0. Import Numpy

```
import numpy as np
```

1. How to create a numpy array?

This function will create an array of zeros: `np.zeros()`.

```
a = np.zeros(3)
print(a)
```

```
[0. 0. 0.]
```

What type does the **array** belong to?

```
type(a)
```

```
numpy.ndarray
```

What is the type to which the **members of the array** belong?

```
type(a[0])
```

```
numpy.float64
```

> The items of a numpy array must belong to the same type.

An alternative way to create an array and populate it with zeros is using the function: `np.empty()`.

```
e = np.empty(3)
```

An array of ones can be produced using the function: `np.ones()`.

```
o = np.ones(5)
print(o)
```

```
[1. 1. 1. 1. 1.]
```

An array can be populated within a range using: `np.linspace(start, stop, num_items)`.

```
# For example, an array in the range 0 to 10
# containing five items with equal spaces between them:

l = np.linspace(0,10,5)
print(l)
```

```
[ 0.   2.5  5.   7.5 10. ]
```

And of course, you can create a numpy array from a normal array:

```
n = np.array([10,20,30])
type(n)
```

```
numpy.ndarray
```

Why turn a regular Python array into a Numpy array?

> Because the calculation speed of Numpy arrays is higher and working with the arrays is much more convenient thanks to the functions that save us the need to work inside loops.

2. How to write a multidimensional array?

```
# This array contains 2 rows and 3 columns

m = np.array([[1, 2, 3] , [4, 5, 6]])
print(m)
```

```
[[1 2 3]
 [4 5 6]]
```

1. A real table with rows and columns, hence the obligation to make sure the number of items is the same in all rows.
2. It is mandatory to observe a uniform data type in all items (all int in our example).

```
# 1 Row
b = np.array([[1, 2, 3]])

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

# 3 Row
d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# And so on ..
```

3. How to review the lineups?

Displays a tuple with the number of rows and columns: `(Nrows, Ncols)`

```
d.shape
```

```
(3, 3)
```

Number of array items.

```
d.size
```

```
9
```

4. How to access items within the array?

```
v = np.array([1, 2, 3, 4, 5])
```

We can retrieve an item from the array according to its index.

```
# For example, you can retrieve the first item:

print(v[0])
```

```
1
```

```
# To retrieve the last item:

print(v[-1])
```

```
5
```

```
# To retrieve the last item:

print(v[1:3])
```

```
[2 3]
```

pay attention!

> The retrieval includes the first item but not the last.

```
# The entire range can be retrieved:

print(v[:])
```

```
[1 2 3 4 5]
```

5. What are scalars, vectors, matrices and tensors?

A **scalar** is a single numerical value:

```
s = 11
```

A **vector** is an object that can be written using a one-dimensional array in one line:

```
v = np.array([1, 2, 3])
```

A **matrix** can be described using a two-dimensional array. A matrix consisting of two vectors:

```
# Matrix contain two vectors
m2 = np.array([[2, 3, 4], [5, 6, 7]])

# Matrix contain three vectors
m3 = np.array([[2, 3, 4], [5, 6, 7], [-1, 4, 6]])
```

Tensors are high-order multidimensional arrays:

```
# We will creat a matrix m1 with the same size of m2:

m1 = np.array([[4, -2, 5],  [2, 3, -1]])
```

We will create a **tensor** from the **two matrices** as follows:

```
t = np.array([m2, m1])
```

> On a **technical level**, a tensor is a collection of matrices. In **machine learning** a tensor is a multidimensional array.

Let's summarize:

Dimension	Name
0	scalar
1	vector
2	matrices
n	tensor

6. How to access items within the multidimensional array?

```
# For example:
print("Array:\t", b)

print("Second item in the first row:\t", b[0, 1])

# To change the value of an item, you can place the new value into it:
b[0, 1] = 7

print("Array after the change:\t", b)
```

```
Array:   [[1 2 3]]
Second item in the first row:    2
Array after the change:  [[1 7 3]]
```

How to access all items of a particular **row** or **column**?

```
print("Array:\n", c)

# Access all items in the first row?
print("Access all items in the first row:\t\t", c[0, :])

# How to access all the items in the second column?
print("Access all the items in the second column:\t", c[: , 1])

# To access all items in the last column:
print("Access all items in the last column:\t\t", c[: , -1])
```

```
Array:
 [[0 1 2]
 [3 4 5]]
Access all items in the first row:              [0 1 2]
Access all the items in the second column:      [1 4]
Access all items in the last column:            [2 5]
```

