PW 3: Arrays, NumPy library

1 - NumPy library

1.1 - Definition

- **NumPy** is a Python digital library for handling arrays. It allows you to work with the mathematical functions used with these tables.

1.2 - Using NumPy

- **NumPy** is a python library. It must be imported before it can be used. For ease of use, it will be imported under the alias `np` :

  ```
  import numpy as np
  ```

- This import is done only once. It is thus generally placed at the start of the program or the notebook.
- The elements of the library (functions, constants, *etc*), are called via the keyword `np`

  ```
      print(np.sqrt(x))
      print(np.pi)
  ```

- For a simple writing of algorithms, many mathematical functions have been implemented. For more details on the basic functions of **NumPy**, see https://docs.scipy.org/doc/numpy/reference/ufuncs.html#available-ufuncs (https://docs.scipy.org/doc/numpy/reference/ufuncs.html#available-ufuncs). Some constants (like pi) are also defined.

Exercise: Using the features of the **NumPy** library, calculate and display:

- π
- The square root of 9.
- the tangent of π, sine and cosine of $\frac{\pi}{4}$.
- e the base of the logarithm, the logarithm of e and the exponential and the logarithm of 1.

Exercise :

In [5]:
```
import numpy as np

print (np.pi)
print (np.sqrt(9))
print (np.tan(np.pi))
print (np.sin(np.pi/4))
print (np.cos(np.pi/4))
print (np.e)
print (np.log(np.e))
print (np.exp(1))
print (np.log(1))
```

```
3.141592653589793
3.0
-1.2246467991473532e-16
0.7071067811865475
0.7071067811865476
2.718281828459045
1.0
2.718281828459045
0.0
```

2 - Arrays

- One of the main features of the **NumPy** library is the arrays creation and manipulation.
- An array is a new type named **numpy.ndarray** defined by this library.
- Tables are very useful tools for programming, for scientific calculation and data management:
  - they can for example be used with the functions defined above;
  - they can be used to store data from files;
  - they can be used to save and read data in files.

2.1 - Definitions

Several ways exist to define a **Numpy** Array (non-exhaustive list):

- using the `array` function. The `list` from which the array is taken, is one of the arguments.

```
lis = [1, 2, 3]
tab = np.array(lis)
# or, more directly,
tab = np.array([1, 2, 3])
```

- using the function `zeros` or `ones` . The **integer** representing the number of elements of the array (its length) is given as an argument.

```
np.zeros(20)
np.ones(20)
```

- using the function `arange` . The starting value `N1` (included), the end value `N2` (excluded) and the step `p` , which can be integer or real, are given as arguments.

```
tab = np.arange(N1, N2, p)
```

- `N1` and `p` are optional and will default to 0 and 1, respectively, if not given. Otherwise you will have to specify these values. For an array starting with `2` and ending with `N2` (excluded value) with a step of `2` , for example, we will have:

```
tab = np.arange(2, N2, 2)
```

Exercises: Execute the different cells and create a markdown cell below to explain the functionnality and what the short code is doing:

In [6]:
```
lis1 = [6, 7, 8, 9, 10]
print("lis1 =", lis1)
a1 = np.array(lis1)
print("a1 =",a1)
```

```
lis1 = [6, 7, 8, 9, 10]
a1 = [ 6  7  8  9 10]
```

In this code, a list of integers is created and put in a variable. Then, thanks to this list stored in a variable, an array is created as well with the function np.array.

In [7]:
```
a2 = np.array([1, 2, 3, 4, 5])
print("a2 =", a2)
```
```
a2 = [1 2 3 4 5]
```

This code looks like the previous one. The difference is that the list of integers is directly put in the function array so there is not variable created for this list.

In [8]:
```
a3 = np.array([True, True, False])
print("a3 =", a3)
```
```
a3 = [ True  True False]
```

This code looks like the previous ones. The difference is that it is a boolean array.

In [9]:
```
a4 = np.zeros(20)
print("a4 =", a4)
```
```
a4 = [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0.]
```

In this code, an array composed only of zeros is created thanks to the function np.zeros. The argument, 20, represents the number of elements of the array.

