NumPy is Numerical Python

- NumPy is a Python library used for working with arrays.
- In Python we have lists that serve the purpose of arrays, but they are slow to process.
- NumPy aims to provide an array object that is up to 50x faster than traditional Python lists.
- The array object in NumPy is called ndarray, it provides a lot of supporting functions that make working with ndarray very easy.
- NumPy arrays are stored at one continuous place in memory unlike lists, so processes can access and manipulate them very efficiently. This behavior is called locality of reference in computer science.
- This is the main reason why NumPy is faster than lists. Also it is optimized to work with latest CPU architectures.
- NumPy is a Python library and is written partially in Python, but most of the parts that require fast computation are written in C or C++.

```
#to install numpy library

!pip install numpy
```

```
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages (1.25.2)
```

```
import numpy as np
```

```
a = [1,2,3,"jameer"]
a
```

```
[1, 2, 3, 'jameer']
```

```
type(a)
```

```
list
```

```
b = np.array(a)
b
```

```
array(['1', '2', '3', 'jameer'], dtype='<U21')
```

```
np.__version__
```

```
'1.25.2'
```

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

arr = np.array(a)
arr
```

```
array(['1', '2', '3', 'jameer'], dtype='<U21')
```

```
arr.dtype
```

```
dtype('<U21')
```

```
arr.ndim
```

```
1
```

```
'''
  An array that has 0-D arrays as its elements is called uni-dimensional or 1-D array.
  These are the most common and basic arrays.
'''

arr1 = np.array([1,2,3,4,5,6,7,8,9,10])
arr1
```

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

```
arr1.ndim
```

```
1
```

```
'''
  An array that has 1-D arrays as its elements is called a 2-D array.
  These are often used to represent matrix or 2nd order tensors.
'''

arr2 = np.array([[1,2,3],[4,5,6]])
arr2
```

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

```
arr2.ndim
```

```
2
```

```
'''
  An array that has 2-D arrays (matrices) as its elements is called 3-D array.
  These are often used to represent a 3rd order tensor.
'''

arr3 = np.array([[[1,2,3],[4,5,6]], [[7,8,9],[10,11,12]]])
arr3
```

```
array([[[ 1,  2,  3],
        [ 4,  5,  6]],

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

```
arr3.ndim
```

```
3
```

```
arr3.dtype
```

```
dtype('int64')
```

```
arr4 = arr1 + arr1
arr4
```

```
array([ 2,  4,  6,  8, 10, 12, 14, 16, 18, 20])
```

```
arr[3]
```

```
'jameer'
```

```
'''
  Array indexing is the same as accessing an array element.
  You can access an array element by referring to its index number.
  The indexes in NumPy arrays start with 0, meaning that the first element has index 0, and the second has index
'''

arr1[2]
```

```
3
```

```
'''
  To access elements from 2-D arrays we can use comma separated integers representing the dimension and the index
'''

arr2[1,2]
```

```
6
```

```
arr3
```

```
array([[[ 1,  2,  3],
        [ 4,  5,  6]],

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

```
'''
  To access elements from 3-D arrays we can use comma separated integers representing the dimensions and the inde
'''

arr3[1,1,2]
```

```
12
```

Accessing array elements through negative indexing

```
arr[-2]
```

```
'3'
```

```
arr1[-3]
```

```
8
```

```
arr2[-1,2]
```

```
6
```

```
arr3[-2,-1,2]
```

```
6
```

```
arr2[1:2:2]
```

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

By default Python have these data types:

- strings - used to represent text data, the text is given under quote marks. e.g. "ABCD"
- integer - used to represent integer numbers. e.g. -1, -2, -3
- float - used to represent real numbers. e.g. 1.2, 42.42
- boolean - used to represent True or False.

- complex - used to represent complex numbers. e.g. 1.0 + 2.0j, 1.5 + 2.5j

Data Types in NumPy

NumPy has some extra data types, and refer to data types with one character, like i for integers, u for unsigned integers etc. Below is a list of all data types in NumPy and the characters used to represent them.

```
i - integer
b - boolean
u - unsigned integer
f - float
c - complex float
m - timedelta
M - datetime
O - object
S - string
U - unicode string
V - fixed chunk of memory for other type ( void )
```

```
#converting data type from one to another
print(arr1)
print(arr1.dtype)
arr4 = arr1.astype('f')
print(arr4)
print(arr4.dtype)
```

```
[ 1  2  3  4  5  6  7  8  9 10]
int64
[ 1.  2.  3.  4.  5.  6.  7.  8.  9. 10.]
float32
```

```
#defining datatype at array creation
arr5 = np.array([0,1,2,0,1,3], dtype='f')
arr5
```

```
array([0., 1., 2., 0., 1., 3.], dtype=float32)
```

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.

The copy owns the data and any changes made to the copy will not affect original array, and any changes made to the original array will not affect the copy.

The view does not own the data and any changes made to the view will affect the original array, and any changes made to the original array will affect the view.

```
print(arr2)
```

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

```
arr6 = arr2.copy()
print(arr6)
```

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

```
arr6[0,2] = 5
arr6
```

```
array([[1, 2, 5],
       [4, 5, 6]])
```

```
arr2
```

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

```
arr7 = arr2.view()
arr7
```

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

```
arr7[0,2] = 5
arr7
```

```
array([[1, 2, 5],
       [4, 5, 6]])
```

```
arr2
```

```
array([[1, 2, 5],
       [4, 5, 6]])
```

```
'''
  Every NumPy array has the attribute base that returns None if the array owns the data.
  Otherwise, the base  attribute refers to the original object.

  The copy returns None.
  The view returns the original array.
'''
print(arr6.base)
```

```
None
```

```
arr7.base
```

```
array([[1, 2, 5],
       [4, 5, 6]])
```

```
#The shape of an array is the number of elements in each dimension.
#NumPy arrays have an attribute called shape that returns a tuple with each index having the number of correspond

print(arr1.shape)
print(arr2.shape)
print(arr3.shape)
print(arr4.shape)
```

```
(10,)
(2, 3)
(2, 2, 3)
(10,)
```

```
'''
  Reshaping arrays
  Reshaping means changing the shape of an array.
  The shape of an array is the number of elements in each dimension.
  By reshaping we can add or remove dimensions or change number of elements in each dimension.
'''

arr1.reshape(5,2)
```

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

```
       [ 5,  6],
       [ 7,  8],
       [ 9, 10]])
```

```
arr2
```

```
array([[1, 2, 5],
       [4, 5, 6]])
```

```
arr2.reshape(2,3,1)
```

```
array([[[1],
        [2],
        [5]],

       [[4],
        [5],
        [6]]])
```

```
'''
  Flattening the arrays
  Flattening array means converting a multidimensional array into a 1D array.
  We can use reshape(-1) to do this.
'''

arr3 = arr3.reshape(-1)
arr3
```

```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12])
```

```
arr3.shape
```

```
(12,)
```

```
arr3.ndim
```

```
1
```

```
#Numpy.zeros() – returns an array of given shape and type filled with zeroes.

arr8 = np.zeros(10)
arr8
```

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

```
arr9 = np.zeros((4,5))
arr9
```

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

```
arr10 = np.zeros((3,4,5))
arr10
```

```
array([[[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.],
        [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.]]])
```

```
arr10.shape
```

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

```
arr10.ndim
```

```
3
```

```
arr10.dtype
```

```
dtype('float64')
```

```
arr11 = np.zeros((3,2,3), dtype='i')
arr11
```

```
array([[[0, 0, 0],
        [0, 0, 0]],

       [[0, 0, 0],
        [0, 0, 0]],

       [[0, 0, 0],
        [0, 0, 0]]], dtype=int32)
```

```
#Numpy.ones()- returns an array  of given shape and type filled with ones.

arr12 = np.ones(5)
arr12
```

```
array([1., 1., 1., 1., 1.])
```

```
arr13 = np.ones((2,3), dtype='U')
arr13
```

```
array([['1', '1', '1'],
       ['1', '1', '1']], dtype='<U1')
```

```
arr13.dtype
```

```
dtype('<U1')
```

```
arr13.ndim
```

```
2
```

```
arr13.shape
```

```
(2, 3)
```

```
arr14 = np.ones((4,3,2), dtype='i')
arr14
```

```
array([[[1, 1],
        [1, 1],
        [1, 1]],

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

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

        [[1, 1],
         [1, 1],
         [1, 1]]], dtype=int32)
```

```
arr14.ndim
```

```
3
```

```
arr14.dtype
```

```
dtype('int32')
```

```
arr14.shape
```

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

```
#Numpy.linspace()- returns equally spaced numbers within the given range based on the sample number.

arr15 = np.linspace(1,10,15)
arr15
```

```
array([ 1.        ,  1.64285714,  2.28571429,  2.92857143,  3.57142857,
        4.21428571,  4.85714286,  5.5       ,  6.14285714,  6.78571429,
        7.42857143,  8.07142857,  8.71428571,  9.35714286, 10.        ])
```

```
arr16 = np.linspace(2,12,16,retstep = True)
arr16
```

```
(array([ 2.        ,  2.66666667,  3.33333333,  4.        ,  4.66666667,
         5.33333333,  6.        ,  6.66666667,  7.33333333,  8.        ,
         8.66666667,  9.33333333, 10.        , 10.66666667, 11.33333333,
        12.        ]),
 0.6666666666666666)
```

```
arr17 = np.linspace(2,12,16,retstep = False)
arr17
```

```
array([ 2.        ,  2.66666667,  3.33333333,  4.        ,  4.66666667,
        5.33333333,  6.        ,  6.66666667,  7.33333333,  8.        ,
        8.66666667,  9.33333333, 10.        , 10.66666667, 11.33333333,
       12.        ])
```

```
arr18 = np.linspace(2,12,16,endpoint = False,retstep = True)
arr18
```

```
(array([ 2.   ,  2.625,  3.25 ,  3.875,  4.5  ,  5.125,  5.75 ,  6.375,
         7.   ,  7.625,  8.25 ,  8.875,  9.5  , 10.125, 10.75 , 11.375]),
 0.625)
```

```
type(arr18)
```

```
tuple
```

```
arr19 = np.linspace(2,12,16,endpoint = False,retstep = False).reshape((4,4))
arr19
```

```
array([[ 2.   ,  2.625,  3.25 ,  3.875],
       [ 4.5  ,  5.125,  5.75 ,  6.375],
       [ 7.   ,  7.625,  8.25 ,  8.875],
       [ 9.5  , 10.125, 10.75 , 11.375]])
```

```
type(arr19)
```

```
numpy.ndarray
```

```
#Numpy.arange()- returns equally spaced numbers with in the given range based on step size.

arr20 = np.arange(0,15,2)
arr20
```

```
array([ 0,  2,  4,  6,  8, 10, 12, 14])
```

```
#Numpy.random.rand()- returns an array of given shape filled with random values.

arr21 = np.random.rand(3,2)
arr21
```

```
array([[0.8843601 , 0.65093257],
       [0.27064088, 0.64335749],
       [0.09239051, 0.36464532]])
```

```
arr22 = np.random.rand(3,2,2)
arr22
```

```
array([[[0.36744638, 0.41749633],
        [0.06279292, 0.49319322]],

       [[0.1012238 , 0.3389282 ],
        [0.65718498, 0.30011636]],

       [[0.43535608, 0.30283205],
        [0.8353265 , 0.51519461]]])
```

```
type(arr22)
```

```
numpy.ndarray
```

```
#Numpy.logspace()- returns equally spaced numbers based on log scale.

arr23 = np.logspace(7,23,10, endpoint = True, base=8, dtype='f')
arr23
```

```
array([2.0971520e+06, 8.4551872e+07, 3.4089178e+09, 1.3743895e+11,
       5.5411915e+12, 2.2340683e+14, 9.0071993e+15, 3.6314753e+17,
       1.4641190e+19, 5.9029581e+20], dtype=float32)
```

```
arr23.dtype
```

```
dtype('float32')
```

```
arr24 = np.logspace(5,23,12, endpoint = True, base=2)
arr24
```

```
array([3.20000000e+01, 9.94820020e+01, 3.09270898e+02, 9.61465253e+02,
       2.98901526e+03, 9.29228818e+03, 2.88879822e+04, 8.98073221e+04,
       2.79194131e+05, 8.67962223e+05, 2.69833186e+06, 8.38860800e+06])
```

```
arr25 = np.logspace(4,24,20, endpoint = True, base=5)
arr25
```

```
array([6.25000000e+02, 3.40124500e+03, 1.85095481e+04, 1.00728813e+05,
       5.48165394e+05, 2.98311169e+06, 1.62340700e+07, 8.83456789e+07,
       4.80776478e+08, 2.61638175e+09, 1.42383286e+10, 7.74848702e+10,
       4.21672044e+11, 2.29473589e+12, 1.24879344e+13, 6.79592390e+13,
       3.69833635e+14, 2.01263168e+15, 1.09527255e+16, 5.96046448e+16])
```

```
arr26 = np.add(arr1,arr1)
arr26
```

```
array([ 2,  4,  6,  8, 10, 12, 14, 16, 18, 20])
```

```
arr27 = np.subtract(arr26, arr1)
arr27
```

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

```
arr28 = np.multiply(arr27, arr26)
arr28
```

```
array([  2,   8,  18,  32,  50,  72,  98, 128, 162, 200])
```

```
arr29 = np.divide(arr28, arr27)
arr29
```

```
array([ 2.,  4.,  6.,  8., 10., 12., 14., 16., 18., 20.])
```

```
arr30 = np.remainder(arr28, arr27)
arr30
```

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

NumPy Array Interation

```
for x in arr29:
  print(x)
```

```
2.0
4.0
6.0
8.0
10.0
12.0
14.0
16.0
18.0
20.0
```

```
#terating through each scalar of an array we need to use n for loops which can be difficult to write for arrays w

for x in np.nditer(arr25):
  print(x)
```

```
625.0
3401.2450022824587
18509.548104882244
100728.81277798011
548165.3936751197
2983111.6890587243
16234069.957858192
88345678.89739037
480776478.1167998
2616381749.455563
14238328559.038033
77484870163.65393
421672043834.6142
2294735890711.5317
12487934367745.29
67959238971400.22
369833635056629.8
2012631684659712.8
1.0952725534214646e+16
5.960464477539062e+16
```

```
arr22
```

```
array([[[0.36744638, 0.41749633],
        [0.06279292, 0.49319322]],

       [[0.1012238 , 0.3389282 ],
        [0.65718498, 0.30011636]],

       [[0.43535608, 0.30283205],
        [0.8353265 , 0.51519461]]])
```

```
for x in np.nditer(arr22[:, ::2]):
  print(x)
```

```
0.36744638428881327
0.4174963273040371
0.10122379968068762
0.33892819677791675
0.43535607988641345
0.3028320505462032
```

Joining of NumPy Arrays

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

In SQL we join tables based on a key, whereas in NumPy we join arrays by axes.

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

```
arr31 = np.array([1, 2, 3])
arr32 = np.array([4, 5, 6])
arr33 = np.concatenate((arr31, arr32))
print(arr33)
```

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

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

```
[[1 2 5 6]
 [3 4 7 8]]
```

```
arr1 = np.array([[1, 2], [3, 4]])
arr2 = np.array([[5, 6], [7, 8]])
arr = np.concatenate((arr1, arr2), axis=0)
print(arr)
```

```
[[1 2]
 [3 4]
 [5 6]
 [7 8]]
```

Splitting NumPy Arrays

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 number of splits.

```
arr4 = np.array_split(arr, 3)
arr4
```

```
[array([[1, 2],
        [3, 4]]),
 array([[5, 6]]),
 array([[7, 8]])]
```

Searching elements in NumPy Arrays

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

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

```
x = np.where(arr == 5)
x
```

```
(array([2]), array([0]))
```

Sorting of Arrays

Sorting means putting elements in an ordered sequence.

Ordered sequence is any sequence that has an order corresponding to elements, like numeric or alphabetical, ascending or descending.

The NumPy ndarray object has a function called sort(), that will sort a specified array.

```
arr5 = np.sort(arr3)
arr5
```

```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12])
```

```
arr
```

```
array([[1, 2],
       [3, 4],
       [5, 6],
       [7, 8]])
```

```
#transpose
arr6 = np.transpose(arr)
arr6
```

```
array([[1, 3, 5, 7],
       [2, 4, 6, 8]])
```

```
#append()-> adds value at the end of the array.

arr7 = np.append(arr6, [[9,10,11,12]], axis = 0)
arr7
```

```
array([[ 1,  3,  5,  7],
       [ 2,  4,  6,  8],
       [ 9, 10, 11, 12]])
```

```
#insert()-> adds values at a given position and axis in an array.

arr8 = np.insert(arr7,2,[5,5,5,5], axis = 0)
arr8
```

```
array([[ 1,  3,  5,  7],
       [ 2,  4,  6,  8],
       [ 5,  5,  5,  5],
       [ 9, 10, 11, 12]])
```

```
#delete()- > removes values at a given position and axis in an array.

arr9 = np.delete(arr8, 2, axis = 0)
arr9
```

```
array([[ 1,  3,  5,  7],
       [ 2,  4,  6,  8],
       [ 9, 10, 11, 12]])
```

```
arr = np.array([41, 42, 43, 44])
filter_arr = arr > 42
newarr = arr[filter_arr]
print(filter_arr)
print(newarr)
```

```
[False False  True  True]
[43 44]
```

```
# Vector addition
v1 = np.array([1, 2, 3])
v2 = np.array([4, 5, 6])
v_sum = v1 + v2
```

```
# Dot product
dot_product = np.dot(v1, v2)
dot_product
```

```
32
```

```
# Cross product (for 3D vectors)
cross_product = np.cross(v1, v2)
cross_product
```

```
array([-3,  6, -3])
```

```
np.linalg.norm(v1)
```

```
3.7416573867739413
```

```
# Solving linear equations Ax = b
A = np.array([[2, 1],
              [1, 1]])
b = np.array([3, 2])
x = np.linalg.solve(A, b)
x
```

```
array([1., 1.])
```

```
# Computing eigenvalues and eigenvectors
eigenvalues, eigenvectors = np.linalg.eig(A)
eigenvalues
```

```
array([2.61803399, 0.38196601])
```

```
eigenvectors
```

```
array([[ 0.85065081, -0.52573111],
       [ 0.52573111,  0.85065081]])
```