You can be more **specific**:

```
# To access all items in the first two rows and the first two columns:

c[0:2 , 0:2]
```

```
array([[0, 1],
       [3, 4]])
```

7. Arrays and data types

The most useful data type is a decimal `float`.

```
# We will create an array of consecutive decimal numbers from 0 to 4:

x = np.array([0, 1, 2, 3], dtype=np.float16)
print(x)
```

```
[0. 1. 2. 3.]
```

```
# What is the data type?

print(x.dtype)
```

```
float16
```

To convert type `float` to type `int`:

```
y = x.astype(int)

print(y)          # [0 1 2 3]

print(y.dtype)    # int64
```

```
[0 1 2 3]
int64
```

8. Arrays and conditions

When working with Numpy arrays it is very easy to do **logic**.

```
# For example, which items in the following array are greater than 3?

z = np.array([8, -2, 13, 4, 1.2])
```

```
# The result is an array of booleans:

z > 3
```

```
array([ True, False,  True,  True, False])
```

```
# Pull out the items that meet the condition:

z[z > 3]
```

```
array([ 8., 13.,  4.])
```

The condition can be perfected by using **operators**:

- **| - OR**
- **& - AND**

```
a = z[z>3]            # [ 8. 13.  4.]
print("a:\t", a)

b = z[(z>3)&(z<9)]    # [8. 4.]
print("b:\t", b)

c = z[(z<3)|(z>9)]    # [-2.  13.   1.2]
print("c:\t", c)
```

```
a:      [ 8. 13.  4.]
b:      [8. 4.]
c:      [-2.  13.   1.2]
```

9. How to perform invoicing operations on arrays?

Multiplication of an array:

```
# Multiply the value of each of the items in the array by 2.

2 * a
```

```
array([16., 26.,  8.])
```

Connection of arrays:

```
# Connect each of the items in array a with each
# of the corresponding items in the other array.

a + c
```

```
array([ 6. , 26. ,  5.2])
```

Multiplication between **arrays**:

```
a * c
```

```
array([-16. , 169. ,   4.8])
```

Division between **arrays**:

```
a / c
```

```
array([-4.        ,  1.        ,  3.33333333])
```

Note:

> Since the account operations are performed on parallel items, the structure of the arrays must be the same (rows and columns).

10. Change shape - reshape

We can change the shape of an array using the `reshape` command.

```

```

```
# Convert an array of 1-dim with 6 items into 2-dim with 3 items in each of them:

a = np.array([1, 2, 3, 4, 5, 6])
r = a.reshape(2, 3)
print(r)
```

```
[[1 2 3]
 [4 5 6]]
```

```
# Another example, we can turn a 1-dim array into a 3-dim array:

a = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
r = a.reshape(2, 2, 3)
print(r)
```

```
[[[ 1  2  3]
  [ 4  5  6]]

 [[ 7  8  9]
  [10 11 12]]]
```

Note:

> We can change the shape of the array as much as needed as long as we make sure that the number of items in the original and new array are the same.

```
# If we try to change the array to an incompatible
# number of items we will get an error:

a = np.array([1, 2, 3, 4, 5, 6])
r = a.reshape(2, 3)
print(r)
```

```
[[1 2 3]
 [4 5 6]]
```

```
# There may be a situation where we want the reshape function to calculate one of the dimensions.
# The place of the dimension will be marked with -1.

a = np.array([1, 2, 3, 4, 5, 6])
r = a.reshape(2, 3, -1)
print(r)
```

```
[[[1]
  [2]
  [3]]

 [[4]
  [5]
  [6]]]
```

To flatten an array and turn it into a one-dimensional array, we pass the value -1 to the `reshape` function.

```
a = np.array([[1,2,3],[4,5,6]])
r = a.reshape(-1)
print(r)
```

```
[1 2 3 4 5 6]
```

To add a dimension to a multidimensional array, you can use the `reshape()` method:

```
x = np.arange(1, 7)

print("original:\t\t", x)
print("original shape:\t\t", x.shape)
print("row vector:\t\t", x.reshape((1, -1)))
print("row vector shape:\t", x.reshape((1, -1)).shape)
```

```
original:               [1 2 3 4 5 6]
original shape:         (6,)
row vector:             [[1 2 3 4 5 6]]
row vector shape:       (1, 6)
```

11. Transposition

Vectors are divided into two types:

1. Row vectors
2. Column vectors

You can use transposition to turn a row vector into a column vector, and vice versa.