In [10]:
```
a5 = np.ones(100)
print("a5 =", a5)
```
```
a5 = [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
1. 1. 1.
 1. 1. 1. 1.]
```

In this code, almost as the previous one, an array composed only of onee is created thanks to the function np.ones. The argument, 100, represents the number of elements of the array.

In [13]:
```
a6 = np.arange(11)
print("a6 =", a6)
```
```
a6 = [ 0  1  2  3  4  5  6  7  8  9 10]
```

In this code an array is created thanks to the function arrange. This function takes in argument the starting value, the end value (which is excluded), and a step so that values from the starting one to the end one with a defined step are included in the array. The starting value and the step are not specified here so it is default to 0 and 1, respectively. Only the end value is given and it equals to eleven. So, we have here an array composed of the integers from 0 to ten.

```
In [12]: a7 = np.arange(0, 1, 0.2)
         print("a7 =", a7)
```

```
a7 = [0.  0.2 0.4 0.6 0.8]
```

This code looks like the previous one but here the starting and end values are specified. Thus, we have an array composed of the floats from zero to one with a step of 0.2.

2.2 - Operations

- One of the great advantages of **NumPy** is that the functions presented in section 1 are applicable to arrays. We can thus apply the function on the array as a whole rather than element by element.
- The result returned by the function is itself an array of the same length as the argument array of the function. The values in this array are the results of the operation on each of the elements,row by row.
- The usual operations (addition, subtraction, multiplication, etc.) are also applicable to tables.

Exercises: Execute the different cells and create a markdown cell below to explain the functionnality and what the short code is doing:

```
In [14]: b1 = 4 * np.ones(5)
         print("b1 =", b1)
```

```
b1 = [4. 4. 4. 4. 4.]
```

In this code, we take an array composed of five one. Then we multiply the whole array. Thus, the array is finally composed of five four.

```
In [18]: b2 = np.sqrt(b1)
         print("b2 =", b2)
```

```
b2 = [2. 2. 2. 2. 2.]
```

In this code, we create a new array composed of the square roots of the elements of the array b1. So, we have an array composed of five elements which the value is two.

```
In [19]: b3 = b1 + b2
         print("b3 =", b3)
```

```
b3 = [6. 6. 6. 6. 6.]
```

In this code, we create a new array whose elements are the sum of the elements of the two previous arrays. Thus, we have an array composed of five elements whose value is six.

```
In [17]: b4 = b1 * b2
         print("b4 =", b4)
```

```
b4 = [8. 8. 8. 8. 8.]
```

In this code, we create a new array whose elements are the product of the elements of the two first arrays. Thus, we have an array composed of five elements whose value is six.

2.3 - Input / output file

- It is possible to save an array **NumPy** in a binary file with the extension **.npy**. Conversely, the content of files with the **.npy** extension can be read and stored in **NumPy** arrays. For these operations, the `save` and `load` functions are used.

```
    np.save("name.npy",array1)
    array2 = np.load("name.npy")
```

Exercise:

- save the array `b4` in a file caled **temp.npy**
- load and display **temp.npy** in an array `b5`
- Check the presence of the file thus created in the Jupyterlab file browser (the left column). Be careful, do not try to open this binary file directly by clicking on it!

```
In [20]: np.save("temp.npy", b4)
         b5 = np.load("temp.npy")
```

- Although the binary format is more efficient for storing large arrays, it is actually often more convenient, if the size of the tables is reasonable, to save them in text format.
- As for the binary, the **NumPy** library offers a function for writing a file in text format `savetxt` and one for reading `loadtxt` .
- The file extension used in this case is usually **.txt** or **.dat**.

Exercise: Repeat the previous exercise but this time using the text format (with the file extension **.txt**).

```
In [21]: np.savetxt("temp.npy", b4)
         b6 = np.loadtxt("temp.npy")
```

2.4 - Two-dimensional arrays

The **NumPy** library also allows you to handle 2-dimensional arrays. Note: there are matrix type variables, `numpy.matrix` , which should not be confused with these arrays. The method of creating 2-dimensional arrays is very similar to that of one-dimensional arrays. The same function - `array` - is used, only the arguments are different.

