CSE1006

Foundations of Data Analytics

Module - 5

Using Python for Data Science

- Overview of Python,Introduction to NumPy, NumPy standard data types, the basics of NumPy Arrays: NumPy Array Attributes, Array Indexing: Accessing Single Elements, Array Slicing: Accessing Subarrays, Reshaping of Arrays, Array Concatenation and Splitting, Aggregations,Computations on Arrays, NumPy's Structured arrays

Introduction to NumPy

- **NumPy** stands for Numerical Python.
- NumPy is a Python library used for working with arrays.
- It also has functions for working in domain of linear algebra, Fourier transform, and matrices.
- NumPy was created in 2005 by Travis Oliphant. It is an open-source project and you can use it freely.
- 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++.

Uses of NumPy:

- 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.
- Arrays are very frequently used in data science, where speed and resources are very important.

Why is NumPy Faster Than Lists?

- NumPy arrays are stored at one continuous place in memory unlike lists, so processes can access and manipulate them very efficiently.
- This behaviour 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.

Getting started with NumPy:

- If you have Python and PIP already installed on a system, then installation of NumPy is very easy.
- Install it using this command:

```
C:\Users\Your Name>pip install numpy
```

- If this command fails, then use a python distribution that already has NumPy installed like, Anaconda, Spyder etc.

NumPy standard Datatypes

Data Types in Python

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.

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)

NumPy Arrays

- Python lists are a substitute for arrays, but they fail to deliver the performance required while computing large sets of numerical data.
- To address this issue we use the NumPy library of Python. NumPy offers an array object called **ndarray**.
- They are similar to standard Python sequences but differ in certain key factors.

What is a NumPy Array?

- NumPy array is a multi-dimensional data structure that is the core of scientific computing in Python.
- All values in an array are homogenous (of the same data type).
- They provide efficient memory management, support various data types and are flexible with Indexing and slicing.

Dimensions in Arrays

NumPy arrays can have multiple dimensions, allowing users to store data in multilayered structures..

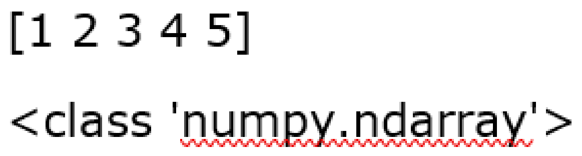
0D (zero-dimensional)	Scalar – A single element
1D (one-dimensional)	Vector- A list of integers.
2D (two-dimensional)	Matrix- A spreadsheet of data
3D (three-dimensional)	Tensor- Storing a color image

Create a NumPy ndarray Object

- 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.

Example:

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



```
[1 2 3 4 5]
<class 'numpy.ndarray'>
```

type(): This built-in Python function tells us the type of the object passed to it. Like in above code it shows that arr is numpy.ndarray type. 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**.

```
arr = np.array((1, 2, 3, 4, 5))
print(arr)
```

Check Number of Dimensions:

NumPy Arrays provides the **ndim** attribute that returns an integer that tells us how many dimensions the array have

Example:

```
import numpy as np
a = np.array(42)
b = np.array([1, 2, 3, 4, 5])
c = np.array([[1, 2, 3], [4, 5, 6]])
d = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])
print(a.ndim)                    #0
print(b.ndim)                    #1
print(c.ndim)                    #2
print(d.ndim)                    #3
```

NumPy Array Indexing

Accessing Array Elements:

- Array indexing is as 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 1 etc.

Example:

```
import numpy as np
arr = np.array([1, 2, 3, 4])
for i in range(0,4):
        print(arr[i]) # 1 2 3 4
```

```
import numpy as np
arr = np.array([1, 2, 3, 4])
print(arr[0])
print(arr[1])
print(arr[2] + arr[3])  #7
```

Access 2-D Arrays:

To access elements from 2-D arrays we can use comma separated integers representing the dimension and the index of the element.

Example:

```
import numpy as np
arr1 = np.array([[1,2,3,4,5], [6,7,8,9,10]])
# 2nd element on 1st row: 2
print('2nd element on 1st row: ', arr1[0, 1])    # 2
print('5th element on 2nd row: ', arr1[1, 4])   # 10
```

Access 3-D Arrays:

To access elements from 3-D arrays we can use comma separated integers representing the dimensions and the index of the element.

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])        #6
print(arr[1, 0, 2])        #9
```

Negative Indexing: Use negative indexing to access an array from the end.

Example:

```
import numpy as np
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])
print('Last element from 2nd dim: ', arr[1, -1])        #10
```

NumPy Array Slicing

Slicing arrays:

- Slicing in python means taking elements from one given index to another given index.
- We pass slice instead of index like this: [start:end].
- We can also define the step, like this: [start:end:step].
- If we don't pass start its considered 0
- If we don't pass end its considered length of array in that dimension
- If we don't pass step its considered 1

Example:

```
import numpy as np

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

print(arr[1:5])                    # 2 3 4 5
print(arr[4:])                     # 5 6 7
```

Slicing 1D arrays:

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

Example:

Slice elements from the beginning to index 4 (not included):

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

Use the minus operator to refer to an index from the end:

```
print(arr[-3:-1])                 # 5 6
```

Use the step value to determine the step of the slicing:

```
print(arr[1:6:2])                 # 2 4 6
```

Return every other element from the entire array:

```
print(arr[::2])                   # 1 3 5 7
```

Slicing 2D arrays:

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

Example:

From the second element, slice elements from index 1 to 4 (not included)

```
print(arr[1,1:4])                    # 7 8 9
```

From both elements, return index 2:

```
print(arr[0:2,2])                    # 3 8
```

From both elements, slice index 1 to index 4, this will return a 2-D array:

```
print(arr[0:2, 1:4])                 # [[2 3 4]  [7 8 9]]
```

Return every other element from the entire array:

```
print(arr[::2])                      # [[1 2 3 4 5]]
```

NumPy Array Shape

- 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 corresponding elements.

Example:

```
#Print the shape of a 2-D array
import numpy as np
arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
print(arr.shape)                          # (2,4)
```

- The example above returns (2, 4), which means that the array has 2 dimensions, where the first dimension has 2 elements and the second has 4.

```
arr = np.array([1, 2, 3, 4], ndmin=5)
print(arr)                                # [[[[[1 2 3 4]]]]]
print('shape of array :', arr.shape)      # (1, 1, 1, 1, 4)
```

- In the above example, Creating an array with 5 dimensions using ndmin using a vector with values 1,2,3,4 and observe that last dimension has value 4

NumPy Array Reshaping

- 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.

Example: Reshape From 1-D to 2-D

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

Output:

```
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
```

Example: Reshape From 1-D to 2-D

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

Output:

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

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

Note : We can reshape an 8 elements 1D array into 4 elements in 2 rows 2D array but we cannot reshape it into a 3 elements 3 rows 2D array as that would require 3x3 = 9 elements.

- You are allowed to have one "unknown" dimension.
- Meaning that you do not have to specify an exact number for one of the dimensions in the reshape method.
- Pass -1 as the value, and NumPy will calculate this number for you.

Example:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])
newarr = arr.reshape(2, 2, -1)   #observe that more than one -1 can’t be given
print(newarr)
```

Output:

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

 [[5 6]
  [7 8]]]
```

Flattening the arrays

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

Example:

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

Output:

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

Iterating Arrays

- Iterating means going through elements one by one.
- As we deal with multi-dimensional arrays in NumPy, we can do this using basic for loop of python.
- If we iterate on a 1-D array it will go through each element one by one.

Example:

```
import numpy as np
arr = np.array([1, 2, 3])
for x in arr:
        print(x)
```

Output:

```
1
2
3
```

Example: Iterate 2-D array

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

Output:

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

Example: Iterate 2-D array element wise

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

Output:

```
1
2
3
4
5
6
```

Example: Iterate 3-D array

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

Output:

```
[[1 2 3]
 [4 5 6]]
[[ 7  8  9]
 [10 11 12]]
```

Example: Iterate 3-D array element wise

```
import numpy as np
arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
for x in arr:
    for y in x:
        for z in y:
            print(z)
```

Array Concatenation

- 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() function, along with the axis. If axis is not explicitly passed, it is taken as 0.

Example: Join two 1D arrays

```
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]
```

Example: Join two 2D arrays (axis=1):

```
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]]
```

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

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 1-D arrays along the second axis which would result in putting them one over the other, i.e, 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 0.

Example: Join two 1D arrays

```
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]]
```

Array Splitting

- 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.

Example: **Split the array in 3 parts**

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

Output:

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

- If the array has less elements than required, it will adjust from the end accordingly.

Example: **Split the array in 4 parts**

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

Output:

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

- Use the same syntax when splitting 2-D arrays.
- Use the array_split() method, pass in the array you want to split and the number of splits you want to do.

Example: **Split the 2-D array into three 2-D arrays.**

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

Output:

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

Array Aggregations

- NumPy is a powerful library in Python for numerical and mathematical operations, and it provides various aggregation functions to perform operations on arrays.
- Aggregation functions in NumPy allow you to perform computations across the entire array or along a specified axis. Here are some commonly used NumPy aggregation functions.

numpy.sum():

This function returns the sum of array elements over the specified axis.

Syntax : numpy.sum(arr, axis, dtype, out)

arr : input array.

axis : axis along which we want to calculate the sum value. Otherwise, it will consider arr to be flattened(works on all the axis). axis = 0 means along the column and axis = 1 means working along the row.

out : Different array in which we want to place the result. The array must have same dimensions as expected output. Default is None.

dtype : [data-type, optional]Type we desire while computing mean.

Example:

```
import numpy as np
arr = np.array([[1, 2, 3], [4, 5, 6]])
# Sum of all elements in the array
total_sum = np.sum(arr)
print(total_sum)                                  #21
# Sum along a specific axis (axis=0 for columns, axis=1 for
rows)
column_sum = np.sum(arr, axis=0)
row_sum = np.sum(arr, axis=1)
print(column_sum)                                 #[ 5 7 9 ]
print(row_sum)                                    #[ 6  15 ]
```

numpy. mean():

Compute the arithmetic mean (average) of the given data (array elements) along the specified axis

Syntax : numpy.mean(arr, axis, out)

arr : input array.

axis : axis along which we want to calculate the sum value. Otherwise, it will consider arr to be flattened(works on all the axis). axis = 0 means along the column and axis = 1 means working along the row.

out : Different array in which we want to place the result. The array must have same dimensions as expected output. Default is None.

Example:

```
# Python Program illustrating numpy.mean() method
import numpy as np
arr = [20, 2, 7, 1, 34]
print("arr : ", arr)                #arr :  [20, 2, 7, 1, 34]
print("mean of arr : ", np.mean(arr))  #mean of arr :  12.8
```

numpy. median():

Compute the median of the given data (array elements) along the specified axis.

Syntax : numpy.median(arr, axis, out)

arr : input array.

axis : axis along which we want to calculate the sum value. Otherwise, it will consider arr to be flattened(works on all the axis). axis = 0 means along the column and axis = 1 means working along the row.

out : Different array in which we want to place the result. The array must have same dimensions as expected output. Default is None.

Example:

```
# Python Program illustrating numpy.mean() method
import numpy as np
arr = [20, 2, 7, 1, 34]
print("arr : ", arr)                #arr :  [20, 2, 7, 1, 34]
print("mean of arr : ", np.median(arr))  #median of arr :  7.0
```

- **numpy.min and numpy.max**: Compute the minimum and maximum values of an array
- **numpy.ceil()** is a mathematical function that returns the ceil of the elements of array.
- **numpy.floor()** is a mathematical function that returns the floor of the elements of array.
- **numpy.fix()** is a mathematical function that rounds elements of the array to the nearest integer towards zero.
- **numpy.exp()** : This mathematical function helps user to calculate exponential of all the elements in the input array.
- **np.sort()** : This is a function that performs sorting of array elements.

Computations on Numpy Arrays

- The below numpy functions are used to perform arithmetic operations on array in NumPy

np.add(): Compute the addition of 2 arrays

np.subtract() : Compute the subtraction of 2 arrays

np.multiply() : Compute the multiplication of 2 arrays

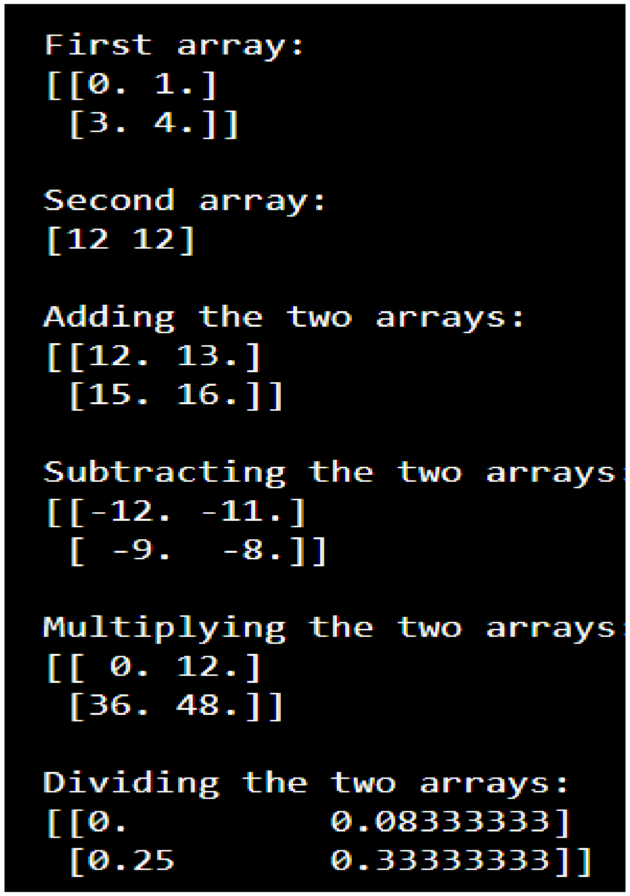
np.divide() : Compute the division of 2 arrays

numpy.power() : This function treats elements in the first input array as the base and returns it raised to the power of the corresponding element in the second input array.

numpy.mod() This function returns the remainder of division of the corresponding elements in the input array. The function numpy.remainder() also produces the same result.

Example: Arithmetic Operations

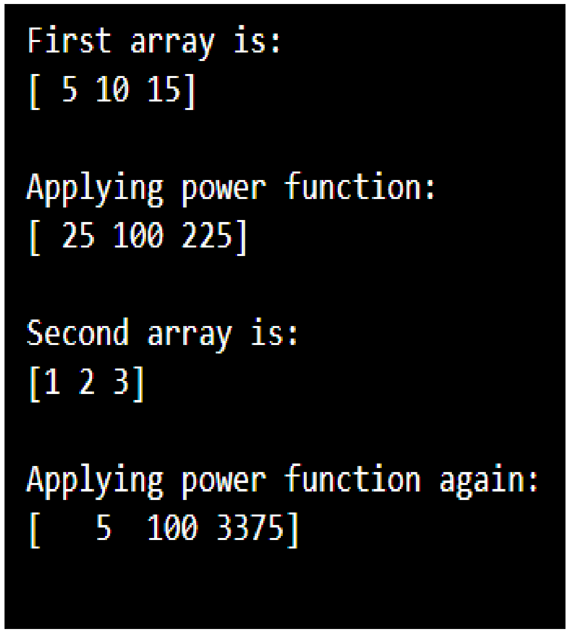
```
# Python code to perform arithmetic operations on NumPy array
import numpy as np
arr1 = np.array([[0.0, 1.0], [3.0, 4.0]])
print('First array:’)
print(arr1)
print('\nSecond array:’)
arr2 = np.array([12, 12])
print(arr2)
print('\nAdding the two arrays:’)
print(np.add(arr1, arr2))
print('\nSubtracting the two arrays:’)
print(np.subtract(arr1, arr2))
print('\nMultiplying the two arrays:’)
print(np.multiply(arr1, arr2))
print('\nDividing the two arrays:’)
print(np.divide(arr1, arr2))
```



```
First array:
[[0. 1.]
 [3. 4.]]

Second array:
[12 12]

Adding the two arrays:
[[12. 13.]
 [15. 16.]]

Subtracting the two arrays
[[-12. -11.]
 [ -9.  -8.]]

Multiplying the two arrays
[[ 0. 12.]
 [36. 48.]]