> In order for Python to perform a transposition, you need to use the following syntax: `vector.T`.

Transpose just like that on a vector **doesn't work**:

```
v = np.array([1, 2, 3])
v.T
```

```
array([1, 2, 3])
```

The reason is that Python requires that we first turn a vector into a **multidimensional array**.

```
v_reshaped = v.reshape(1,3)
v_reshaped
```

```
array([[1, 2, 3]])
```

Only on the multidimensional array we can do the **transposition**:

```
v_reshaped_transposed = v_reshaped.T
v_reshaped_transposed
```

```
array([[1],
       [2],
       [3]])
```

Transposition can also be done on **matrices**:

> The result is that in every item the row and column are reversed.

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

print("\nOriginal:\n", mat)
print("\nAfter transposition to the matrix:\n", mat)
```

```
Original:
 [[1 2 3]
 [4 5 6]]

After transposition to the matrix:
 [[1 2 3]
 [4 5 6]]
```

12. How to multiply matrices?

Matrix multiplication is the most common operation in machine learning.

Under certain conditions it is possible to multiply between arrays of **different dimensions** using the `dot` function of numpy.

```
# For example, multiply vector by matrix:

vector = np.array([1, 2, 3])
matrix = np.array([[2, 3, 4], [5, 6, 7], [1, 5, 8]])

np.dot(vector, matrix)
```

```
array([15, 30, 42])
```

Condition:

> Matrix multiplication is possible when the number of **columns in the right array** is equal to the number of **rows in the left array**.

Note:

> Given that the role of the transpose operation, we can sometimes use transpose as a precondition for performing multiplications. Another way is to use python's `reshape` operatio.

Error example:

```
m1 = np.array([[-5, 8], [2,1], [-7, 3]])
m2 = np.array([[2, 3], [6, -4], [7, 42]])
```

```
# Columns number in the right matrix is not equal to the rows number in the left matrix.

print("m1 shape:\t", m1.shape)
print("m2 shape:\t", m2.shape)
```

```
m1 shape:        (3, 2)
m2 shape:        (3, 2)
```

```
# We will get an error.

np.dot(m1,m2)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-50-fc79073f7528> in <module>
      1 # We will get an error.
      2
----> 3 np.dot(m1,m2)

<__array_function__ internals> in dot(*args, **kwargs)

ValueError: shapes (3,2) and (3,2) not aligned: 2 (dim 1) != 3 (dim 0)
```

The solution in this case:

```
# Change the shape of the m1 matrix so that it contains 2 row vectors each with 3 values.

m1_reshape = m1.reshape(2,3)
```

```
# Columns number in the right matrix equal to the rows number in the left matrix.

print("m1_reshape shape:\t", m1_reshape.shape)
print("m2 shape:\t\t", m2.shape)
```

```
m1_reshape shape:        (2, 3)
m2 shape:                (3, 2)
```

```
# Now perform the multiplication:

np.dot(m1_reshape, m2)
```

```
array([[ 52,  37],
       [-19, 157]])
```

13. Invoicing operations on each of the array items

> Numpy allows us to perform the arithmetic operations in a short time.

```
# We will do some calculation operations on the following array:

x = np.arange(5)
print(x)
```

```
[0 1 2 3 4]
```

```
# We can add, reduce, multiply, divide a scalar to each item:

print("x + 3 = ", x+3)
print("x - 3 = ", x-3)
print("x * 3 = ", x*3)
print("x / 3 = ", x/3)
```

```
x + 3 =  [3 4 5 6 7]
x - 3 =  [-3 -2 -1  0  1]
x * 3 =  [ 0  3  6  9 12]
x / 3 =  [0.         0.33333333 0.66666667 1.         1.33333333]
```

```
# You can also divide and round down:
print("x // 3 = ", x//3)

