MODULE -4

Introduction to Arrays

Introduction to Arrays

- An **array** is a collection of elements **of the same data type**, stored in **contiguous memory locations**.
- Arrays help store **multiple values under a single name** and allow fast access using **index numbers**.
- They are used when **large amounts of data** of the same type need to be stored and manipulated efficiently

Example:

marks = [78, 65, 90, 88, 72]

Characteristics of Arrays

- Stores multiple values in a single variable.
- All elements are of the **same data type**.
- Elements are stored in **continuous memory** locations.
- Each element is accessed using an **index number** (starting from 0).
- Size of the array is **fixed** once declared.

Syntax

import array

or

import array as arr

Here,

- array is the **module name**, and
- arr is a **short alias name** (optional).

Creating an Array

After importing the array module, we can create an array using this syntax:

array.array(typecode, [elements])

Parameters:

typecode → represents the type of elements in the array

[elements] → list of values to be stored in the array

Importing and Creating an Array

Example

```
import array as arr  #Imports with alias name
numbers = arr.array('i', [10, 20, 30, 40]) #Creates integer array
print(numbers)
```

Output:

```
array('i', [10, 20, 30, 40])
```

Importance of arrays in numerical computing

- **Store Large Data Efficiently**
  Arrays can hold hundreds or thousands of numeric values in a single structure, allowing efficient handling of big datasets.
- **Fast Data Access**
  Since array elements are stored in contiguous memory, accessing or modifying any element using an index is very fast.
- **Simplifies Mathematical Operations**
  Arrays make it easy to perform operations such as addition, subtraction, multiplication, and division on large sets of data.

- **Support for Multi-dimensional Data**
  Arrays can represent matrices and higher-dimensional data easily, which are essential in numerical computing fields like:

Linear Algebra

Machine Learning

Scientific Simulations

- **Integration with Numerical Libraries**
  Python libraries like **NumPy and SciPy**, are built around arrays for high-performance numerical operations.
- **Memory Efficiency**
  Arrays use less memory compared to other data structures like lists, since they store elements of the same data type compactly.

Difference between Python lists and NumPy arrays

Feature	Python List	NumPy Array
Definition	Built-in data structure in Python	Data structure provided by NumPy library
Data Type	Can store different data types	Stores only one data type (homogeneous)
Memory Usage	High memory consumption	Low memory usage
Computation Speed	Slower	Faster due to C-based implementation
Mathematical Operations	Performed element by element using loops	Performed on entire array at once
Indexing	Supports simple indexing	Supports multi-dimensional indexing
Performance	Not suitable for large data	Suitable for large-scale numerical data
Library Requirement	No need to import any library	Requires NumPy to be imported
Example	`[1, "a", 3.5]`	`np.array([1, 2, 3])`

Setting up NumPy

Steps in Setting Up NumPy

Setting up NumPy involves three main steps:

- **Installing NumPy**
- **Importing NumPy into the program**
- **Creating and using NumPy arrays**

1. Installing NumPy

Before using NumPy, it must be installed on your system.

To install NumPy, use the **Python Package Installer (pip)**.

Command:

pip install numpy

pip – Python's package management tool.

install numpy – downloads and installs the NumPy package from the Python Package Index (PyPI).

Once installed, NumPy is ready to use in Python programs.

2)Importing NumPy

- Once NumPy is installed, it must be **imported** into your Python program before use.

Syntax:

```
import numpy as np
```

- import numpy → loads the NumPy library.
- as np → gives NumPy a **short alias name** for convenience.
- This alias np is commonly used by developers to make the code cleaner and shorter.

3)Creating NumPy Arrays

- NumPy's core feature is the **ndarray (n-dimensional array)**, which is used to store elements of the same data type.

Example:

```
import numpy as np
 arr = np.array([10, 20, 30, 40])
print(arr)
```

Output:

```
[10 20 30 40]
```

- np.array() creates an array.
- Arrays can be **1D**, **2D**, or **multi-dimensional**.

Indexing and slicing arrays

- **Indexing** means accessing **individual elements** or **specific positions** in an array using their **index numbers**.
- Like lists, NumPy arrays also use **zero-based indexing**, i.e., the **first element** has index **0**, the **second** has index **1**, and so on.