- For a n rows and p columns array, it is necessary to give as argument of the function `array` a list of n lists, each of length p. This is an example fo an array of 2 rows and 3 colums. Pay attention to the succession `([ [ ],[ ] ])` . The order is always the same; **the number (or index) of rows first, the number (or index) of column, second**.

```
pi=np.pi
neper=np.e
arr1 = np.array([[1, 2, 3], [pi, neper, 7]])
```

- with the `zeros` or `ones` functions, the argument is a list that contains the number of rows and the number of columns (in that order).

```
dim = [20, 10]
arr2 = np.zeros(dim)
arr3 = np.ones(dim)
```

- The `reshape` function allows you to reorganise the display of the elements of an array by varying its dimensions. The `reshape` 's arguments are: the array to modify, the list of dimensions of the new array.

```
arr4 = np.zeros(200)
dim = [20, 10]
arr5 = np.reshape(arr4, dim)
```

- `reshape` considers that all the elements of the array form a single sequence which is broken down into pieces of length the number of columns. The total number of elements remains constant.

Exercises: In different cells:

- Create an array `c1` with 3 rows and 4 columns using the function `array` and a list. Display them.
- Create an array with 3 rows and 2 columns in a variable `c2` using the `zeros` function. Display it.
- Transform the previous 2 arrays respectively into a `c3` array with 4 rows and 3 columns and a `c4` with two rows and three columns. Display them. Can we transform `c1` into an array with two lines and three columns?

In [23]:
```
c1 = ([[1, 2, 3, 4],[np.pi, 0, 0, np.e],[np.e, 1, 1, np.e]])
print(c1)
c2 = np.zeros([3, 2])
print(c2)
```

```
[[1, 2, 3, 4], [3.141592653589793, 0, 0, 2.718281828459045], [2.
718281828459045, 1, 1, 2.718281828459045]]
[[0. 0.]
 [0. 0.]
 [0. 0.]]
```

In [24]:
```
c3 = np.reshape(c1, [4,3])
print(c3)
c4 = np.reshape(c2, [2,3])
print(c4)
```

```
[[1.         2.         3.        ]
 [4.         3.14159265 0.        ]
 [0.         2.71828183 2.71828183]
 [1.         1.         2.71828183]]
[[0. 0. 0.]
 [0. 0. 0.]]
```

No, we can not transform c1 into an array with two lines and three columns because it will change the total number of elements but this one has to remain constant.

2.5 - Random numbers

- The NumPy library is equiped with a random number generator, a sub-library of NumPy, called **random**. It allows you to draw numbers randomly according to certain distributions. Several functions can be used, the main ones being:
  - **randint** which allows to draw N integers between 2 limits (the upper limit being excluded).
  - **random** which allows to draw N reals between 0 (included) and 1 (excluded).
  - **uniform** which allows to draw N real values between 2 limits (the upper limit being excluded).

```
arr4 = np.random.randint(lowerbound, upperbound, N)
arr5 = np.random.random(N)
arr6 = np.random.uniform(lowerbound, upperbound, N)
```

- If the number `N` is given as a list of two integers, **NumPy** generates a two-dimensional array of lengths corresponding to the values in the list.

```
arr7 = np.random.randint(lowerbound, upperbound, [3, 4])
```

- For more information on the module **random**, see https://docs.scipy.org/doc/numpy-1.14.0/reference/routines.random.html (https://docs.scipy.org/doc/numpy-1.14.0/reference/routines.random.html)

Exercises:

- create and display an array `ran1` of 10 random reals between 0 included and 1 excluded.
- create and display an array `ran2` of 10 real numbers between 0 included and 100 excluded.
- create and display a two-dimensional `ran3` array of 5$\times$5 integers between 0 inclusive and 100 excluded.

Note : run cells multiple times and see the value changes.

In [30]:
```
ran1 = np.random.random(10)
print(ran1)
ran2 = np.random.uniform(0, 100, 10)
print(ran2)
ran3 = np.random.randint(0, 100, [5, 5])
print(ran3)
```

```
[0.31826233 0.1835616  0.71608789 0.16778205 0.69190221 0.442843
93
 0.20951574 0.42052188 0.54796263 0.72913621]
[84.53661105 80.82814621 36.559267   33.13914766 20.23751515 26.
59102269
 44.53090186 59.66551009 61.74971332 79.26694627]
[[63 14 47 89 53]
 [ 0 47 78 97 69]
 [48 82 46 21 22]
 [85 40 44  6 52]
 [35 49 69 37 74]]
```