Dividing the two arrays:
[[0.         0.08333333]
 [0.25       0.33333333]]
```

Example: Power Operations

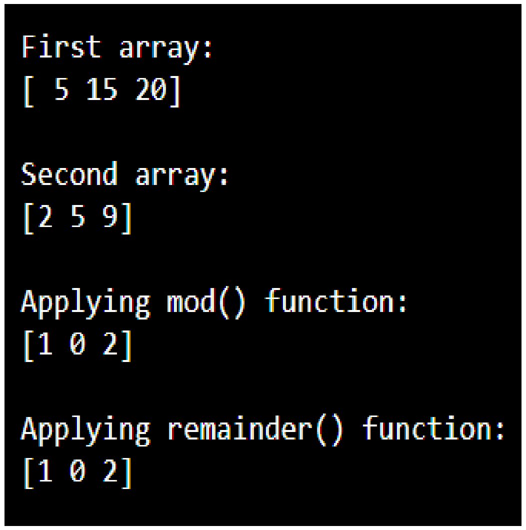
```
import numpy as np

arr = np.array([5, 10, 15])

print('First array is:’)
print(arr)

print('\nApplying power function:’)
print(np.power(arr, 2))

print('\nSecond array is:’)
arr1 = np.array([1, 2, 3])
print(arr1)

print('\nApplying power function again:’)
print(np.power(arr, arr1))
```



```
First array is:
[ 5 10 15]

Applying power function:
[ 25 100 225]

Second array is:
[1 2 3]

Applying power function again:
[    5   100 3375]
```

Example: mod/reminder Operations

```
import numpy as np

arr = np.array([5, 15, 20])
arr1 = np.array([2, 5, 9])

print('First array:’)
print(arr)

print('\nSecond array:’)
print(arr1)

print('\nApplying mod() function:’)
print(np.mod(arr, arr1))

print('\nApplying remainder() function:’)
print(np.remainder(arr, arr1))
```



```
First array:
[ 5 15 20]

Second array:
[2 5 9]

Applying mod() function:
[1 0 2]

Applying remainder() function:
[1 0 2]
```

```
import numpy as np
arr = np.array([10, 20, 30, 40, 50])
print("Original Array:", arr)

# Aggregation functions
print("1. Sum:", np.sum(arr))
print("2. Mean (Average):", np.mean(arr))
print("3. Median:", np.median(arr))
print("4. Standard Deviation:", np.std(arr))
print("5. Variance:", np.var(arr))
print("6. Minimum Value:", np.min(arr))
print("7. Maximum Value:", np.max(arr))
print("8. Index of Minimum Value:", np.argmin(arr))
print("9. Index of Maximum Value:", np.argmax(arr))
print("10. Cumulative Sum:", np.cumsum(arr))
print("11. Cumulative Product:", np.cumprod(arr))
print("12. 25th Percentile:", np.percentile(arr, 25))
print("13. 50th Percentile (Median):", np.percentile(arr, 50))
print("14. 75th Percentile:", np.percentile(arr, 75))
```

NumPy’s Structured Array

- Numpy’s Structured Array is similar to the Struct in C. It is used for grouping data of different data types and sizes.
- Structured array uses data containers called fields. Each data field can contain data of any data type and size.
- Array elements can be accessed with the help of dot notation.
- To create a structured array in NumPy, we need to define a dtype (data type) that specifies the names and types of each field.

Example :

```
import numpy as np
dt = np.dtype([('name', 'U20'), ('age’, np.int32),
                              ('grade', np.float64)])
```

- In this example, we defined a dtype with three fields: 'name' as a Unicode string of length 20 characters, 'age' as a 32-bit integer, and 'grade' as a 64-bit floating-point number.

#create a structured array using dtype

data = np.array([('Alice', 25, 4.8), ('Bob', 23, 3.9), ('Charlie', 27, 4.5)], dtype=dt)

Example:

```
import numpy as np
dt = np.dtype([('name', 'U20'), ('age’, np.int32), ('grade', np.float64)], dtype=dt)
a = np.array([('Sana', 2, 21.0), ('Mansi', 7, 29.0)])
# Sorting according to the name
b = np.sort(a, order='name’)
print('Sorting according to the name', b)
# Sorting according to the age
b = np.sort(a, order='age’)
print('\nSorting according to the age', b)
```

```
# New record
new_record = np.array([('Anjali', 5, 24.5)], dtype=dt)
# Concatenate the new record to the existing array
a = np.concatenate((a, new_record))
print("After adding new record:\n", a)
# New dtype with an added column 'status'
new_dt = np.dtype([('name', 'U20'), ('age', np.int32), ('grade',
np.float64), ('status', 'U10')])
# Create new structured array with default status 'Pass'
new_array = np.empty(a.shape, dtype=new_dt)
# Copy old data
new_array['name'] = a['name']
new_array['age'] = a['age']
new_array['grade'] = a['grade']
new_array['status'] = 'Pass'  # default value
print("After adding new column:\n", new_array)
```

```
# remove record
filtered = a[a['name'] != 'Sana']
print("After removing record:\n", filtered)



# New dtype without 'grade'
reduced_dt = np.dtype([('name', 'U20'), ('age', np.int32)])

# Create new array and copy only the required fields
reduced_array = np.empty(a.shape, dtype=reduced_dt)
reduced_array['name'] = a['name']
reduced_array['age'] = a['age']

print("After removing 'grade' column:\n", reduced_array)
```

```r
# Correlation matrix
df <- data.frame(
  Age = c(25, 30, 28, 35, 40),
  Salary = c(50000, 60000, 55000, 75000, 80000),
  Experience = c(2, 5, 4, 7, 10)
)

# Compute correlation matrix
cor_matrix <- cor(df)
print(cor_matrix)
```

```
# Create a 5x5 numeric matrix
data_matrix <- matrix(c(10, 20, 30, 40, 50,
                        15, 25, 35, 45, 55,
                        5, 10, 15, 20, 25,
                        12, 22, 32, 42, 52,
                        8, 18, 28, 38, 48),
                      nrow = 5, ncol = 5, byrow = TRUE)

# Assign correct row and column names
rownames(data_matrix) <- c("A", "B", "C", "D", "E")
colnames(data_matrix) <- c("X1", "X2", "X3", "X4", "X5")
heatmap(data_matrix, col = heat.colors(10), main = "Basic Heatmap in
R")
```

Example 1:

Create the following bank data frame and perform the below operations.
Bank_data <- data.frame(Customer_ID = 1:10, Age = c(25, 34, 45, 29, 38, 50, 62, 28, 41, 33),Balance = c(5000, 12000, 8000, 4000, 15000, 2000, 30000, 4500, 11000, 7500), Loan_Status = c("No", "Yes", "Yes", "No", "No", "Yes", "Yes", "No", "Yes", "No"),Credit_Score = c(750, 680, 720, 800, NA, 650, 600, NA, 700, 730))

a) If Credit_Score is NA, replace it with the average Credit_Score.
b) Select customers with balance more than 10,000 and credit score below 700.
c) Add a column Risk_Level based on Credit_Score: If Credit_Score ≥ 750, set as "Low Risk". If Credit_Score between 650 and 749, set as "Medium Risk". If Credit_Score < 650, set as "High Risk".
d) Find the average balance for customers grouped by Loan_Status
e) Sort customers by Balance in descending order where Loan_Status as No.
f) Add 'Remarks' column: 'Excellent' if Score > 90, 'Good' if 75 <= Score <= 90, else 'Needs Improvement'

Example 2:

Create a 3D NumPy array of shape (3,4,5) with random floating-point values.

a) Compute the mean along axis 0.
b) Extract the first two layers of the 3D array.
c) Reshape it into a 2D array with shape (6,10).
d) Find the cumulative product of all elements.
e) Replace all values less than 0.5 with 0.

Example 3:

Data frame as follows Student<- data.frame(

Name=c(rohit shaw', 'ankita raj’, 'tallapally virat', 'tarun singh'),

Age=c(24, np.nan, 22, 32, None),

Score=c(88, 95, 70, np.nan, 992),

City=c('New York', None, 'Chicago', 'Houston', 'Phoenix'))

a) Fill missing values with mean for all columns in place of NaN.
b) Add Exam_Result column based on the Score column if Score > 75 ‘Pass’ else ‘Fail’.
c) Convert names, City names to uppercase format.
d) Add a new column 'Age Group' based on Age ‘Young' if Age < 25 else 'Adult'.
e) Identify outliers and remove the outliers in Score field.
f) Arrange Score in descending, Age in ascending, and NA values at the end of the output.

Example 4:

Create a structured NumPy array for a company's sales data with attributes: Sales_ID (integer), Product_Name (string, max length 12), Units_Sold (integer), and Revenue (float).

a) Add a new sales record to the array.
b) Retrieve all sales records where Units_Sold is greater than 50.
c) Sort the array in descending order based on Revenue.
d) Compute the total revenue(sum of sales data) for all products.
e) Extract only the Product_Name and Revenue columns.