Syntax:

array_name[index]

One-Dimensional Indexing

```
import numpy as np
arr = np.array([10, 20, 30, 40, 50])
print(arr[0]) # Access first element
print(arr[3]) # Access fourth element
```

Output:

10

40

Negative Indexing

- Negative indexing starts counting from the **end** of the array.

```
print(arr[-1]) # Last element
print(arr[-2]) # Second last element
```

Output:

50

40

Indexing in Multi-Dimensional Arrays

- In a **2D array**, elements are accessed using **row and column indexes**.

Syntax:

```
array_name[row_index, column_index]
```

Example – 2D Array Indexing

```
import numpy as np
arr = np.array([[10, 20, 30],
                [40, 50, 60],
                [70, 80, 90]])
print(arr[0, 0]) # Element at 1st row, 1st column
print(arr[1, 2]) # Element at 2nd row, 3rd column
print(arr[2, -1]) # Last column of last row
```

Output:

```
10
60
90
```

Slicing in NumPy Arrays

- Slicing extracts a range of elements.

array[start:stop:step]

- **start** → starting index (inclusive)
- **stop** → ending index (exclusive)
- **step** → increment (default is 1)

Slicing in 1D Array Example

```
arr = np.array([10, 20, 30, 40, 50, 60, 70])
print(arr[1:5])       # Elements from index 1 to 4
print(arr[:4])        # From start to index 3
print(arr[3:])        # From index 3 to end
print(arr[::2])        # Every second element
```

Output:

```
[20 30 40 50]
[10 20 30 40]
[40 50 60 70]
[10 30 50 70]
```

Array Arithmetic & Mathematical Operations

Array arithmetic means performing **mathematical operations directly on entire arrays** rather than looping through elements.

1) Element-wise Operations

Performed between arrays of the **same shape**.

- Addition (+)
- Subtraction (-)
- Multiplication (*)
- Division (/)
- Exponentiation (**)

Example – Basic Array Arithmetic

```
import numpy as np
a = np.array([10, 20, 30, 40])
b = np.array([1, 2, 3, 4])
print(a + b)   # Addition
print(a - b)   # Subtraction
print(a * b)   # Multiplication
print(a / b)   # Division
```

Output:

```
[11 22 33 44]
[ 9 18 27 36]
[10 40 90 160]
[10. 10. 10. 10.]
```

Array and Scalar Operations

- Operations between an **array** and a **scalar (single number)** are also performed element-wise.

If a is an array and k is a scalar:

- a + k → adds k to every element
- a - k → subtracts k from every element
- a * k → multiplies every element
- a / k → divides every element
- a ** k → raises every element to power k

```
arr = np.array([5, 10, 15, 20])
print(arr + 5)
print(arr * 2)
```

Output:

```
 [10 15 20 25]
[10 20 30 40]
```

```
import numpy as np
# Create an array
a = np.array([5, 10, 15, 20])
print("Original Array:", a)
print("Add Scalar (a + 5):", a + 5)
print("Subtract Scalar (a - 3):", a - 3)
print("Multiply Scalar (a * 2):", a * 2)
print("Divide Scalar (a / 5):", a / 5)
print("Power Scalar (a ** 2):", a ** 2)
```

Output

```
Original Array: [ 5 10 15 20]
Add Scalar (a + 5): [10 15 20 25]
Subtract Scalar (a - 3): [ 2  7 12 17]
Multiply Scalar (a * 2): [10 20 30 40]
Divide Scalar (a / 5): [1. 2. 3. 4.]
Power Scalar (a ** 2): [ 25 100 225 400]
```

Common Mathematical Functions

Function	Description	Example
`np.add(x, y)`	Adds elements	`np.add(a, b)`
`np.subtract(x, y)`	Subtracts elements	`np.subtract(a, b)`
`np.multiply(x, y)`	Multiplies elements	`np.multiply(a, b)`
`np.divide(x, y)`	Divides elements	`np.divide(a, b)`
`np.sqrt(x)`	Square root	`np.sqrt(a)`
`np.power(x, y)`	Power function	`np.power(a, 2)`
`np.exp(x)`	Exponential (e^x)	`np.exp(a)`
`np.log(x)`	Natural logarithm	`np.log(a)`
`np.sin(x)`	Sine of elements	`np.sin(a)`
`np.cos(x)`	Cosine of elements	`np.cos(a)`
`np.tan(x)`	Tangent of elements	`np.tan(a)`

