→ Python Libraries

1) NumPy

Numpy is a Python library used for working with arrays

It also has functions for working in domain of linear algebra, and matrices.

↳ Import Numpy

Once Numpy is installed, we import it in our application by adding the import keyword (Numpy is usually imported under the np alias)

```
import numpy as np
```

↳ Installation:

```
pip install numpy
```

↳ NumPy Creating Arrays

Numpy is used to work with arrays. The array object in Numpy is called ndarray
We can create a NumPy ndarray object using the array() function

Example:

```
import numpy as np
array = np.array([1, 2, 3, 4, 5])
print(arr) # Output: [1 2 3 4 5]
print(type(arr)) # Output: <class 'numpy.ndarray'>
```

!! To create an ndarray we can pass a list, tuple or any array-like object into the array() method, and it will be converted into an ndarray

↳ Array Dimension

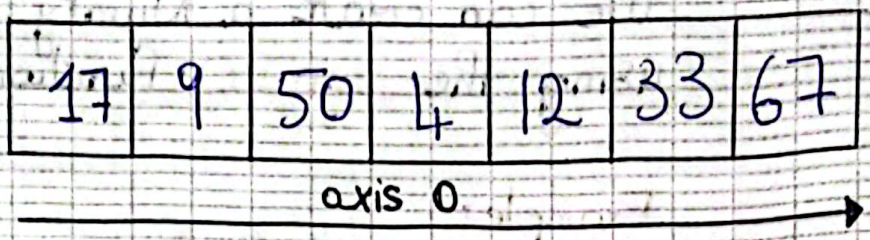
A dimension in arrays is one level of array depth (nested arrays)

. 0-D Arrays

0-D arrays, or Scalars, are the elements in an array. Each value in an array is a 0-D array

1-D Arrays:

An array that has 0-D arrays as its elements is called uni-dimentional or 1-D array



Example:

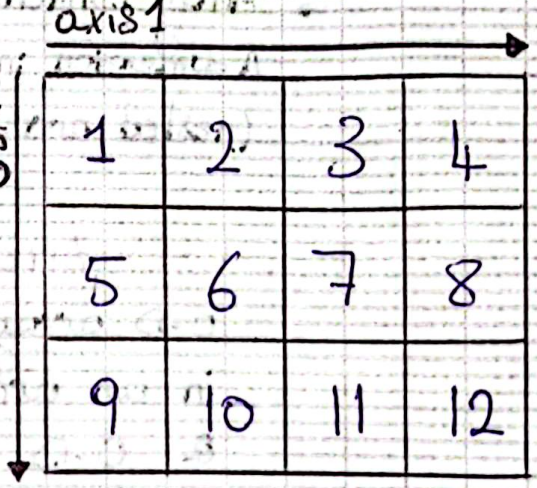
```
import numpy as np
array = np.array([17,9,50,4,12,33,67])
print(arr)#output:[17 9 50 4 12 33 67]
```

2-D Arrays

An array that has 1-D arrays as its elements is called a 2-D array

Axis-0 is the direction that runs downward down the rows.