2.6 Indexing arrays

- An element of index `i` of an one dimensional **NumPy** array is selected by writing the name of the array followed by the index of the element enclosed in square brackets.
- It is also possible to extract a sub-array included between the indices `i1` (included) and `i2` (excluded) by separating them with `:` .
  - By default, if no value of `i1` is given, it is set to the value 0. If no value for `i2` is given, it is set by default to the length of the array.
  -
    - The last value of the array can be indexed with the index -1, the penultimate with -2 and so on.

```
print(arr[0]) # displays the first element of an array
print(arr[i1:i2]) # displays elements between indices i1 (inc
luded) and i2 (excluded)
print(arr[:i2]) # displays elements between indices 0 (includ
ed) and i2 (excluded)
print(arr[i1:]) # displays all array elements from index i1
(included) and above
```

Exercise: Define an array of 10 integers of your choice `d1` . Display different sub array for different values of i1 and i2 (including -1 and -2). What is the equivalent of the statement: `print (d1 [:])` (answer in a cell).

In [34]:
```
d1 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(d1)
print(d1[-1])
print(d1[0])
print(d1[2:5])
print(d1[-4:-1])
print(d1[-2:])
print(d1[:6])
print(d1[:])
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
10
1
[3, 4, 5]
[7, 8, 9]
[9, 10]
[1, 2, 3, 4, 5, 6]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

The statement `print (d1 [:])` is the equivalent of `print (d1)` .

- It is also possible to index 2-dimensional arrays following the same principle.
- The indexation by dimension is then separated by a `,`
- we also indicate to take all the elements of a row or a column with `:`.

```
tab_2d[l1, c1] # displays the element at the intersection of the row
with indexs l1 and the column with index c1
tab_2d[l1:l2, c1:c2] # displays the sub-array between the lines l1
(included) and l2 (excluded) and the columns c1 (included) and c2 (e
xcluded)
tab_2d[l1, :] # displays all the columns of the index row l1
tab_2d[:, c1] # displays all the rows of the column with index c1.
```

Exercise:

- Display the second element of the third row of the array `ran3`.
- Display the antepenultimate row of the array `ran3`.
- Display the antepenultimate column of the `ran3` table.
- Store and display a variable `d2` the two-dimensional array corresponding to the last two rows and last two columns of the array `ran3`.

In [38]:
```
print(ran3[2, 1])
print(ran3[-3, :])
print(ran3[:, -3])
d2 = ran3[-2:, -2:]
print(d2)
```

```
82
[48 82 46 21 22]
[47 78 46 44 69]
[[ 6 52]
 [37 74]]
```

2.7 Arrays manipulation

- The library **NumPy** proposes a set of functions acting directly on the totality of a table (with 1 or more dimensions.
- Sum and product:
  - The `sum` and `prod` functions. They take an array as argument and calculate and return the sum and the product of its elements, respectively.
  - In the case of a two-dimensional array, it is possible to pass the optional argument `axis` which will take the value 0 to calculate the sum (or the product, depending on the function used) of the rows, and 1 to perform the calculations on the columns.

    Example:

    ```
    np.sum (array, axis = 0)
    ```

Exercise: Execute the different cells and create a markdown cell below to explain the functionnality and what the short code is doing:

```
In [36]: m1 = np.arange(1, 11)
         print(np.sum(m1))
         print(np.prod(m1))
```

```
55
3628800
```

In this code, we created an array composed of the integers from one to ten. Then, thanks to np.sum and np.prod, we calculate the sum and the products of all the elements of the array.

```
In [37]: m2 = np.arange(15)
         m2 = np.reshape(m2, [3, 5])
         print(m2)
         print(np.sum(m2, axis=0))
         print(np.sum(m2, axis=1))
         print(np.sum(m2))
```

```
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]]
[15 18 21 24 27]
[10 35 60]
105
```

In this code, at firt we create an array composed of the integers from zero to fourteen. Then we reshape the array into a matrix of three rows and two columns. Then, we calculate the sum of the elements of each column and then the sum of the elements of each row. (Besides, i think there is a mistake because when axis equals to zero, it calculates the sum of each column instead of each row and the opposite happens when axis equals to one). Finally, we calculate the sum of all the elements of the array.