Mathematical Operations

Function	Description
`np.sum()`	Sum of all elements
`np.mean()`	Average of elements
`np.median()`	Median value
`np.std()`	Standard deviation
`np.var()`	Variance
`np.min()`	Minimum value
`np.max()`	Maximum value

Example program

```
import numpy as np
a = np.array([5, 10, 15, 20, 25])
print("Array:", a)
print("Sum:", np.sum(a))
print("Mean:", np.mean(a))
print("Median:", np.median(a))
print("Maximum:", np.max(a))
print("Minimum:", np.min(a))
```

Output

```
Array: [ 5 10 15 20 25]
Sum: 75
Mean: 15.0
Median: 15.0
Maximum: 25
Minimum: 5
```

Reshaping and Transposing arrays

Reshaping an array means **changing its dimensions** (number of rows and columns) while **keeping the total number of elements the same**.
It is done using the **reshape()** function in NumPy.

Syntax:

array.reshape(new_rows, new_columns)

Example 1 – Reshaping a 1D Array to 2D

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

Output:

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

Explanation:

The original array had 6 elements → reshaped into **2 rows × 3 columns**.

Reshaping 2D to 1D

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

Output:

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

Explanation:

The 2D array is **flattened** into a single-dimensional array.

Transposing Arrays

- Transposing **flips rows and columns** (matrix rotation over its diagonal).

Syntax:

array.T

Example:

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

Output:

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

Stacking and Splitting arrays

1. Stacking Arrays

- Stacking means **combining multiple arrays** into one.

a) Vertical Stacking (vstack)

- Stacks arrays **row-wise** (one on top of another).

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

Output:

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

b)Horizontal Stacking (hstack)

- Stacks arrays **column-wise** (side by side).

```
c = np.array([[1, 2], [3, 4]])
d = np.array([[5, 6], [7, 8]])
result = np.hstack((c, d))
print(result)
# Output:
# [[1 2 5 6]
#  [3 4 7 8]]
```

c)Depth Stacking (dstack)

Stacks arrays along a **third dimension**.

```
e = np.array([[1, 2], [3, 4]])
f = np.array([[5, 6], [7, 8]])
result = np.dstack((e, f))
print(result)
```

Output:

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

Splitting Arrays

- Splitting means **dividing one array into multiple arrays**.

a) Horizontal Split (hsplit)

Splits array **column-wise**.

```
g = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
result = np.hsplit(g, 2)
print(result)
```

Output:

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

b) Vertical Split (vsplit)

np.vsplit() is used to **split an array row-wise**, meaning it divides the array into multiple sub-arrays **horizontally** (based on rows).

```
import numpy as np

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

result = np.vsplit(h, 3)
print(result)
```

Output

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

c) Depth Split (dsplit)

np.dsplit() splits an array **along the 3rd dimension (depth / z-axis)** of a 3-D array.

```
import numpy as np

i = np.array([
            [[1, 2],
             [3, 4]],

            [[5, 6],
             [7, 8]]
            ])

result = np.dsplit(i, 2)
print(result)
```

Output

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

         [[5],
          [7]]]),

  array([[[2],
          [4]],

         [[6],
          [8]]])
]
```

Copying Arrays- Different types of Copying technique

- When working with **NumPy arrays**, sometimes you **copy** data, and sometimes you only **reference** it.
- The difference affects **memory usage and changes** in arrays.

Types of Copying Techniques

A) Assignment (= operator) — No Copy

- Only creates a **reference** to the same data.
- Changing one array **affects the other**.

```
import numpy as np
a = np.array([1, 2, 3])
b = a
b[0] = 99
print(a)
```

Output: [99 2 3]

B) Shallow Copy (view())

- Creates a **new array object** but shares the **same data**.
- Changing data in one changes the other.
- Shape changes in the new array **do not affect** the original, but element changes do.

```
c = np.array([1, 2, 3])
d = c.view()
d[0] = 88
print(c)
```

Output: [88 2 3]

C)Deep Copy (copy())

- Creates a **completely independent array** with separate memory.
- Changing one array **does not affect** the other.

```
e = np.array([1, 2, 3])
f = e.copy()
f[0] = 77
print(e)
```

Output: [1 2 3]