Axis-1 is the direction that runs horizontally across the columns



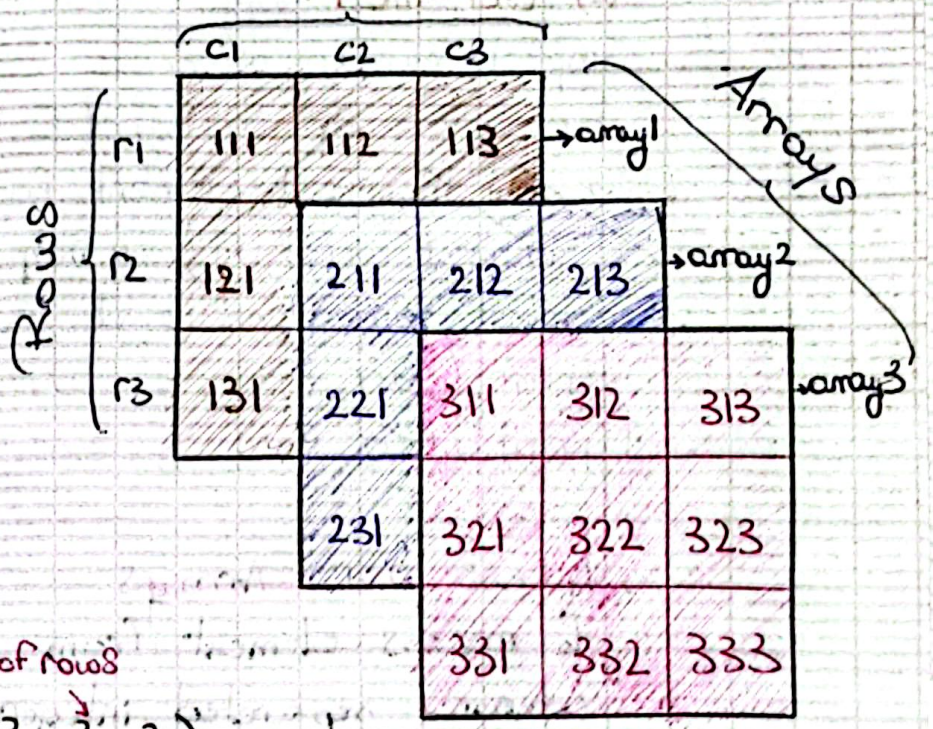
its elemen

Example:

```
import numpy as np
array = np.array([[1,2,3],[4,5,6]])
print(arr) #Output: [ [1 2 3]
                     [4 5 6] ]
```

- 3-D arrays

An array that has 2-D arrays (matrices) as its elements is called 3-D array



Shape: (3, 3, 3)

nb of arrays — nb of rows — nb of columns

Example:

```
import numpy as np
arr = np.array([[[1,2,3],[4,5,6]],[[1,2,3],[4,5,6]]])

print(arr) #Output [ [[1 2 3]
                      [4 5 6]]

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

!! arr.ndim : attribute that return an int that tells us how many dimensions the array have

↳ Numpy Array Indexing

we can access an array element by referring to its index number (start with 0)

Example:

```
import numpy as np
array = np.array([1, 2, 3, 4])
# Get the first element from the array
  print(arr[0]) # output: 1
# Get the third and fourth element and add them
  print(arr[2] + arr[3]) # output: 7
!! print(arr[2], arr[3]) # output: 3 4
```

↳ Access 2-D Arrays

To access element from 2-D array we can use comma separated integers representing the row and the column in wich the element is placed

Example:

```
import numpy as np
arr = np.array([1,2,3,4,5],[6,7,8,9,10])
print('2nd element on 1st row:', arr[0,1]
                                    row  column
print('5th element on 2nd row:', arr[1,4]
```

→ Access 3D Arrays

To access element from 3-D array we can use comma separated integers representing the array, the row, and the column in which the element is placed

Example:

```
import numpy as np
arr = np.array([[[1,2,3],[4,5,6]],
                [[7,8,9],[10,11,12]]])

print(arr[0,1,2]) # output: 6
```

(0 → array, 1 → row, 2 → column)

→ Numpy Array Slicing

Slicing in python means taking elements from one given index to another given index

↳ we pass slice instead of index like this

[start : end] (end not included)

Example:

```
import numpy as np
arr = np.array([1,2,3,4,5,6,7])
print(arr[1:5]) # Output: [2 3 4 5]
```

↳ If we don't pass start its considered 0

```
print(arr[:4]) # Output: [1 2 3 4]
```

↳ If we don't pass end its go to the end of arr

```
print(arr[4:]) # Output: [5 6 7]
```

↳ we can also define the step : [start : end : step]
(by default : 1)

```
print (arr [1:5:2]) # Output: [2 4]
```

↳ Slicing 2-D Arrays

Example:

```
import numpy as np
arr = np.array ([[1,2,3,4,5], [6,7,8,9,10]])
# From the second row, slice elements from
  index 1 to index 4
  print (arr[1, 1:4]) # Output: [7 8 9]
# From both rows return index 2
  print (arr[0:2, 2]) # Output: [3 8]
# From both rows slice index 1 to 4, this will
  return a 2-D array
  print (arr [0:2, 1:4]) # Output: [[2 3 4] [7 8 9]]
```

→ Numpy Data Types

Numpy has some extra data types, and refer to data types with one character:

- i : integer
- b : boolean
- u : unsigned integer
- f : float
- c : complex float
- m : time delta
- M : datetime
- O : Object
- S : String
- U : unicode String
- V : void

Check if Array Owns its Data

We can check if an array owns its data using the base attribute that return None if the array owns the data. Otherwise it refers to the original array

Example:

```
import numpy as np
arr= np.array([1,2,3,4,5])
x = arr.copy()
y = arr.view()
print(x.base) # Output: None
print(y.base) # Output: [1 2 3 4 5]
```

Numpy Array Shape

The shape of an array is the nb of elements in each dimension

Numpy arrays have an attribute called Shape that returns a tuple with each index having the nb of corresponding elements

Example:

Print the shape of 2-D array

```
import numpy as np
arr =np.array([[1,2,3,4],[5,6,7,8]])
print(arr.shape) # Output: (2, 4)
```

(2 → row, 4 → column)

→ Numpy Array Reshaping

We can change the shape of an array using the reshape() function.

Example: 1D → 2D / 1D → 3D

```
import numpy as np
arr = np.array([1,2,3,4,5,6,7,8,9,10,11,12])
arr2D = arr.reshape(4,3)
print(arr2D) # output: [[1  2  3]
                        [4  5  6]
                        [7  8  9]
                        [10 11 12]]
arr3D = arr.reshape(2,3,2)
print(arr3D) # output: [[[1 2]
                        [3 4]
                        [5 6]]
                       [[7 8]
                        [9 10]
                        [11 12]]]
```

→ Flattening the array:

It means converting a multidimensional array into a 1D array. We can use reshape(-1) to do this.

Example: 2D → 1D

```
import numpy as np
arr = np.array([[1,2,3],[4,5,6]])
arr1D = arr.reshape(-1)
print(arr1D) # output: [1 2 3 4 5 6]
```

↳ Checking the Data Type of an Array

We can check the data type of the array using the property called .dtype

Example:

```
import numpy as np
arr = np.array ([1, 2, 3, 4])
print (arr.dtype) # output: int64
```

↳ Creating Arrays with a Defined Data Type

We can creat an array with a specified data type using the optional argument dtype in the array() function

Example:

```
import numpy as np
arr = np.array ([1, 2, 3, 4], dtype = 'S')
print (arr) # output : [b'1' b'2' b'3' b'4']
print (arr.dtype) # output : S1
```

↳ Converting Data Type on Existing Arrays

The best way to change the data type of an existing array, is to make a copy of the array with the astype() method that creates a copy of the array, and allow us to specify the data type as a parameter

Example:

```
import numpy as np
arr = np.array ([1.1, 2.1, 3.1])
newarr = arr.astype ('i')
print (newarr) # output : [1 2 3]
print (newarr.dtype) # output: int32
```

Numpy Array Copy vs View

The main difference between a copy and a view of an array is that, the copy is a new array, and the view is just a view of the original array

COPY

Example:

```
import numpy as np
arr = np.array([1,2,3,4,5])
x = arr.copy()
arr[0] = 42
print(arr) # output: [42 2 3 4 5]
print(x) # output: [1 2 3 4 5]
```

!! The copy should not be affected by the changes made to the original array

VIEW

Example:

```
import numpy as np
arr = np.array([1,2,3,4,5])
x = arr.view()
arr[0] = 42
print(arr) # output: [42 2 3 4 5]
print(x) # output: [42 2 3 4 5]
```

!! The view should be affected by the changes made to the original array and The original array should be affected by the changes made to the view

Numpy Array Iterating

Iterating means going through elements one by one

iterating on 1-D array

```
arr = np.array([1, 2, 3])
for x in arr:
    print(x)  # Output: 1
                        2
                        3
```

iterating on 2-D array

```
arr = np.array([[1, 2, 3], [4, 5, 6]])
for x in arr:
    print(x)  # Output : [1 2 3]
                         [4 5 6]
```

To return the actual values, we have to iterate the array in each dimension

```
for x in arr:
    for y in x:
        print(y)  # Output: 1
                            2
                            3
                            4
                            5
                            6
```

iterating on 3-D array

```
arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
for x in arr:  (x represent the 2D array)
    print(x)  # Output: [[1 2 3]
                         [4 5 6]]
                        [[7 8 9]
                         [10 11 12]]
```

To return the actual values, we have to iter
in each dimension.

