NumPy Revision

What is NumPy?

- NumPy stands for Numerical Python
- It is the fundamental library for scientific computing in Python
- Created in 2005 by Travis Oliphant

What Does NumPy Do?

- Provides a multi-dimensional array object
- Supports fast mathematical operations
- Performs calculations much faster than Python lists

Main Uses of NumPy:

NumPy provides functions for:

- Linear Algebra
- Matrix operations
- Fourier Transformations
- Scientific and mathematical computing

Features:

- Provides powerful n-dimensional array (ndarray)
- Faster than normal Python lists
- Uses less memory
- Supports vectorized operations (no loops needed)
- Has built-in math & statistical functions
- Supports matrix and linear algebra operations
- Allows reshaping and dimension change of arrays
- Provides random number generation
- Base library for Data Science, ML & AI

NumPy Array:

- Array is the core data structure of NumPy
- Called ndarray (N-Dimensional Array)
- Stores data in rows & columns
- All elements must be of the same data type
- Faster and more memory-efficient than Python lists

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

Types of NumPy Arrays:

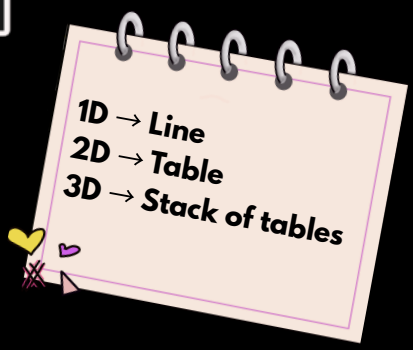
1. 1D Array (One-Dimensional)

- Looks like a single row of values
- Used to store one line of data

```
a = np.array([10, 20, 30, 40])
print(a)
```

2. 2D Array (Two-Dimensional)

- Looks like a table (rows and columns)
- Used to store matrix-like data
- Very useful for marks, tables, grids, etc.




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

3. 3D Array (Three-Dimensional)

- Collection of multiple 2D arrays stacked together
- Used in images, videos, deep learning, etc.

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

Creation of Arrays in NumPy

1. np.array()

Creates array from list

```
import numpy as np

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

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

2. np.zeros()

Creates array filled with 0

```
arr = np.zeros(3)
print(arr)
```

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

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

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

3. np.ones()

Creates array filled with 1

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

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

4. np.full()

Creates array filled with same value

```
arr = np.full((2,2),7)
print(arr)
```

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

5. np.arange()

- Creates sequence of numbers
- Format → arange(start, stop, step)

```
arr = np.arange(1,10,2)
print(arr)
```

```
[1 3 5 7 9]
```

6. np.eye()

- Creates identity matrix
- Diagonal elements are 1, others are 0

```
arr = np.eye(3)
print(arr)
```

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

NumPy Array Properties

1. shape → Structure of Array (Rows, Columns)

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

```
(2, 3)
```

These are the checking properties

2. size → Total Number of Elements

```
arr = np.array([[10,20,30],[40,50,60]])
print(arr.size)
```

```
6
```

3. ndim → Number of Dimensions

```
arr1 = np.array([1,2,3])
arr2 = np.array([[1,2,3],[4,5,6]])
arr3 = np.array([[[1,2],[3,4],[5,6],[7,8]]])

print(arr1.ndim)
print(arr2.ndim)
print(arr3.ndim)
```

```
1
2
3
```

4. dtype → Data Type of Elements

```
arr = np.array([10,20,30.5,40])
print(arr.dtype)
```

```
float64
```

5. astype() → change the data type of an array

```
arr = np.array([1.2, 2.5, 3.8])
int_arr = arr.astype(int)
print(int_arr)
```

```
[1 2 3]
```

Operators

- NumPy operators perform element-wise operations
- Same operation is applied to every element of the array
- No loops are required (vectorized working)

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

print(arr + 5)
```

```
[15 25 35]
```

```
print(arr * 10)
```

```
[100 200 300]
```

```
print(arr / 10 )
```

```
[1. 2. 3.]
```

Aggregation Functions

These functions summarize array data into a single value.

Key Points

- Used to analyze data quickly
- Convert large data into single summary values

```
import numpy as np
arr = np.array([10, 20, 30, 40, 50])
print("Sum =", np.sum(arr))
print("Mean =", np.mean(arr))
```

```
Sum = 150
Mean = 30.0
```

Function	Use	Code	Output
np.sum()	Adds all values	np.sum(arr)	150
np.mean()	Finds average	np.mean(arr)	30
np.min()	Finds minimum	np.min(arr)	10
np.max()	Finds maximum	np.max(arr)	50
np.std()	Standard deviation	np.std(arr)	14.14
np.var()	Variance	np.var(arr)	200

Indexing & Slicing

Given Array:

```
import numpy as np
arr = np.array([10,20,30,40,50,60])
```

1. Indexing

- Used to access single element of array
- Indexing starts from 0
- Negative indexing accesses elements from end

```
print(arr[0])     # first element
print(arr[-1])    # last element
```

```
10
60
```

2. Slicing

- Used to access multiple elements at once
- Format: array[start : stop : step]
- Stop value is excluded
- Step controls jumping of elements
- [::-1] is used to reverse array

```
print(arr[1:5])    # index 1 to 4
print(arr[:4])     # start to index 3
print(arr[::2])    # every 2nd element
print(arr[::-1])   # reverse array
```

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

3. Fancy Indexing

- Used to select specific positions at once
- Pass list of index numbers
- Returns a new array

```
print(arr[[0, 2, 4]])
```

```
[10 30 50]
```

4. Boolean Masking

- Used to filter elements based on conditions
- Returns elements that satisfy condition
- Very useful in Data Analysis

```
print(arr[arr > 25])
```

```
[30 40 50 60]
```

Reshaping & Manipulation

1. reshape() – Change Shape of Array

- Used to change rows and columns
- Total elements must be same
- Does not create copy (returns a view)

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

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

2. Flattening Array (Convert to 1D)

a. ravel()

- Returns a view
- No new memory created
- Faster & memory-efficient
- Changes may affect original array

b. flatten()

- Returns a copy
- New memory is created
- Slightly slower
- Changes do not affect original array

```
arr_2d = np.array([[1,2,3],[4,5,6]])
print(arr_2d.ravel())
print(arr_2d.flatten())
```

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

Array Modification

1. insert

- Used to add new element, row or column into an array.
- np.insert(array, index, value, axis=None)

Meaning of Axis in NumPy

• axis = 0 → Works row-wise (vertical direction)
• axis = 1 → Works column-wise (horizontal direction)
• axis = None → Flattens array first, then performs operation on entire array

```
arr = np.array([10,20,30])
new_arr = np.insert(arr,1,100)
print(new_arr)
```

```
[ 10 100  20  30]
```

Insert Row (2D, axis = 0)

```
arr = np.array([[1,2],[3,4]])
new_arr = np.insert(arr,1,[5,6],axis=0)
print(new_arr)
```

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

Insert Column (2D, axis = 1)

```
arr = np.array([[1,2],[3,4]])
new_arr = np.insert(arr,1,[9,9],axis=1)
print(new_arr)
```

```
[[1 9 2]
 [3 9 4]]
```

2. Append

- Adds elements at the end of array.
- np.append(array, values, axis=None)

```
arr = np.array([10,20,30])
new_arr = np.append(arr,[40,50])
print(new_arr)
```

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

Append Row (2D)

```
arr = np.array([[1,2],[3,4]])
new_arr = np.append(arr,[[5,6]],axis=0)
print(new_arr)
```

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

Append Column (2D)

```
arr = np.array([[1,2],[3,4]])
new_arr = np.append(arr,[[9],[9]],axis=1)
print(new_arr)
```

```
[[1 2 9]
 [3 4 9]]
```

3. Delete

- Used to remove elements, rows or columns.
- np.delete(array, index, axis=None)

```
arr = np.array([10,20,30])
new_arr = np.delete(arr,1)
print(new_arr)
```

```
[10 30]
```

Delete Row (2D)

```
arr = np.array([[1,2],[3,4]])
new_arr = np.delete(arr,0,axis=0)
print(new_arr)
```

```
[[3 4]]
```

Delete Column (2D)

```
arr = np.array([[1,2],[3,4]])
new_arr = np.delete(arr,1,axis=1)
print(new_arr)
```

```
[[1]
 [3]]
```

4. Concatenate

- Used to merge two or more arrays.
- np.concatenate((arr1, arr2), axis=0)

```
a = np.array([1,2])
b = np.array([3,4])
new_arr = np.concatenate((a,b))
print(new_arr)
```

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

Concatenate Row (2D)

```
a = np.array([[1,2]])
b = np.array([[3,4]])
print(np.concatenate((a,b),axis=0))
```

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

Concatenate Column (2D)

```
a = np.array([[1,2]])
b = np.array([[3,4]])
print(np.concatenate((a,b),axis=1))
```

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

5. Stack

- Stack arrays vertically or horizontally.
- **np.vstack() & np.hstack()**
- They are used to join arrays.

np.vstack() (Vertical Stack)

• Joins arrays row-wise (one below another)

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

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

np.hstack() (Horizontal Stack)

• Joins arrays column-wise (side by side)

```
a = np.array([1,2,3])
b = np.array([4,5,6])
print(np.hstack((a,b)))
```

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

6. Split

- Used to divide array into parts.
- returns a **LIST of NumPy arrays**, not a single array.

np.split()

• Splits an array into equal parts
• Works for both 1D and 2D arrays

```
arr = np.array([10,20,30,40,50,60])
new_arr = np.split(arr,2)
print(new_arr)
```

```
[array([10, 20, 30]), array([40, 50, 60])]
```

np.hsplit() – Horizontal Split

• Splits column-wise
• Used for 2D arrays

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

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

np.vsplit() – Vertical Split

• Splits row-wise
• Used for 2D arrays

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

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

NumPy Broadcasting

Broadcasting is a feature that allows NumPy to perform operations on arrays of different shapes without using loops.

Why Broadcasting?

- Removes need of for loops
- Makes code shorter & faster
- Automatically expands smaller array to match larger one

Example 1: Scalar with Array

```
arr = np.array([100,200,300])
print(arr * 2)
```

```
[200 400 600]
```

Example 2: 1D with 2D

```
matrix = np.array([[1,2,3],[4,5,6]])
vector = np.array([10,20,30])
print(matrix + vector)
```

```
[[11 22 33]
 [14 25 36]]
```

Broadcasting Rules

1. Matching Dimensions

When both arrays have the same shape, NumPy directly performs element-wise operation.

```
a = np.array([1,2,3])
b = np.array([5,7,9])
print(a + b)
```

```
[ 6  9 12]
```

2. Expanding Single Element

When a single value (scalar) is used, NumPy automatically expands it to match the array.

```
a = np.array([1,2,3])
print(a + 10)
```

```
[11 12 13]
```

3. Compatible Shapes

If one array has dimension 1, NumPy stretches it to match the larger array.

```
a = np.array([1,2,3])
b = np.array([1])
print(a + b)
```

```
[2 3 4]
```

4. Incompatible Shapes (Error)

If dimensions are not same and none is 1 → Error

```
a = np.array([1,2,3])
b = np.array([1,2])
print(a + b)
```

```
---------------------------
ValueError
```

NumPy Vectorization

Vectorization means performing operations on whole arrays at once instead of using loops.

Why Vectorization?

- Makes code short & clean
- Much faster than loops
- Uses optimized C-based operations

Example 1 – Add Two Arrays

```
arr1 = np.array([1,2,3])
arr2 = np.array([4,5,6])
result = arr1 + arr2
print(result)
```

```
[5 7 9]
```

Example 2 – Multiply Array by Scalar

```
arr = np.array([10,20,30])
multiplied = arr * 3
print(multiplied)
```

```
[30 60 90]
```

📌 Broadcasting vs Vectorization

Broadcasting	Vectorization
Works with **different** shaped arrays	Works with **same** shaped arrays
Expands **smaller** array automatically	Performs operations on whole arrays
Focus: **shape compatibility**	Focus: **speed & no loops**
Makes **shapes match**	Makes code fast & clean

Handling Missing Values in NumPy

1. np.isnan() – Detect Missing Values (NaN)

- Checks whether a value is NaN (Not a Number)
- Returns True where missing values exist
- Helps in finding empty / missing data
- Used before cleaning dataset
- Output is a boolean mask

```
arr = np.array([1,2,np.nan,4,np.nan,6])
print(np.isnan(arr))
```

```
[False False  True False  True False]
```

2. np.nan_to_num() – Replace Missing Values

- Converts NaN to numeric values
- Default replacement value = 0
- Can also replace +∞ and −∞
- Helps in preparing clean data for ML models
- Returns a new cleaned array

```
arr = np.array([1,2,np.nan,4,np.nan,6])
cleaned = np.nan_to_num(arr, nan=100)
print(cleaned)
```

```
[  1.   2. 100.   4. 100.   6.]
```

3. np.isinf() – Detect Infinite Values

- Detects infinite values (+∞, −∞)
- Returns True where infinity exists
- Used in financial & scientific calculations
- Prevents calculation errors

```
arr = np.array([1,2,np.inf,4,-np.inf,6])
print(np.isinf(arr))
```

```
[False False  True False  True False]
```

4. Replace Infinity Values

- Replaces positive infinity with custom value
- Replaces negative infinity with custom value
- Keeps dataset within valid numeric range
- Useful for data normalization

```
arr = np.array([1,2,np.inf,4,-np.inf,6])
cleaned = np.nan_to_num(arr, posinf=1000, neginf=-1000)
print(cleaned)
```

```
[    1.     2.  1000.     4. -1000.     6.]
```