99-DA_Pandas_Exercises.ipynb
03-DA_Numpy_matplotlib.ipynb      99-DA_Pandas_Solutions.ipynb
04-DA_Numpy_indexing.ipynb        My_first_plot.png
05-DA_Numpy_combining_arrays.ipynb SNSF_data.ipynb
06-DA_Pandas_introduction.ipynb    Untitled.ipynb
07-DA_Pandas_structures.ipynb     __pycache__/
08-DA_Pandas_import.ipynb         ipyleaflet.ipynb
09-DA_Pandas_operations.ipynb     multiple_arrays.npz
10-DA_Pandas_combine.ipynb        my_saved_array.npy
11-DA_Pandas_splitting.ipynb      raw.githubusercontent.com/
12-DA_Pandas_plotting.ipynb       svg.py
13-DA_Pandas_ML.ipynb             unused/
98-DA_Numpy_Exercises.ipynb
```

Now that this array is saved on disk, we can load it again using `np.load` :

In [38]: `new_array = np.load('my_saved_array.npy')`  
`new_array`

Out[38]: `array([[ 5.41052227, 11.78370736, 9.22402365, 9.91645679, 9.48495895],  
[10.10853493, 8.75839699, 8.26026504, 12.51736441, 9.80407577],  
[10.09084097, 7.27962072, 11.05963249, 14.37978527, 9.00654627],  
[ 6.01521954, 10.25115807, 10.28647927, 10.12389832, 8.91184397]])`

If you have several arrays that belong together, you can also save them in a single file using `np.savez` in `npz` format.  
Let's create a second array:

In [39]: `array_to_save2 = np.random.normal(10, 2, (1,2))`  
`array_to_save2`

Out[39]: `array([[14.57759687, 7.62340049]])`

In [40]: `np.savez('multiple_arrays.npz', array_to_save=array_to_save, array_to_save2=`  
`array_to_save2)`

In [41]: `ls`

```
01-DA_Numpy_arrays_creation.ipynb  98-DA_Numpy_Solutions.ipynb
02-DA_Numpy_array_maths.ipynb     99-DA_Pandas_Exercises.ipynb
03-DA_Numpy_matplotlib.ipynb      99-DA_Pandas_Solutions.ipynb
04-DA_Numpy_indexing.ipynb        My_first_plot.png
05-DA_Numpy_combining_arrays.ipynb SNSF_data.ipynb
06-DA_Pandas_introduction.ipynb    Untitled.ipynb
07-DA_Pandas_structures.ipynb     __pycache__/
08-DA_Pandas_import.ipynb         ipyleaflet.ipynb
09-DA_Pandas_operations.ipynb     multiple_arrays.npz
10-DA_Pandas_combine.ipynb        my_saved_array.npy
11-DA_Pandas_splitting.ipynb      raw.githubusercontent.com/
12-DA_Pandas_plotting.ipynb       svg.py
13-DA_Pandas_ML.ipynb             unused/
98-DA_Numpy_Exercises.ipynb
```

And when we load it again:

```
In [42]: load_multiple = np.load('multiple_arrays.npz')
         type(load_multiple)
```

```
Out[42]: numpy.lib.npyio.NpzFile
```

We get here an `NpzFile` object from which we can read our data. Note that when we load an `npz` file, it is only loaded *lazily*, i.e. data are not actually read, but the content is parsed. This is very useful if you need to store large amounts of data but don't always need to re-load all of them. We can use methods to actually access the data:

```
In [43]: load_multiple.files
```

```
Out[43]: ['array_to_save', 'array_to_save2']
```

```
In [44]: load_multiple.get('array_to_save2')
```

```
Out[44]: array([[14.57759687,  7.62340049]])
```

### 1.3.2 Importing data as arrays

Images are a typical example of data that are array-like (matrix of pixels) and that can be imported directly as arrays. Of course, each domain will have its own *importing libraries*. For example in the area of imaging, the `scikit-image` package is one of the main libraries, and it offers an importer of images as arrays which works both with local files and web addresses:

```
In [45]: import skimage.io
```

```
image = skimage.io.imread('https://upload.wikimedia.org/wikipedia/commons/f/fd/%27%C3%9Cbermut_Exub%C3%A9rance%27_by_Paul_Klee%2C_1939.jpg')
```

We can briefly explore that image:

```
In [46]: type(image)
```

```
Out[46]: numpy.ndarray
```

```
In [47]: image.dtype
```

```
Out[47]: dtype('uint8')
```

```
In [48]: image.shape
```

```
Out[48]: (584, 756, 3)
```

We see that we have an array of integers with 3 dimensions. Since we imported a jpg image, we know that the third dimension corresponds to three color channels Red, Green, Blue (RGB).

You can also read regular CSV files directly as Numpy arrays. This is more commonly done using Pandas, so we don't spend much time on this, but here is an example on importing data from the web:

```
In [49]: oilprice = np.loadtxt('https://raw.githubusercontent.com/guiwitz/Rdatasets/master/csv/quantreg/gasprice.csv',
                               delimiter=',', usecols=range(2,3), skiprows=1)
```



In [50]: oilprice

```
Out[50]: array([126.6, 127.2, 132.1, 133.3, 133.9, 134.5, 133.9, 133.4, 132.8,
132.3, 131.1, 134.1, 119.2, 116.8, 113.9, 110.6, 107.8, 105.4,
102.5, 104.5, 104.3, 104.7, 105.2, 106.6, 106.9, 109. , 110.4,
111.3, 112.1, 112.9, 114. , 113.8, 113.5, 112.6, 111.4, 110.4,
109.8, 109.4, 109.1, 109.1, 109.9, 111.2, 112.4, 112.4, 112.7,
112. , 111. , 109.7, 109.2, 108.9, 108.4, 108.8, 109.1, 109.1,
110.2, 110.4, 109.9, 109.9, 109.1, 107.5, 106.3, 105.3, 104.2,
102.6, 101.4, 100.6, 99.5, 100.4, 101.1, 101.4, 101.2, 101.3,
101. , 101.5, 101.3, 102.6, 105.1, 105.8, 107.2, 108.9, 110.2,
111.8, 112. , 112.8, 114.3, 115.1, 115.3, 114.9, 114.7, 113.9,
113.2, 112.8, 112.6, 112.3, 111.6, 112.3, 112.1, 112.1, 112.4,
112.3, 111.8, 111.5, 111.5, 111.3, 111.3, 112. , 112. , 111.2,
110.6, 109.8, 108.9, 107.8, 107.4, 106.9, 106.5, 106.6, 106.1,
105.5, 105.5, 106.2, 105.3, 104.7, 104.2, 104.8, 105.8, 105.6,
105.7, 106.8, 107.9, 107.9, 108.6, 108.6, 109.7, 110.6, 110.6,
110.7, 110.4, 110.1, 109.5, 108.9, 108.6, 108.1, 107.5, 106.9,
106.2, 106. , 105.9, 106.5, 106.2, 105.5, 105.1, 104.5, 104.7,
109.2, 109. , 109.3, 109.2, 108.4, 107.5, 106.4, 105.8, 105.1,
103.6, 101.8, 100.3, 99.9, 99.2, 99.5, 100.1, 99.9, 100.5,
100.7, 101.6, 100.9, 100.4, 100.7, 100.5, 100.7, 101.2, 101.1,
102.8, 103.3, 103.7, 104. , 104.5, 104.6, 105. , 105.6, 106.5,
107.3, 107.9, 109.5, 109.7, 110.3, 110.9, 111.4, 113. , 115.7,
116.1, 116.5, 116.1, 115.6, 115. , 114. , 112.9, 112. , 111.4,
110.6, 110.7, 112.1, 112.3, 112.2, 111.3, 108.2, 107.5, 106.4,
105.6, 104.4, 106.3, 107. , 106.2, 106.8, 106.8, 106.2, 105.8,
105.2, 106. , 106.3, 105.6, 105.5, 106.3, 107.7, 109.4, 111. ,
113.3, 114.1, 116.4, 117.3, 119.1, 119.3, 119.4, 119. , 118.3,
117.7, 116.9, 115.9, 114.8, 113.8, 112.6, 112.4, 112.1, 112.2,
111.3, 111.1, 110.7, 110.6, 110.6, 110. , 109.2, 108.1, 107.3,
106.2, 106. , 105.9, 105.6, 105.7, 105.8, 105.7, 107.2, 107.5,
107.7, 108.6, 109.2, 108.4, 107.9, 107.6, 107.3, 107.8, 109.9,
111.5, 111.6, 112.8, 115.8, 117.2, 119.5, 123.4, 124.3, 125.7,
125.9, 126.2, 126.9, 126. , 125.2, 124.7, 124.1, 123. , 121.9,
121.7, 121.5, 121.5, 120.9, 119.9, 119.6, 119.9, 120.1, 119.3,
120.1, 120.3, 120.3, 119.9, 119.1, 120.3, 120.5, 121.7, 122.5,
122.9, 123.8, 124.6, 124.2, 124.1, 123.3, 122.7, 122.4, 122. ,
123.5, 123.6, 123.2, 123. , 122.7, 122. , 121.7, 120.8, 119.9,