```
for x in arr:
  for y in x:
    for z in y:
      print(z)  # Output: 1
                          2
                          3
                          4
                          5
                          6
                          7
                          8
                          9
                          10
                          11
                          12
```

Iterating Arrays Using nditer()

In basic for loops, iterating through each scalar of an array we need to use n for loops which can be difficult to write for arrays with very hight dimension.

So the nditer() function make it easier

Example:

```
import numpy as np
arr = np.array([[[1,2],[3,4]],[[5,6],[7,8]]])
for x in np.nditer(arr):
  print(x) # Output: 1
                     2
                     3
                     4
                     5
                     6
                     7
                     8
```

Numpy Joining Array

Joining means putting contents of two or more arrays in a single array

In Numpy we join arrays by axes

We pass a sequence of arrays that we want to join to the concatenate() function, along with the axis. If axis is not explicity passed, it is taken as 0.

Join two arrays

Example:

```
import numpy as np
arr1 = np.array([1,2,3])
arr2 = np.array([4,5,6])
arr = np.concatenate((arr1, arr2))
print(arr) # output: [1 2 3 4 5 6]
```

Join two 2-D arrays along rows (axis = 1)

Example:

```
import numpy as np
arr1 = np.array([[1,2],[3,4]])
arr2 = np.array([[5,6],[7,8]])
arr = np.concatenate((arr1, arr2), axis=1)
print(arr) # output: [[1 2 5 6]
                      [3 4 7 8]]
```

!! axis 0 : tahet ba3doun
   axis 1 : had ba3doun

Joining Arrays Using Stack Functions

Stacking is same as concatenation, the only difference is that stacking is done along a new axis.

We can concatenate two 1D arrays along the second axis wich would result in putting them one over the other (stacking).

We pass a sequence of arrays that we want to join to the stack() method along with the axis. If axis is not explicitly passed it is taken as

Example:

```
import numpy as np
arr1 = np.array([1,2,3])
arr2 = np.array([4,5,6])
arr = np.stack((arr1, arr2), axis = 1)
print(arr) # output : [[1 4]
                       [2 5]
                       [3 6]]
```

```
arr = np.stack((arr1, arr2), axis = 0)
print(arr) # output : [[1 2 3]
                       [4 5 6]]
```

Stacking Along Rows

Numpy provides a helper function : hstack()
to stack along rows

Example:

```
import numpy as np
arr1 = np.array([1,2,3])
arr2 = np.array([4,5,6])
arr = np.hstack((arr1, arr2))
print(arr) # output: [1 2 3 4 5 6]
```

Stacking Along Columns

Numpy provides a helper function: vstack()
to stack along columns

Example:

```
import numpy as np
arr1 = np.array([1,2,3])
arr2 = np.array([4,5,6])
arr = np.vstack((arr1, arr2))
print(arr) # output : [[1 2 3]
                       [4 5 6]]
```

Stacking Along Height (depth)

Numpy provides a helper function: dstack() to stack along depth

Example:

```
import numpy as np
arr1 = np.array([1,2,3])
arr2 = np.array([4,5,6])
arr = np.dstack((arr1, arr2))
print(arr) # output: [[1 4]
                      [2 5]
                      [3 6]]
```

→ NumPy Splitting Array

Splitting is reverse operation of Joining
Joining merges multiple arrays into one and Splitting breaks one array into multiple
We use array_split() for splitting arrays, we pass it the array we want to split and the numbre of splits

Example:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6])
# Split the array in 3 parts.
newarr = np.array_split(arr, 3)
print(newarr) # Output: [array([1,2]), array([3,4]),
                                              array([5,6])]
```

