NumPy

- **Introduces ndarrays.** Those are n-dimensional arrays of homogeneous data types, with many operations being performed on compiled code for perfomance.

Working with NumPy

The first thing we need to do when installing a Python library we can type `pip install numpy` in the terminal. If we are using JupyterNotebook we can run the following, which will basically call our terminal.

In [1]:
```
!pip install numpy
```

```
Requirement already satisfied: numpy in c:\users\sergi\anaconda3\lib\site-packages
(1.21.5)
```

Once NumPy is installed we might want to **import** it. To import a function we just need to type:

In [2]:
```
import numpy as np
```

Create an array from a list

In [4]:
```
# Create a 1-dimensional array

A = np.array([1,3,4,5,6])     # A is our first array

print(A)   # allows to visualize A
type(A)    # check that the type is ndarray
```

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

Out[4]:
```
numpy.ndarray
```

In [6]:
```
# Create a 2-dimensional array

B = np.array([[1,2],[3,4],[5,6]])    # 2-d array
print(B)
```

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

- **shape** : number of dimensions of the array.
- **size** :number of elements in the array
- **ndim** : number of dimensions in the arrya

In [11]:
```
print(f'The dimensions of array A are{A.shape}')
print(f'The dimensions of array B are{B.shape}')
```

```
The dimenions of array A are(5,)
The dimenions of array B are(3, 2)
```

In [17]:
```python
# We can also check the number of dimnesions:

print(A.ndim)
print(B.ndim)

# And we can also check the type
print(A.dtype)


# We can also check the total number of elements of the array

print(A.size)
print(B.size)
```

```
1
2
int32
5
6
```

Creating Arrays

np.array()

Can take as inputs Python lists among others.

In [24]:
```python
list1 = [1,2,4]
list2 = [9,29129,12]
list3 = [list1,list2]

array1 = np.array(list3)
print(array1)
print(array1.dtype)

# if we wish we could specify the data type:

array2 = np.array(list3,dtype= np.int64)
print(array2.dtype)
```

```
[[    1     2     4]
 [    9 29129    12]]
int32
int64
```

np.zeros()

It allows to create an array full of 0s.

In [26]:
```python
zeros = np.zeros(10)     # create a 10x1 array full of 0s.
print(zeros)
```

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

In [27]:
```python
zeros = np.zeros((10,3)) # create a 10x3 array full of 0s.
print(zeros)
```

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

np.ones()

It allows to create an array full of 1s.

In [28]:
```
ones = np.ones(12)
print(ones)
```

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

np.empty()

It allows to create an array with random numbers. The reason to use empty over zeros is speed.

In [29]:
```
empty = np.empty((2,2))
print(empty)
```

```
[[2.12199579e-314 4.67296746e-307]
 [5.98807563e-321 3.79442416e-321]]
```

np.arange()

It allows to create arrays in a range of elements, considering the initial valyue, the final value, and the step size.

In [33]:
```
C = np.arange(1,10,2)
print(C)

# A simpler version is
D = np.arange(4)
print(D)
```

```
[1 3 5 7 9]
[0 1 2 3]
```

np.linspace()

It allows to create arrays with values that are spaced linearly in a specified interval. The difference with `np.arange()` is that here instead of step size we choose the number of steps.

In [34]:
```
E = np.linspace(5,25,6)
print(E)
```

```
[ 5.  9. 13. 17. 21. 25.]
```

np.ones_like()

It allows to create an arrays of ones with the same shape as an specified array.

In [35]:
```
array1 = np.array([[1,3],[3,4],[2,3]])

array2 = np.ones_like(array1)

print(array2)
```

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

The same can be done with `np.zeros_like()` and `np.empty_like()` .

np.identity()

It allows to create the identity matrix.

In [38]:
```
identity = np.identity(3)
print(identity)
```

```
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

Basic array operations

Operations are **element-wise** :

In [45]:
```
a = np.array([20, 30, 40, 50])
b = np.arange(4)

#Sum:

print('The sum is :',a+b)
print('The sum is :',np.add(a,b))


# Subtraction

print('The subctraction is :',a-b)
print('The subctraction is :',np.subtract(a,b))
```

```
The sum is : [20 31 42 53]
The sum is : [20 31 42 53]
The subctraction is : [20 29 38 47]
The subctraction is : [20 29 38 47]
```

- **The product operator** `*` is element-wise.
- **Matrix product** with `@`.

In [2]:
```
# Multiplication:
A = np.array([[1, 1],
              [0, 1]])
B = np.array([[2, 0],
              [3, 4]])
```

