NUMPY

What is Numpy?

NumPy is an open source mathematical and scientific computing library for Python programming tasks. The name NumPy is shorthand for Numerical Python. The NumPy library offers a collection of high-level mathematical functions including support for multi-dimensional arrays, masked arrays and matrices.

What are the advantages of NumPy in Python?

Numpy uses less memory and storage space, which is the main advantage. In addition, NumPy offers better performance in terms of execution speed. However, it is easier and more convenient to use.

Week 1

a. Basic ndarray

An ndarray (n-dimensional array) is the core data structure of NumPy.

Program:

```
import numpy as np

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

print("Basic ndarray:", arr_basic)
```

output:

```
Basic ndarray: [1 2 3 4 5]
```

b. Array of Zeros

To create an array filled with zeros, use np.zeros().

Program:

```
import numpy as np

arr_zeros = np.zeros((3,3))

print("Array of Zeros:", arr_zeros)
```

Output:

```
Array of Zeros: [[0. 0. 0.]
                 [0. 0. 0.]
                 [0. 0. 0.]]
```

Program:

```
#array of zeros with datatype int

import numpy as np

arr_zeros = np.zeros((3,3), dtype = int)

print("Array of Zeros:", arr_zeros)
```

output:

```
Array of Zeros: [[0 0 0]
                 [0 0 0]
                 [0 0 0]]
```

c. Array of Ones

To create an array filled with ones, use np.ones().

Program:

```
import numpy as np
arr_ones_2d = np.ones((2, 3))  # 2x3 array of ones
print("2D Array of Ones:", arr_ones_2d)
```

output:

```
2D Array of Ones: [[1. 1. 1.]
                   [1. 1. 1.]]
```

#Array of ones with datatype=int

Program:

```
import numpy as np
arr_ones_2d = np.ones((2, 3), dtype = int)  # 2x3 array of ones
print("2D Array of Ones:", arr_ones_2d)
```

output:

```
2D Array of Ones: [[1 1 1]
                   [1 1 1]]
```

d. Random Numbers in ndarray

You can create arrays of random numbers using NumPy's random module.

Generating a 3x3 array with random numbers from a normal distribution

Program:

```
import numpy as np
arr_normal = np.random.randn(3, 3)
print("array with normal distribution", arr_normal)
```

output:

```
array with normal distribution [[ 0.70010265  0.24423018  0.6645415 ]
                                [-0.61964002 -0.37546671  1.4069859 ]
                                [ 0.56407729  0.08975851 -1.89958066]]
```

Generating a 2x3 array with random integers from 1 to 10

program

```
import numpy as np
arr_integers = np.random.randint(1, 10, (2, 3))
print("A (2,3) matrix with random numbers from 1 to 9", arr_integers)
```

Output: (example)

```
A (2,3) matrix with random numbers from 1 to 9 [[1 5 8]
                                                [5 1 4]]
```

e. An Array of Your Choice

Description: You can create an array of any values you want. For example:

Program:

```
# Creating a custom array
arr_custom = np.array([3.14, 1.59, 2.65, 3.58])
print("Custom Array:", arr_custom)
```

output:

```
Custom Array: [3.14 1.59 2.65 3.58]
```

f. Matrix (Imatrix) in NumPy

An Imatrix creates an array with ones in its diagonals.

Program:

```
import numpy as np
arr = np.eye(3)
print("The identity matrix is:", arr)
```

output:

```
The identity matrix is: [[1. 0. 0.]
                         [0. 1. 0.]
                         [0. 0. 1.]]
```

Program:

```
import numpy as np
arr = np.eye(3, dtytpe = int)
print("The identity matrix is:", arr)
```

output:

```
The identity matrix is: [[1 0 0]
                         [0 1 0]
                         [0 0 1]]
```

g. **Evenly Spaced ndarray**

You can create an array with evenly spaced values using np.linspace() or np.arange().

- Using np.linspace():

Create 5 values from 0 to 10, evenly spaced

Program:

```
import numpy as np

arr_linspace = np.linspace(0, 10, 5)

print("Evenly Spaced Array (Linspace):", arr_linspace)
```

output:

```
Evenly Spaced Array (Linspace): [ 0.   2.5  5.   7.5 10. ]
```

- Using np.arange():

Create an array of values starting from 0 to 10 with a step of 2

Program:

```
import numpy as np

arr_arange = np.arange(0, 10, 2)

print("Evenly Spaced Array (Arange):", arr_arange)
```

Output:

```
Evenly Spaced Array (Arange): [0 2 4 6 8]
```

Each of these will create an evenly spaced sequence in the range provided, with np.linspace() allowing for a specified number of points, while np.arange() uses a step size.

EXERCISE-2

The Shape and Reshaping of NumPy Array:

a) Dimensions of NumPy Array:

Aim: To determine and display the dimensions of a NumPy array

Description: The dimensions of a NumPy array represent how the data is organized across different levels, like rows, columns, or more. **ndim** specifies the total number of dimensions. For example, a 2D array has rows and columns, while a 3D array has an additional depth level.

Program:

```
import numpy as np
arr=np.array([[1,2,3],[4,5,6]])
print(arr)
print("Dimension of an arr is:", arr.ndim)
arr1=np.array([[[1,2,3]]])
print(arr2)
print("Dimension of an arr1 is:",arr1.ndim)
```

Output:

```
[[1 2 3]
 [4 5 6]]
Dimension of  an arr is: 2
[[[1 2 3]]]
Dimension of  an arr1 is: 3
```

b) Shape of NumPy Array:-

Aim: To determine and display the shape of a NumPy array

Description: The **NumPy ndarray.shape attribute** is used to get the shape of a NumPy array. It returns a tuple representing the dimensions of the array, where each value in

the tuple represents the size of the array along that dimension.

Program:

```
import numpy as np
arr=np.array([[1,2,3],[4,5,6]])
print(arr)
print("Shape of an arr is:",arr.shape)
```

Output:-

```
[[1 2 3]
 [4 5 6]]
Shape of an arr is: (2, 3)
```

c) Size of NumPy Array:-

Aim: To display the size of a NumPy Array

Description: To find the number of elements in the NumPy array we use ndarray.size() method of the NumPy library in Python.It displays the number of elements present in an array.

Program:

```
import numpy as np
arr=np.array([[1,2,3],[4,5,6]])
print(arr)
print("Size of an arr is:",arr.size)
```

Output:

```
[[1 2 3]
 [4 5 6]]
Size of an arr is: 6
```

d) Reshaping a NumPy Array:-

Aim: To reshape a NumPy array into a specified shape without changing its data.

Description: Gives a new shape to the array without changing the data. It adjusts the array's dimensions while keeping the original data intact, allowing it to be reorganized for different tasks or operations.

Program:

```
import numpy as np
arr=np.array([1,2,3,4,5,6])
print("Reshaping of an arr is:",arr.reshape(2,3))
```

Output:

```
Reshaping of an arr is:
[[1 2 3]
 [4 5 6]]
```

e) Flattening a NumPy Array:-

Aim: To convert a multidimensional array into 1D array

Description: Flattening a NumPy array means converting a multi-dimensional array into a one-dimensional array. You can flatten a NumPy array using the .flatten() method.

Program:

```
import numpy as np
arr=np.array([[1,2,3],[4,5,6]])
print("Flattening of an arr is:",arr.flatten())
```

Output:

```
Flattening of an arr is: [1 2 3 4 5 6]
```

f) Transpose of a NumPy Array:-

Aim: To transpose a NumPy Array

Description: Transposing a NumPy array means flipping it over its diagonal, switching the rows and columns. This is useful for matrix operations, such as when you need to change the orientation of data or perform linear algebra operations.

You can transpose a NumPy array using the .T attribute or the np.transpose() function.

Program:

```
import numpy as np
arr=np.array([[1,2,3],[4,5,6]])
print("Transpose of an arr is:",arr.transpose())
```

Output:

```
Transpose of an arr is:
[[1 4]
 [2 5]
 [3 6]]
```

WEEK – 3

3.Expanding and Squeezing a NumPy Array

A . Expanding a NumPy Array.

Description: In NumPy, expanding an array refers to adding a new dimension (or axis) to the array's shape. This is often done to make the array compatible with certain operations or functions that require a specific number of dimensions.

Program:

```
#expand 1D to 2D
import numpy as np
arr = np.array([1,2,3])
print("The number of dimensions before expansion:",arr.ndim)
print("The shape of an array before expansion:",  arr.shape)
expand_arr = np.expand_dims(arr , axis = 0)
print("expanded array:", expand_arr)
print("The number of dimensions after expansion:", expand_arr.ndim)
print("The shape of an array after expansion:",  expand_arr.shape)
```

Output:

```
The number of dimensions before expansion: 1
The shape of an array before expansion: (3,)
expanded array: [[1 2 3]]
The number of dimensions after expansion: 2
The shape of an array after expansion: (1, 3)
```

#expand 2D to 3D

```
import numpy as np
arr = np.array([[1,2,3],[4,5,6]])
print("The number of dimensions before expansion:",arr.ndim)
print("The shape of an array before expansion:",  arr.shape)
expand_arr = np.expand_dims(arr , axis = 0)
```

```
print("expanded array:", expand_arr)

print("The number of dimensions after expansion:", expand_arr.ndim)

print("The shape of an array after expansion:",  expand_arr.shape)
```

Output:

```
The number of dimensions before expansion: 2

The shape of an array before expansion: (2, 3)

expanded array: [[[1 2 3]

  [4 5 6]]]

The number of dimensions after expansion: 3

The shape of an array after expansion: (1, 2, 3)
```

B . Squeezing a NumPy Array

Description: In NumPy , squeezing an array means removing any dimensions of size 1 from the array . This is useful when you have an array with extra singleton dimensions that you don't need.

Program:

```
#squeeze 3D to 2D

import numpy as np

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

print ("The array is:", arr)

print("The number of dimensions before squeezing:", arr.ndim)

squeeze_arr = np.squeeze(arr)

print("Array after squeezing is:", squeeze_arr)

print("The number of dimensions after squeezing is:", squeeze_arr.ndim)
```

Output:

```
The array is: [[[1 2 3]

  [4 5 6]]]

The number of dimensions before squeezing: 3

Array after squeezing is: [[1 2 3]
```

[4 5 6]]

The number of dimensions after squeezing is: 2

#squeezing 2D to 1D

```
import numpy as np
# Squeeze 2D to 1D
arr=np.array([[1,2,3],[4,5,6]])
print("The array is:", arr)
print("The number of dimensions of input array are:", arr.ndim)
print("The shape of input array is:", arr.shape)
arr1=arr.flatten()
print(arr1)
print("The number of dimensions after squeezing:", arr1.ndim)
print("The shape of array after squeezing is:", arr1.shape)
```

Output:

The array is: [[1 2 3]

[4 5 6]]

The number of dimensions of input array are: 2

The shape of input array is: (2, 3)

[1 2 3 4 5 6]

The number of dimensions after squeezing: 1

The shape of array after squeezing is: (6,)

C . Sorting in NumPy Arrays

Description: Sorting in NumPy arrays involves arranging the elements of an array in ascending or descending order. NumPy provides the numpy.sort() function and related methods to perform sorting operations efficiently.

Program:

```
import numpy as np
#sorting 1D array
```

```
arr = np.array([34,43,32,53,23])
print("Array after sorting is:", np.sort(arr))
#sorting 2D array
#column wise
arr1 = np.array([[34,43,32,53,23],[12,24,35,64,31]])
print("column wise sorting:", np.sort(arr1, axis = 0))
#row wise
arr1 = np.array([[34,43,32,53,23],[12,24,35,64,31]])
print("row wise sorting:", np.sort(arr1, axis = 1))
```

Output:

```
Array after sorting is: [23 32 34 43 53]
column wise sorting: [[12 24 32 53 23]
 [34 43 35 64 31]]
row wise sorting: [[23 32 34 43 53]
 [12 24 31 35 64]]
```

WEEK-4

Indexing and Slicing of NumPy Array

1. Slicing 1-D NumPy arrays

Description:
Slicing in 1-D NumPy arrays extracts a subset of the array using the slice notation `array[start:end:step]`. It allows retrieving specific parts of the array.

Program:

```
import numpy as np
# Creating a 1-D NumPy array
arr = np.array([10, 20, 30, 40, 50, 60])
# Slicing the array
sliced_arr = arr[1:5]  # Elements from index 1 to 4
print("Slicing 1-D Array:", sliced_arr)
```

Output:
Slicing 1-D Array: [20 30 40 50]

2. Slicing 2-D NumPy arrays

Description:
In 2-D NumPy arrays, slicing extracts submatrices using the format `array[row_start:row_end, col_start:col_end]`.

Program:

```
import numpy as np
# Creating a 2-D NumPy array
arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
# Slicing rows and columns
sliced_arr = arr[1:, 1:]  # Elements from row 1 and column 1 onwards
print("Slicing 2-D Array:\n", sliced_arr)
```

Output:
Slicing 2-D Array:
[[5 6]
 [8 9]]

3. Slicing 3-D NumPy arrays

Description:
Slicing in 3-D NumPy arrays involves selecting submatrices using `array[depth_start:depth_end, row_start:row_end, col_start:col_end]`.

Program:

```
import numpy as np

# Creating a 3-D NumPy array
arr = np.array([[[1, 2, 3], [4, 5, 6]],
```

```
        [[7, 8, 9], [10, 11, 12]]])
# Slicing 3-D array
sliced_arr = arr[:, 0, 1:]  # All depths, row 0, columns from 1 onwards
print("Slicing 3-D Array:\n", sliced_arr)
```

Output:

```
Slicing 3-D Array:
[[2 3]
 [8 9]]
```

4. Negative slicing of NumPy arrays

Description:

Negative slicing retrieves elements in reverse order using negative indices. It can be applied to all dimensions.

Program:

```
import numpy as np
# Creating a 1-D NumPy array
arr = np.array([10, 20, 30, 40, 50, 60])
# Negative slicing
sliced_arr = arr[-4:-1]  # Elements from 4th last to 2nd last
print("Negative Slicing of Array:", sliced_arr)
```

Output:

```
Negative Slicing of Array: [30 40 50]
```

WEEK 5

Stacking and concatenating Numpy Arrays

a)Stacking ndarrays

Description: In Python, stacking ndarrays refers to combining multiple arrays into a single ndarray along a specified axis. The numpy library provides functions like np.stack(), np.hstack(), and np.vstack() to stack arrays either horizontally (along columns), vertically (along rows), or along a new axis. These functions enable efficient manipulation of multi-dimensional arrays, facilitating tasks like data organization or transformation in scientific computing.

Program:

```
import numpy as np
a=np.arange(10,20).reshape(2,5)
print("Array a is:", a)
b=np.arange(10).reshape(2,5)
print("Array b is:", b)
c=np.stack((a,b), axis = 0)
print("stacking with axis=0 is:", c)
d=np.stack((a,b), axis = 1)
print("stacking with axis=1 is:", d)
e=np.vstack((a,b))
print("vertical stacking:",e)
f=np.hstack((a,b))
print("horizontal stacking:",f)
g=np.dstack((a,b))
print("depth stacking:", g)
```