119.1, 119.6, 119.1, 119.2, 118.7, 118.8, 118.5, 118.2, 118.2,
119.5, 120.4, 120.6, 119.8, 118.9, 117.9, 117.1, 116.9, 116.5,
117. , 116.4, 118.5, 121.9, 121.8, 123. , 122.9, 122.7, 121.9,
120.8, 119.5, 119.5, 118.7, 117.8, 116.8, 116.3, 116.4, 115.6,
115. , 114. , 112.8, 111.8, 110.8, 109.9, 108.9, 108.3, 107.2,
105.5, 105.1, 104.5, 103.2, 103.8, 102.5, 101.7, 100.6, 99.8,
102.6, 102.3, 101.8, 102.1, 103.2, 103.8, 105.2, 105.5, 105.2,
104.7, 106. , 104.9, 104.1, 104.2, 104.1, 103.7, 104.4, 103.5,
102.3, 101.8, 101.1, 100.4, 99.8, 99.1, 98.7, 99.9, 99.9,
100.6, 101. , 100.7, 100.1, 99.7, 99.4, 98.1, 97.1, 95.4,
93.3, 92.3, 92.1, 91.4, 91.3, 92. , 92.1, 91.3, 90.8,
90.7, 89.9, 88.5, 89.1, 90. , 95.8, 99.9, 105.5, 108.7,
110.7, 110.3, 109.9, 110.7, 110.9, 111.2, 110.1, 108.8, 109.2,
108.8, 110.5, 109.5, 111. , 112.3, 114.8, 117.2, 117.2, 118.3,
121.4, 121.2, 121.4, 122.3, 123.4, 125.2, 124.8, 124.2, 123.4,
122. , 122.5, 121.8, 122.2, 124. , 125.8, 126.2, 126. , 126.3,
125.7, 126.3, 126. , 125.2, 126.8, 130.7, 130.7, 131.9, 135. ,
140. , 141.3, 149. , 151.1, 150.8, 148.4, 147.8, 144.7, 141.5,
140.6, 138.6, 142.7, 146.6, 149.4, 150.9, 153.5, 160.7, 166.4,
164.1, 160.6, 157.1, 152.1, 149.9, 144.7, 143.7, 142. , 144.4,
145.6, 150.2, 153.5, 153.9, 152.5, 149.8, 147.3, 151.6, 153.2,
152.3, 150.2, 150.1, 148.7, 148.9, 146.4, 142.5, 139.6, 138.8,
137.7, 140. , 145.8, 145.6, 144.6, 142.6, 146. , 142.9, 141. ,
139.3, 138.7, 137.7, 137.9, 141.1, 146.9, 153.5, 158.6, 158.5,
165.9, 166.3, 163.7, 165.6, 163. , 158. , 152.6, 145.4, 138.4,
135. , 133. , 131.8, 131.9, 131.9, 134.7, 139.9, 148. , 153.8,
151.1, 151.6, 146. , 138.1, 131. , 126.4, 122.1, 119.3, 117. ,
114.7, 114. , 109.7, 108.4, 107.5, 104.2, 106.3, 109.6, 110.9,
109.9, 108.7, 108.1, 109.8, 108.5, 108.9, 108.7, 111.8, 119.4,
126.2, 130.8, 133.9, 138.2, 136.8, 136.7, 135.3, 135.6, 134.9,
136. , 134.8, 135.3, 133.2, 133.5, 134.2, 135.7, 134.5, 136.1,
```

```
138.1, 137.6, 135.5, 135.5, 135.7, 136.5, 135.3, 135.5, 136.7,  
135.7, 138.5, 141.6, 142.2, 144.3, 142.7, 142.7, 140.6, 137. ,  
133.6, 131.6, 131.6, 132.2, 137.1, 141.7, 141.2, 142.3, 142.2,  
143.7, 149.9, 158.2, 163. , 161.7, 164.1, 166.3, 167.3, 162.6,  
157.7, 155.7, 152.1, 150.4, 148.6, 144.1, 142.7, 144.4, 143.9,  
142.8, 145.6, 148. , 145.1, 144.3, 144.8, 148.9, 149.6, 148.8,  
151.6, 155. , 159.4, 169.3, 168.8, 165.3, 163.6, 158. , 152.4,  
151.1, 151.5, 152.7, 149.9, 149.4, 146.4, 145.9, 147.8, 145.4,  
144.1, 143.3, 145.9, 145.4, 149.2, 154.4, 157.9, 160.4, 159.1,  
160.9, 161.7])
```

In [ ]: