NUMPY

What is NumPy?

NumPy, short for Numerical Python, is an open-source Python library. It supports multi-dimensional arrays (matrices) and provides a wide range of mathematical functions for array operations. It is used in scientific computing, and in areas like data analysis, machine learning, etc.

Why to Use NumPy?

The following are some of the key reasons to use NumPy:

- NumPy provides various math functions for calculations like addition, algebra, and data analysis.
- NumPy provides various objects representing arrays and multi-dimensional arrays which can be used to handle large data such as images, sounds, etc.
- NumPy also works with other libraries like SciPy (for scientific computing), Pandas (for data analysis), and scikit-learn (for machine learning).
- NumPy is fast and reliable, which makes it a great choice for numerical computing in Python.

NumPy Applications

The following are some common application areas where NumPy is extensively used:

- Data Analysis: In Data analysis, while handling data, we can create data (in the form of array objects), filter the data, and perform various operations such as mean, finding the standard deviations, etc.
- Machine Learning & AI: Popular machine learning tools like TensorFlow and PyTorch use NumPy to manage input data, handle model parameters, and process the output values.
- Array Manipulation: NumPy allows you to create, resize, slice, index, stack, split, and combine arrays.
- Finance & Economics: NumPy is used for financial analysis, including portfolio optimization, risk assessment, time series analysis, and statistical modelling.
- Image & Signal Processing: NumPy helps process and analyze images and signals for various applications.
- Data Visualization: NumPy independently does not create visualizations, but it works with libraries like Matplotlib and Seaborn to generate charts and graphs from numerical data.

NumPy Example

- The following is an example of Python NumPy:

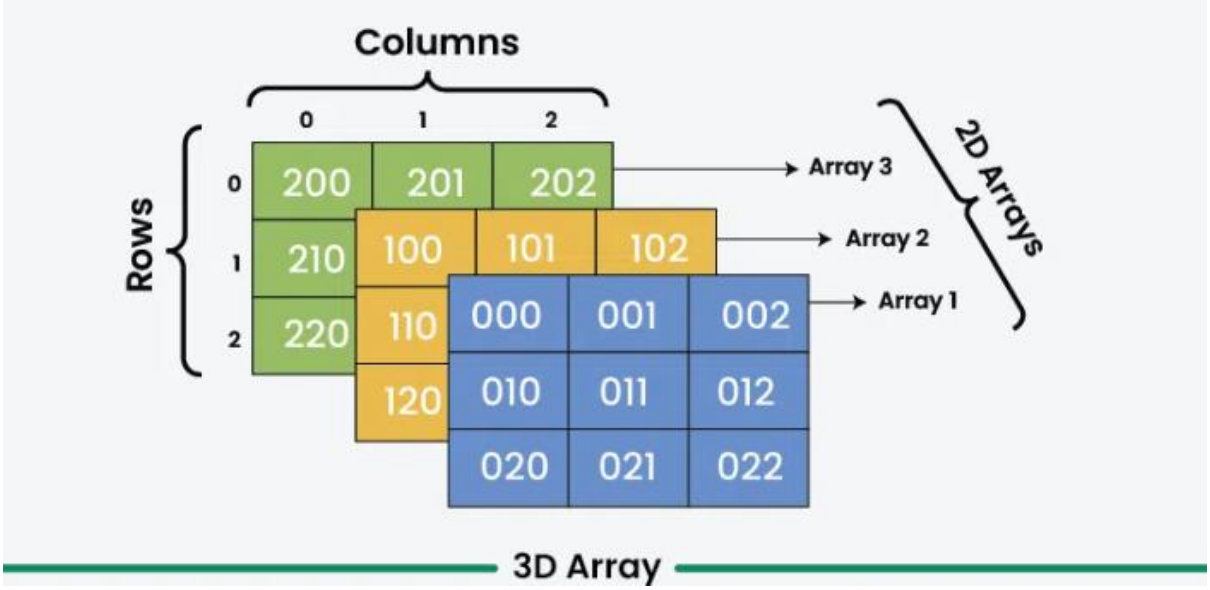
```
# Importing NumPy Array
import numpy as np
# Creating an array using np.array() method
arr = np.array([10, 20, 30, 40, 50])
# Printing
print(arr) # Prints [10 20 30 40 50]
```

What is an “array”?

- In computer programming, an array is a structure for storing and retrieving data. We often talk about an array as if it were a grid in space, with each cell storing one element of the data. For instance, if each element of the data were a number, we might visualize a “one-dimensional” array like a list:

- A two-dimensional array would be like a table:

- A three-dimensional array would be like a set of tables, perhaps stacked as though they were printed on separate pages. In NumPy, this idea is generalized to an arbitrary number of dimensions, and so the fundamental array class is called `ndarray`: it represents an “N-dimensional array”.



Most NumPy arrays have some restrictions. For instance:

- All elements of the array must be of the same type of data.
- Once created, the total size of the array can't change.
- The shape must be "rectangular", not "jagged"; e.g., each row of a two-dimensional array must have the same number of columns.

When these conditions are met, NumPy exploits these characteristics to make the array faster, more memory efficient, and more convenient to use than less restrictive data structures.

Feature	Python List	NumPy Array
Data Type	Mixed allowed	Single type
Speed	Slower	Faster
Memory	More	Less
Operations	Loops needed	Vectorized

Lists vs NumPy Arrays

```
Import numpy as np

# Python list
my_list = [1, 2, 3, 4]

# NumPy array
my_array = np.array([1, 2, 3, 4])

print("List:", my_list)
print("Array:", my_array)
```

Creating Arrays

a) From Python lists

```
a = np.array([10, 20, 30])
```

b) Using arange

- Like Python range(), returns evenly spaced numbers.

```
b = np.arange(0, 10)  # 0 to 9
```

c) Using linspace

- Returns n numbers evenly spaced between start and end.

```
c = np.linspace(0, 1, 5)  # 0, 0.25, 0.5, 0.75, 1
```

Key Point: Arrays can be 1D (row), 2D (table), or higher dimensions.

d) Special arrays

```
np.zeros((2,3))  # 2x3 array of zeros
np.ones((3,2))   # 3x2 array of ones
np.eye(3)        # 3x3 identity matrix
np.full((2,2), 7)  # 2x2 array filled with 7
```

1️⃣ np.zeros(shape) – Array of zeros

- What it is: Creates an array filled entirely with 0s.
- Shape: (rows, columns) → size of the array.

- Why it's used:

    - Initialize a blank array to store results.
    - Useful in preallocating memory before calculations.

Example:

```
import numpy as np

arr = np.zeros((2,3))  # 2 rows, 3 columns
print(arr)
```

✅ Notice: All elements are 0.0 (default float).

Output:

```
[[0. 0. 0.]
 [0. 0. 0.]]
```

2️⃣ np.ones(shape) – Array of ones

- What it is: Creates an array filled entirely with 1s.
- Why it's used:

    - Useful when you need a baseline array to multiply or add later.
    - Often used in weights initialization in machine learning or formulas.

Example:

```
arr = np.ones((3,2))  # 3 rows, 2 columns
print(arr)
```

Output:

```
[[1. 1.]
 [1. 1.]
 [1. 1.]]
```

3️⃣ np.eye(n) – Identity matrix

- What it is: Creates a square matrix (n×n) with 1s on the diagonal and 0s elsewhere.
- Why it's used:
    - Common in linear algebra (matrix multiplication, transformations).
    - Acts like a "do nothing" matrix in multiplication.

Example:

```
arr = np.eye(3)  # 3x3 identity matrix
print(arr)
```

Output:

```
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

✅ Diagonal elements = 1, others = 0.

4️⃣ np.full(shape, fill_value) – Array filled with a specific value

- What it is: Creates an array of given shape filled with any number you choose.

- Why it's used:

    - Initialize arrays with default values other than 0 or 1.
    - Useful in simulations or preparing a constant array.

Example:

```
arr = np.full((2,2), 7)  # 2x2 array filled with 7
print(arr)
```

Output:

```
[[7 7]
 [7 7]]
```

Array Properties

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

print("Shape:", arr.shape)  # (2, 3) → 2 rows, 3 columns
print("Size:", arr.size)    # 6 → total elements
print("Data type:", arr.dtype)  # int64
```

2D Array (Table Example)

```
marks = np.array([
   [85, 90, 78],   # Student 1
   [88, 76, 92],   # Student 2
   [90, 91, 85]    # Student 3
])
print(marks[0])      # Row 0 → [85 90 78]
print(marks[:,0])    # Column 0 → [85 88 90]
print(marks[1,2])    # Row 1, Column 2 → 92
```

Tip: [row, column] → rows = students, columns = subjects.

Indexing 1D & 2D Arrays

1D Array

arr = np.array([10, 20, 30, 40])

print(arr[0]) # 10 → first element

print(arr[-1]) # 40 → last element

Slicing & Fancy Indexing

Slicing

arr = np.array([0,1,2,3,4,5])

print(arr[1:4]) # [1 2 3] → from index 1 to 3

print(arr[:3])   # [0 1 2] → first 3 elements

print(arr[3:])   # [3 4 5] → from index 3 to end

2D Slicing

matrix = np.arange(12).reshape(3,4)

print(matrix[:,1:3]) # all rows, columns 1 & 2

Fancy Indexing

arr = np.array([10,20,30,40,50])

print(arr[[0,2,4]]) # pick 0th, 2nd, 4th elements

Boolean Indexing

arr = np.array([10,15,20,25,30])

print(arr[arr > 20]) # [25 30] → only values >20

Reshaping & Flattening

Reshape

```
nums = np.arange(12)

table = nums.reshape(3,4)

print(table)
```

Flatten / Ravel

```
flat = table.flatten()

print(flat)  # [0 1 2 3 4 5 6 7 8 9 10 11]
```

Why useful?

- Reshape → make data ready for calculations
- Flatten → convert multi-dimensional to 1D

Output:

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

Arithmetic Operations & Broadcasting

Element-wise math

```
scores = np.array([50, 60, 70, 80])

print(scores + 10)  # [60 70 80 90]

print(scores * 2)   # [100 120 140 160]
```

Broadcasting (1D → 2D)

Broadcasting means automatically expanding smaller arrays to match the shape of larger ones during arithmetic operations.

Instead of manually repeating values, NumPy "broadcasts" them across the larger array.

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

add_row = np.array([10,20,30])

print(matrix + add_row)
```

Output:

```
[[11 22 33]

 [14 25 36]]
```

✅ No loops needed. Saves time and makes code clean.

Aggregate Functions

```
sales = np.array([1000, 1200, 900, 1500, 2000])


print("Sum:", sales.sum())      # Total sales

print("Mean:", sales.mean())    # Average

print("Max:", sales.max())      # Highest sale

print("Min:", sales.min())      # Lowest sale

print("Std Dev:", sales.std())  # Variation in sales
```

Axis in 2D

In NumPy, an axis is the direction along which an operation is performed on an array.

- **axis = 0 → operations are performed down the rows (column-wise).**
- **axis = 1 → operations are performed across the columns (row-wise).**

```
marks = np.array([[85,90,78],[88,76,92],[90,91,85]])
print(marks.mean(axis=0))  # Column-wise average
print(marks.mean(axis=1))  # Row-wise average
```

Random Numbers (Bonus tip)

Useful to simulate datasets or for practice.

Random integers

```
marks = np.random.randint(0,101,10)  # 10 random marks 0-100
print(marks)
```

Exercises (Hands-on)

1. Create a 5×5 identity matrix

2. Generate random marks for 10 students in 3 subjects, find average per student

3. Slice [10,20,30,40,50] → pick elements 2nd to 4th

4. Reshape numbers 0–11 into 3×4 table and flatten back

5. Add 10 marks grace to [50,60,70,80] using broadcasting