Output:

```
Array a is: [[10 11 12 13 14]
 [15 16 17 18 19]]
Array b is: [[0 1 2 3 4]
 [5 6 7 8 9]]
stacking with axis=0 is: [[[10 11 12 13 14]
  [15 16 17 18 19]]


 [[ 0  1  2  3  4]
  [ 5  6  7  8  9]]]
```

```
stacking with axis=1 is: [[[10 11 12 13 14]
  [ 0  1  2  3  4]]

 [[15 16 17 18 19]
  [ 5  6  7  8  9]]]
vertical stacking: [[10 11 12 13 14]
 [15 16 17 18 19]
 [ 0  1  2  3  4]
 [ 5  6  7  8  9]]
horizontal stacking: [[10 11 12 13 14  0  1  2  3  4]
 [15 16 17 18 19  5  6  7  8  9]]
depth stacking: [[[10  0]
  [11  1]
  [12  2]
  [13  3]
  [14  4]]

 [[15  5]
  [16  6]
  [17  7]
  [18  8]
  [19  9]]]
```

b) Concatenating ndarrays

Description:

In Python, concatenating ndarrays refers to joining multiple arrays along a specified axis. The numpy library provides the np.concatenate() function, which allows arrays to be merged either horizontally or vertically. Unlike stacking, concatenation works with arrays of compatible shapes and can join them along existing dimensions, enabling seamless data manipulation.

Program:

```
import numpy as np
a=np.arange(10,20).reshape(2,5)
print("array a is:",a)
b=np.arange(10).reshape(2,5)
print("array b is:", b)
c=np.vstack((a,b))
print("vertical stacking:", c)
#It is similar to vstack
d=np.concatenate((a,b),axis=0)
```

```
print("concatenating two arrays w.r.t axis= 0:",d)
e=np.hstack((a,b))
print("horizontal stack:", e)
#It is similar to hstack
f=np.concatenate((a,b),axis=1)
print("concatenating two arrays w.r.t axis= 1:",f)
```

Output:

```
array a is: [[10 11 12 13 14]
 [15 16 17 18 19]]
array b is: [[0 1 2 3 4]
 [5 6 7 8 9]]
vertical stacking: [[10 11 12 13 14]
 [15 16 17 18 19]
 [ 0  1  2  3  4]
 [ 5  6  7  8  9]]
concatenating two arrays w.r.t axis= 0: [[10 11 12 13 14]
 [15 16 17 18 19]
 [ 0  1  2  3  4]
 [ 5  6  7  8  9]]
horizontal stack: [[10 11 12 13 14  0  1  2  3  4]
 [15 16 17 18 19  5  6  7  8  9]]
concatenating two arrays w.r.t axis= 1: [[10 11 12 13 14  0  1  2  3  4]
 [15 16 17 18 19  5  6  7  8  9]]
```

c) BroadCasting in Numpy Arrays:

Description: Broadcasting in NumPy refers to the automatic alignment of arrays with different shapes for element-wise operations. It allows NumPy to perform arithmetic operations on arrays of varying dimensions by "stretching" smaller arrays across the larger ones without duplicating data. This feature simplifies code and optimizes performance by eliminating the need for explicit loops when performing operations on arrays of different shapes.

Program:

```
import numpy as np
a=np.array([[1,2,3],[4,5,6]])
print("array a is:", a)
b=np.array([3,4,6])
```

```
print("array b is:",b)
#addition
print("addition of two arrays:", a+b)
c=np.array([3])
#subtraction
print("subtraction of two arrays:", a-b)
d=np.array([[1,2,3],[2,5,6],[7,8,5]])
#Multiplication
print("multiplication of two arrays:", d*c)
#Division
print("division of two arrays:", d/c)
#modulus
print("modulus of two arrays:", a%b)
```

Output:

```
array a is: [[1 2 3]
 [4 5 6]]
array b is: [3 4 6]
addition of two arrays: [[ 4  6  9]
 [ 7  9 12]]
subtraction of two arrays: [[-2 -2 -3]
 [ 1  1  0]]
multiplication of two arrays: [[ 3  6  9]
 [ 6 15 18]
 [21 24 15]]
division of two arrays: [[0.33333333 0.66666667 1.]
                         [0.66666667 1.66666667 2.]
                         [2.33333333 2.66666667 1.66666667]]
modulus of two arrays: [[1 2 3]
 [1 1 0]]
```