```
# Element-wise:

print('element wise', A*B)
print(np.multiply(A,B))

# Matrix Produt

print('matrix product', A@B)
print(A.dot(B))
```

```
element wise [[2 0]
 [0 4]]
[[2 0]
 [0 4]]
matrix product [[5 4]
 [3 4]]
[[5 4]
 [3 4]]
```

Dvision

In [49]:
```
# Dvision

# element-wise:
print(A/B)
print(np.divide(A,B))
```

```
[[0.5   inf]
 [0.   0.25]]
[[0.5   inf]
 [0.   0.25]]
C:\Users\Sergi\AppData\Local\Temp\ipykernel_23828\790273862.py:4: RuntimeWarning:
divide by zero encountered in true_divide
  print(A/B)
C:\Users\Sergi\AppData\Local\Temp\ipykernel_23828\790273862.py:5: RuntimeWarning:
divide by zero encountered in true_divide
  print(np.divide(A,B))
```

We can also perform operations between an array and a scalar.

In [51]:
```
print(A**2)    # element-wise x^2
print(A*2)     # element-wise x*2
print(A+2)     # element-wise x+2
```

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

Indexing, Slicing and Iterating

Arrays can be indexed, sliced and iterated over much like lists and any other Python sequences. Let's see how it works with one-dimensional arrays:

In [55]:
```
a = np.arange(10)**3
print(a)

print(a[2])    # access the element with index 2, the third element.
```

```
print(a[2:6])  # slice the array
print(a[-4:])  # slice from the -4 element to the end
```

```
[  0   1   8  27  64 125 216 343 512 729]
8
[  8  27  64 125]
[216 343 512 729]
```

We can also choose the step size in the interal slice. For example suppose we would like to obtain a subset containing only every three elements:

In [57]:
```
print(a[1:8:3])
```

```
[  1  64 343]
```

In [58]:
```
# If we want to reverse the array:
print(a[::-1])
```

```
[729 512 343 216 125  64  27   8   1   0]
```

Multidimensional arrays have one index per axis. For example for a 2-dimensional array we would have `array[i][j]` where `i` are the row dimensions and `j` are the column dimensions. We can also write it as `array[i,j]`.

In [63]:
```
a = np.array([[ 0,  1,  2,  3],
       [10, 11, 12, 13],
       [20, 21, 22, 23],
       [30, 31, 32, 33],
       [40, 41, 42, 43]])

# Access one element:

print(a[0,0],a[0][0])  # both will return the same

# Slices

print(a[:,2])       # only the third column
print(a[2:5,3:])

# We can also just inlcude one index

print(a[2])  # third row
```

```
0 0
[ 2 12 22 32 42]
[[23]
 [33]
 [43]]
[20 21 22 23]
```

In [68]:
```
# for a 3-dimensional array we have:

c = np.array([[[  0,  1,  2],
               [ 10, 12, 13]],
              [[100, 101, 102],
               [110, 112, 113]]])

# we can access all its dimensions:

print(c[1,1,1])
print(c[1,...])
```

```
112
[[100 101 102]
 [110 112 113]]
```

The dots (...) represent as many colons as needed to produce a complete indexing tuple. For example, if `x` is an array with 5 axes, then `x[1, 2, ...]` is equivalent to `x[1, 2, :, :, :]`, `x[..., 3]` to `x[:, :, :, :, 3]` and `x[4, ..., 5, :]` to `x[4, :, :, 5, :]`.

Slice based on logical conditions

We can also slice arrays based on logical conditions like `<` or `>`.

In [4]:
```
a = np.array([[ 0,  1,  2,  3],
       [10, 11, 12, 13],
       [20, 21, 22, 23],
       [30, 31, 32, 33],
       [40, 41, 42, 43]])


print(a[a>10])   # only elements greater than 10
print(a[a<5])    # only elements smaller than 5
print(a[(a > 2) & (a < 11)])  # elements between 2 and 11
```

```
[11 12 13 20 21 22 23 30 31 32 33 40 41 42 43]
[0 1 2 3]
[ 3 10]
```

Shape Manipulation and other methods

There are many ways of doing shape manipulation with arrays. We will now review some of them:

np.ravel()

It returns the specified array flattened (with only one dimension).

In [5]:
```
print(a)
print(a.ravel())   # the array is now flattened
```

```
[[ 0  1  2  3]
 [10 11 12 13]
 [20 21 22 23]
 [30 31 32 33]
 [40 41 42 43]]
[ 0  1  2  3 10 11 12 13 20 21 22 23 30 31 32 33 40 41 42 43]
```

np.reshape()

The `reshape` method returns its argument with a modified shape.

In [84]:
```
print(a.size) # the amount of elements needs to be the same
print(a)
print('New a ',a.reshape(2,5,2))
```

```
20
[[ 0  1  2  3]
 [10 11 12 13]
 [20 21 22 23]
 [30 31 32 33]
 [40 41 42 43]]
New a  [[[ 0  1]
  [ 2  3]
  [10 11]
  [12 13]
  [20 21]]

 [[22 23]
  [30 31]
  [32 33]
  [40 41]
  [42 43]]]
```

np.resize()

It does the same as `np.reshape` but instead of returning a new array, it modifies the array itself

In [85]:
```
a.resize(5,2,2)
print(a)   # a is now modified.
```

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

 [[10 11]
  [12 13]]

 [[20 21]
  [22 23]]

 [[30 31]
  [32 33]]

 [[40 41]
  [42 43]]]
```

np.swapaxes()

It allows to change two axis of an array.

In [86]:
```
print(a)
b = np.swapaxes(a,0,1) # we are changing axis 0 for 1
print(b)
```

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

 [[10 11]
  [12 13]]

 [[20 21]
  [22 23]]

 [[30 31]
  [32 33]]

 [[40 41]
  [42 43]]]
[[[ 0  1]
  [10 11]
  [20 21]
  [30 31]
  [40 41]]

 [[ 2  3]
  [12 13]
  [22 23]
  [32 33]
  [42 43]]]
```

np.transpose()

It returns an array with axes transposed.

In [90]:

```
# with a 2-d array
a = np.linspace(0,250,20).reshape(2,10)
print(a)

print(np.transpose(a))
```

```
[[  0.          13.15789474  26.31578947  39.47368421  52.63157895
   65.78947368  78.94736842  92.10526316 105.26315789 118.42105263]
 [131.57894737 144.73684211 157.89473684 171.05263158 184.21052632
  197.36842105 210.52631579 223.68421053 236.84210526 250.        ]]
[[  0.         131.57894737]
 [ 13.15789474 144.73684211]
 [ 26.31578947 157.89473684]
 [ 39.47368421 171.05263158]
 [ 52.63157895 184.21052632]
 [ 65.78947368 197.36842105]
 [ 78.94736842 210.52631579]
 [ 92.10526316 223.68421053]
 [105.26315789 236.84210526]
 [118.42105263 250.        ]]
```

In [91]:

```
# example with more than 2 dimensions, without specifying axis:
a = np.ones((2, 3, 4, 5))
print(np.transpose(a).shape)
```

```
(5, 4, 3, 2)
```

In [93]:

```
# Example 2-d, specifying axis
a = np.ones((1, 2, 3))
print(np.transpose(a, (1, 0, 2)).shape) #we are indicating that we want the first
```

```
(2, 1, 3)
```

Stacking together different arrays

There are different ways. To do it along different axis we can use `np.vstack()` or `np.hstack()` when dealing with 2-dimensional arrays.

In [100…

```
a = np.empty([2,2])
b = np.zeros([2,2])

print(np.vstack((a,b)))
print(np.hstack((a,b)))
```

```
[[2.12199579e-314 4.67296746e-307]
 [5.98807563e-321 3.79442416e-321]
 [0.00000000e+000 0.00000000e+000]
 [0.00000000e+000 0.00000000e+000]]
[[2.12199579e-314 4.67296746e-307 0.00000000e+000 0.00000000e+000]
 [5.98807563e-321 3.79442416e-321 0.00000000e+000 0.00000000e+000]]
```

We can also stack one dimensional arrays as columns in two dimensional arrays using `np.column_stack()` or the row equivalent `np.row_stack()`.

In [101…

```
a = np.empty([2,2])
b = np.zeros([2,1])
print(np.column_stack((a, b)))
```

```
[[2.12199579e-314 4.67296746e-307 0.00000000e+000]
 [5.98807563e-321 3.79442416e-321 0.00000000e+000]]
```

Splitting Arrays

Finally, we can also split one array into multiples arrays using `np.hsplit()` or `np.vsplit()`.

In [104…

```
a = np.zeros([4,3])
print(np.hsplit(a, 3))   # split a into three arrays


b = np.zeros([5,7])
print(np.vsplit(b,5))  #  split into five arrays
```

```
[array([[0.],
       [0.],
       [0.],
       [0.]]), array([[0.],
       [0.],
       [0.],
       [0.]]), array([[0.],
       [0.],
       [0.],
       [0.]])]
[array([[0., 0., 0., 0., 0., 0., 0.]]), array([[0., 0., 0., 0., 0., 0., 0.]]), arr
ay([[0., 0., 0., 0., 0., 0., 0.]]), array([[0., 0., 0., 0., 0., 0., 0.]]), array
([[0., 0., 0., 0., 0., 0., 0.]])]
```

NumPy Functions

Statistics

np.mean()

It allows to compute the arithmetic mean along the specified axis.

In [111…