!! The return value is a list containing 3 arrays

```
# Split the array in 4 parts
newarr = np.array_split(arr, 4)
print(newarr)
# Output: [array([1,2]), array([3,4]), array([5]), array([6])]

# Accessing the splitted arrays
print(newarr[0]) # Output: [1 2]
print(newarr[1]) # Output: [3 4]
print(newarr[2]) # Output: [5]
print(newarr[3]) # Output: [6]
```

→ Splitting 2-D Arrays

We use the same method array_split(), and we pass in the array we want to split and the number of splits.

Example:

```
import numpy as np
arr = np.arr([[1,2],[3,4],[5,6]])
newarr = np.array_split(arr,3)
print(newarr)
# Output: [array([[1,2]]), array([[3,4]]), array([[5,6]])]
```

→ Numpy Searching Arrays

We can search an array for a certain value, and return the indexes that get a match.

To search an array, we use the where() method

Examples:

1)
```
import numpy as np
arr = np.array([1,2,3,4,5,4,4])
x = np.where(arr==4)
print(x) # Output: (array([3,5,6]),)
```
indexes

2)
```
import numpy as np
arr = np.array([1,2,3,4,5,6,7,8])
# Find the indexes where the values are even
x = np.where(arr%2 == 0)
print(x) # Output: (array([1,3,5,7]))
```

Search Sorted

- There is a method called searchSorted() wich perform a binary search in the array, and return the index where the specified value would be inserted to maintain the search order

- The searchsorted() method is assumed to be used on sorted arrays

Example:

Find the indexes where the value 7 should be inserted:

```
import numpy as np
arr = np.array([6,7,8,9])
x = np.searchSorted(arr, 7)
print(x) # Output : 1 (The nb 7 should be inserted on index 1 to remain the sort order)
```

Multiple Values

To search for multiple values, we use an array with the specified values

Example:

```
import numpy as np
arr = np.array([1,3,5,7])
x = np.searchSorted(arr, [2,4,6])
print(x) # Output: [1 2 3]
```

indexes

→ Numpy Array Sort

Sorting means putting elements in an ordered sequence
The Numpy ndarray object has a function called sort() that will sort a specified array
The method will returns a copy of the array leaving the original array unchanged.

Example:

```
import numpy as np
#. Sorting on array of integers
nb = np.array([3,2,0,1])
print(np.sort(nb))# output: [0 1 2 3]

# Sorting an array of strings
strings = np.array(['banana', 'cherry', 'apple'])
print(np.sort(strings))# output: ['apple', 'banana', 'cherry']

# Sorting an array of booleans
bools= np.array([True, False, True])
print(np.sort(bools) # output: [False, True, True]
```

↳ Sorting a 2-D Array

If we use the sort() method on a 2-D array, both arrays will be sorted

Example:

```
import numpy as np
arr= np.arr([[3,2,4],[5, 0, 1]])
print(np.sort(arr))# output: [[2 3 4]
                             [0 1 5]]
```

→ Numpy Filter Array

Getting some elements out of an existing array and creating a new array out of them is called filtering

In Numpy, we filter an array using a boolean index list (list of booleans corresponding to indexes in the array)

If the value at an index is True that element is contained in the filtered array, if the value at that index is False that element is excluded from the filtered array

Example:

Create an array from the elements on index 0 and 2:

```
import numpy as np
arr = np.array([41, 42, 43, 44])
x = [True, False, True, False]
newarr = arr[x]
print(newarr) # output: [41, 43]
```

↳ Creating the Filter Array

In the example above we hard-coded the boolean list, but the common use is to create a filter array based on conditions

Example:
Creat a filter array that will return only values > 42.

```
import numpy as np
arr = np.array[(41, 42, 43, 44)]
# Create an empty list
filter_arr = []
# go through each element in arr
for element in arr:
# if the element > 42, set the value to true, otherwise False
    if element > 42:
        filter_arr.append(True)
    else:
        filter_arr.append(False)

newarr = arr[filter_arr]
print(filter_arr) # output: [False, False, True, True]
print(newarr) # output: [43, 44]
```

- Creating Filter Directly From Array

We can directly substitute the array instead of the iterable variable in our condition.

Example:

```
import numpy as np
arr = np.array[(41, 42, 43, 44)
filter_arr = arr > 42
newarr = arr[filter_arr]
print(filter_arr) # output: [False, False, True, True]
print(newarr) # output: [43, 44]
```

→ Numpy Random Module

Numpy offer the random module to work with randomn numbers

Example:

```
From numpy import random
# Generate a random integer from 0 to 100
x = random.randint(100)
print(x) # Output: 33 (for example)
# Generate a random float between 0 and 1
x = random.rand()
print(x) # Output: 0.5322.. (for example)
```

↳ Generate Random Array

We can use both methods randint() or rand() to generate a random array.
In this case we will pass a size parameter to the method where we can specify the shape of the array

Example:

```
From numpy import random
# Generate a 1-D array containing 5 random integers
  from 0 to 100
x = random.randint(100, size=(5))
print(x) # Output: [22 66 56 97 23] (for ex)
```

Generate a 2-D array with 3 rows containing 5 random float

```
x = random.rand(3,5)
print(x) output: [[0.14  0.44  0.59  0.73  0.22]
                  [0.004 0.36  0.88  0.56  0.15]
                  [0.69  0.75  0.927 0.057 0.62]]
```

↳ Generate Random Number From Array

The choice() method allow us to generate a random value based on an array of values it take an array as parameter and randomly return on of values

Example:

```
From numpy import random
# Return one value in an array
x = random.choice([3,5,7,9])
print(x) #Output: 3 or 5 or 7 or 9
```

The choice() method also allow us to return an array of value by adding a size parameter

```
# Generate a 2-D array that consists of the values in the array parameter (3,5,7 and 9)
x = random.choice([3,5,7,9], size=(3,5))
print(x)#output: [[9 3 5 5 7]
                  [7 5 3 3 9]
                  [7 5 9 3 7]] (for example)
```

→ Random Data Distribution

Data Distribution is a list of all possible values, and how often each value occurs
The random module offer methods that returns randomly generated data distribution

↳ Random Distribution

A random distribution is a set of random numbers that follow a certain probability density function (function that describes a continuous probability i.e. probability of all values in an array).

We can generate random numbers based on defined probabilities using the choice() method of the random module
The choice() method allows us to specify the probability for each value.
The probality is set by a number between 0 and 1, where 0 means that the value will never occur and 1 means that the value will always occur.

Example:
Generate a 1-D array containing 100 values, where each value has to be 3, 5, 7, or 9
The probabilities are:
for 3: 0.1, For 5: 0.3, for 7: 0.6 and for 9: 0

```
From numpy import random
x = random.choice([3,5,7,9], p=[0.1,0.3,0.6,0.0]
                  , size=(10))
print(x) # output: [5 5 7 5 3 5 3 7 7 3]
```

!! The sum of all probability numbers should be 1

↳ also we can return arrays of any shape and size by specifying the shape in the size parameters

Example:

```
from numpy import random
x=random.choice([3,5,7,9], p=[0.1,0.3,0.6,0.0]
                , size=(2,3))
print(x) # output: [[5 7 3]
                   [3 7 7]]
```

→ **Random Permutations**

A permutation refers to an arrangment of elements
e.g. [3,2,1] is a permutation of [1,2,3]
and vice-versa.
The Numpy Random module provides two methods
For this: shuffle() and permutation()

Shuffling Arrays

Shuffle means changing arrangment of elements in-place i.e. in the array itself

Example:

```
from numpy import random
import numpy as np
arr = np.array([1,2,3,4,5])
random.shuffle(arr)
print(arr) # output: [4 5 3 2 1]
```

Generate Permutation of Arrays

The permutation() method a re-arranged array (and leaves the original array un-changed)

Example:

```
from numpy import random
import numpy as np
arr = np.array([1,2,3,4,5])
print(random.permutation(arr))
# output : [2 5 3 1 4]
```