# It is possible to make a % modulus
print("x % 3 = ", x%3)

# Raise to the ** power
print("x ** 3 = ", x**3)

# and reverse the sign of all array items:
print("x * (-1) = ", -x)
```

```
x // 3 =  [0 0 0 1 1]
x % 3 =  [0 1 2 0 1]
x ** 3 =  [ 0  1  8 27 64]
x * (-1) =  [ 0 -1 -2 -3 -4]
```

```
# And you can also combine the functions:

print("-4 * [[(3*x) + 2] ** 3] = ", -4*(3*x+2)**3)
```

```
-4 * [[(3*x) + 2] ** 3] =  [   -32   -500  -2048  -5324 -10976]
```

> The simple syntax we've seen for performing math operations on Numpy arrays isn't the only way. There is a more explicit syntax that calls **functions by name**.

```
# For example, to add to an array:

np.add(x, 3)
```

```
array([3, 4, 5, 6, 7])
```

```
# To reduce:

np.subtract(x, 3)
```

```
array([-3, -2, -1,  0,  1])
```

```
# And to get an absolute value:

np.absolute(x)
```

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

```
# Sum:

np.sum(x)
```

```
10
```

```
# Minimum and maximum:

print("Min:\t", np.min(x))
print("Max:\t", np.max(x))
```

```
Min:    0
Max:    4
```

```
# Numpy function to find the index number of the minimum and maximum value:

print("Min index:\t", np.argmin(x))
print("Max index:\t", np.argmax(x))
```

```
Min index:      0
Max index:      4
```

It is possible to run **trigonometric** functions:

```
# For example, a sine function:

np.sin(x)
```

```
array([ 0.        ,  0.84147098,  0.90929743,  0.14112001, -0.7568025 ])
```

```
# Or run statistics functions:

np.mean(x)
```

```
2.0
```

```
# Standard deviation:

np.std(x)
```

```
1.4142135623730951
```

```
# Statistical variation:

np.var(x)
```

```
2.0
```

14. How to cut a Numpy array?

Intersection of a Numpy array is easiest to demonstrate with an image because an image is an array in 3-dim:

1. length
2. height
3. color

```
# Predict the image:

from skimage import io
photo = io.imread('https://images.indianexpress.com/2021/05/friends-the-reunion-1200-2.jpg')
type(photo)
```

```
numpy.ndarray
```

```
print("The shape of the photo:", photo.shape)
```

```
The shape of the photo: (667, 1200, 3)
```

```
# We will present the image:

import matplotlib.pyplot as plt
plt.imshow(photo)
```

```
<matplotlib.image.AxesImage at 0x7f7f6c9af910>
```


```
# Let's reverse the lines:

plt.imshow(photo[::-1])
```

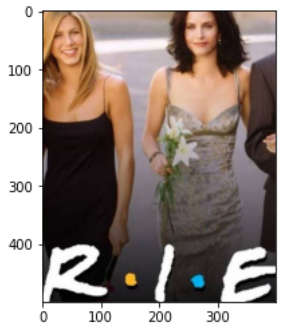
```
<matplotlib.image.AxesImage at 0x7f7f6ac5e1d0>
```

```
# We will reverse the columns:

plt.imshow(photo[:,::-1])
```

```
<matplotlib.image.AxesImage at 0x7f7f6abd46d0>
```

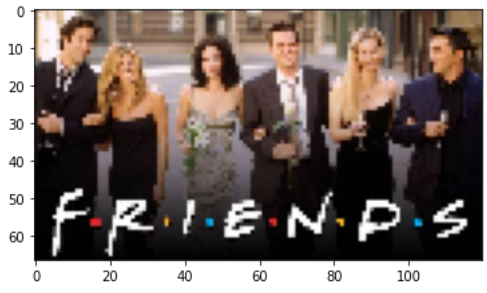
```
# We will crop and show only part of the image:

plt.imshow(photo[100:600,200:600])
```

```
<matplotlib.image.AxesImage at 0x7f7f6ab46490>
```



```
# We will represent every 10'th row and every 10'th column:

plt.imshow(photo[::10,::10])
```

```
<matplotlib.image.AxesImage at 0x7f7f6aa94890>
```



```
# use where and masked np array

masked_photo = np.where(photo > 100, 255, 0)
plt.imshow(masked_photo)
```

```
<matplotlib.image.AxesImage at 0x7f7f6aa02750>
```