```
a = np.column_stack((np.zeros([4,2]),np.ones([4,1])))

print(a)

print(np.mean(a,axis=0))   # Prints the mean for each column

print(np.mean(a,axis=1))  # prints the mean of each row

print(np.mean(a))         # prints the mean of all the values of the array
```

```
[[0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]]
[0. 0. 1.]
[0.33333333 0.33333333 0.33333333 0.33333333]
0.3333333333333333
```

np.std()

Like `np.mean()` it allows to compute the standard deviation along the specified axis.

In [112…

```
print(np.std(a,axis=0))   # there is no deviation
print(np.std(a))
```

```
[0. 0. 0.]
0.4714045207910317
```

np.var()

It allows to copmute the variance along the specified axis.

In [113…

```
print(np.var(a,axis=0))

print(np.var(a))
```

```
[0. 0. 0.]
0.22222222222222224
```

np.average()

It allows to compute the weighted average along the specified axis.

In [114…

```
data = np.arange(6).reshape((3, 2))

np.average(data, axis=1, weights=[1/4, 3/4])  # average across columns
```

Out[114]:

```
array([0.75, 2.75, 4.75])
```

TIP! Sometimes we might have missing observations. With NumPy we can code those as Not a Number using the **np.nan** constant

np.nanmean()

It allows to compute the arithmetic mean along the specified axis, ignoring Nans.

In [119...
```
a = np.array([[1, np.nan], [3, 4]])
print(a)

print(np.mean(a))      # this will be incorrect
print(np.nanmean(a))   # right way to deal with nans
```

```
[[ 1. nan]
 [ 3.  4.]]
nan
2.6666666666666665
```

Similar to `np.nanmean()` we could also use `np.nanstd()` and `np.nanvar()` for the standard deviation and variance counterparts.

Arithmetic Operations

np.sum()

It allows to sum array elements over a given axis.

In [122...
```
data = np.arange(6).reshape((3, 2))
print(np.sum(data,axis=0))   # The function way
print(data.sum(axis=0))      # The method way
```

```
[6 9]
[6 9]
```

np.cumsum()

Returns the cumulative sum of elements along a given axis.

In [124...
```
print(data)
print(np.cumsum(data,axis=0))
```

```
[[0 1]
 [2 3]
 [4 5]]
[[0 1]
 [2 4]
 [6 9]]
```

np.min()

It returns the minimum element along a given axis.

In [125...
```
print(np.min(data,axis=0))
print(np.min(data))
```

```
[0 1]
0
```

The maximum counterpart is `np.max()`.

np.sqrt()

It returns the non-negative square-root of an array, element-wise.

In [126…

```
print(data)
print(np.sqrt(data))
```

```
[[0 1]
 [2 3]
 [4 5]]
[[0.         1.        ]
 [1.41421356 1.73205081]
 [2.         2.23606798]]
```

np.exp()

It returns the exponential of all the elements in the input array

In [127…

```
print(np.exp(data))
```

```
[[  1.           2.71828183]
 [  7.3890561   20.08553692]
 [ 54.59815003 148.4131591 ]]
```

Copmarisons

np.any()

Test wether any array element along a given axis evaluates as True.

In [128…

```
np.any([[True, False], [True, True]])
```

Out[128]:

```
True
```

On its own it is not very informative, unless we combine it with logical operations, to gain information on our array.

In [129…

```
print(np.any(data>1))   # we have some values greater than 1
```

```
True
```

In [130…

```
print(np.any(data>1,axis=0))  # we can also specify the axis
```

```
[ True  True]
```

np.all()

It is similar to `np.any()` but now evaluates if all the elements of the array satisfy the condition.

In [131…

```
a = np.zeros([3,2])
print(np.all(a==0))
```

```
True
```

In [132…

```
# suppose we have a one:

a[0,0] = 1
print(np.all(a==0,axis=0))
```

```
[False  True]
```

Other usefull functions

There are some functions that do not belog to any of the prevoius categories, but still they are useful and worth studying.

np.sort()

Returns a sorted copy of any array

In [136…

```
a = np.array([1,435,3,45,65,4,7,65,65756,878,2])
print(a)
print(np.sort(a))
```

```
[    1   435     3    45    65     4     7    65 65756   878     2]
[    1     2     3     4     7    45    65    65   435   878 65756]
```

np.round()

Rounds an array to the given number of decimals.

In [141…

```
a = np.array([3.4545,4.3435,6.746757,8.67324])
print(a)

print(np.round(a,decimals=2))
```

```
[3.4545   4.3435   6.746757 8.67324 ]
[3.45 4.34 6.75 8.67]
```

np.where()

Allows to locate elements that satisfy a conditon.

In [ ]:

```
a = np.arange(10).reshape([2,5])
print(a)

np.where(a>5)
```

What does the output mean? It is returning the indexs of the elements that satisfy the condition. The first index corresponds to the row, and the second to the column.

NumPy random sampling

NumPy has a module that produces random numbers. The main functions are:

np.random.randint()

It returns random numbers within the specified values.

In [147…
```
a = np.random.randint(5,25,(2,3))   # produces a 2x3 array with numbers in the inte
print(a)
```

```
[[ 5 23  8]
 [ 9  7 21]]
```

np.random.rand()

Returns random numbers for the specified array dimensions.

In [149…
```
a = np.random.rand(2,2)  # creates a random 2x2 array
print(a)
```

```
[[0.09201743 0.79882103]
 [0.6226247  0.94176854]]
```

np.random.random_sample()

Returns random floats on the half-open interval `[0,1)`.

In [150…
```
a = np.random.random_sample((2,3))

print(a)
```

```
[[0.80948258 0.42618003 0.58292804]
 [0.49953237 0.65621386 0.3936935 ]]
```

np.random.seed()

It allow us to set the random number generator seed to achieve replicability of our results.

In [153…
```
np.random.seed(10)  # set seed to 10

a = np.random.randint(0,20,(2,3))
print(a)
```

```
[[ 9  4 15]
 [ 0 17 16]]
```

np.random.uniform()

It allows to draw from a uniform distribution in the prespecified interval.

In [156…
```
a = np.random.uniform(3,7,(2,2))  # a 2x2 array in the interval [3,7] drawn from a
print(a)
```

```
[[3.10068691 5.83683204]
 [4.06226451 4.05441138]]
```

np.random.binomial()

Allows to draw from a binomial distribution. We can choose the amount of drawns and the probability of a 1.

In [158…
```
a = np.random.binomial(5,0.3,size=20)  # a 20x1 array of 5 repeated drawns with su
print(a)
```

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

np.random.normal()

Allows to draw from a normal distribution with a given mean and standard deviation.

In [159…
```
mean = 5
std = 2
size = (4,5)
a = np.random.normal(mean,std,size)
print(a)
```

```
[[4.24490727 6.39126232 7.0496893  8.41551089 1.06576555]
 [3.25808667 2.78634904 5.55024732 4.27108771 2.66969306]
 [2.31386023 7.90678572 5.0023389  4.97615923 6.43176629]
 [2.62846291 3.72734419 3.77837231 2.36121721 3.87654317]]
```

Save and Load Arrays

We have already seen most of the functions that allow us to work with arrays. But how can we save an array to our computer?

np.savetxt()

It allows to save an array in '.txt' format. It has an important limitation, arrays can only have 1 or 2 dimensions.

In [161…
```
# save the previous array
np.savetxt('array.txt',a)  # we are saving with the name array.txt
```

np.loadtxt()

To load an array already saved, we can just use `np.loadtxt()` .

In [162…
```
newarray = np.loadtxt('array.txt')
print(newarray)
```

```
[[4.24490727 6.39126232 7.0496893  8.41551089 1.06576555]
 [3.25808667 2.78634904 5.55024732 4.27108771 2.66969306]
 [2.31386023 7.90678572 5.0023389  4.97615923 6.43176629]
 [2.62846291 3.72734419 3.77837231 2.36121721 3.87654317]]
```

Histograms

NumPy also allow us to prepare our data to future visualizations. We can for example compute histograms with `np.histogram()` which allows to compute the histogram of a dataset.

In [163…

```
a = np.histogram([1, 2, 1], bins=[0, 1, 2, 3])  # we input an array and the bins
print(a)
```

```
(array([0, 2, 1], dtype=int64), array([0, 1, 2, 3]))
```

The function returns the values of the histogram and the bin edges. We can also obtain the histogram normalized as a density.

In [164…

```
a = np.histogram([1, 2, 1], bins=[0, 1, 2, 3],density=True)
print(a)
```

```
(array([0.        , 0.66666667, 0.33333333]), array([0, 1, 2, 3]))
```

In [ ]:

```

```