NumPy

🔢 What is NumPy?

NumPy (Numerical Python) is a powerful Python library used for:

- Numerical and scientific computing
- Working with **multi-dimensional arrays**
- Performing mathematical operations like matrix multiplication, statistics, linear algebra, etc.

It is the foundation for many other Python libraries such as **Pandas**, **SciPy**, and **TensorFlow**.

💻 How to Install NumPy

You can install NumPy using pip:

```
pip install numpy
```

📚 What is a NumPy Array?

A **NumPy array** is like a list in Python, but more **powerful and efficient** for numerical operations. It is also called `ndarray` (n-dimensional array).

Example:

```python
import numpy as np

# Creating a numpy array

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

print(arr)
```

🧭 What is Dimension in NumPy?

In NumPy, the number of **dimensions (or axes)** of an array is called its **rank**.

📏 NumPy Array Dimensions Explained

Dimension	Description	Example
0D	Scalar (Single value)	np.array(5) → 5
1D	Vector (Single list)	np.array([1, 2, 3])
2D	Matrix (List of lists)	np.array([[1, 2], [3, 4]])
3D	Tensor (List of 2D arrays)	np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])

🧪 Examples of Different Dimensions

```
import numpy as np


# 0D Array (Scalar)

a = np.array(42)

print("0D:", a, "| Dimensions:", a.ndim)


# 1D Array

b = np.array([1, 2, 3])

print("1D:", b, "| Dimensions:", b.ndim)


# 2D Array

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

```python
print("2D:\n", c, "| Dimensions:", c.ndim)


# 3D Array
d = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
print("3D:\n", d, "| Dimensions:", d.ndim)
```

📘 NumPy Array Attributes

---

1. ndim

👉 Returns the number of dimensions (axes) of the array.
🧪 **Example:**

```python
arr = np.array([[1, 2, 3], [4, 5, 6]])
print(arr.ndim)  # Output: 2
```

---

2. shape

👉 Returns a tuple showing the size of the array in each dimension.
🧪 **Example:**

```python
print(arr.shape)  # Output: (2, 3)
```

---

3. size

👉 Returns the total number of elements in the array.
🧪 **Example:**

```python
print(arr.size)  # Output: 6 (2 rows × 3 columns)
```

---

4. dtype

👉 Returns the data type of the array elements.
🧪 **Example:**

```
print(arr.dtype)  # Output: int64 (depends on system)
```

🔹 astype()

- Used to **convert the data type** of an array.

Example:

```
float_arr = arr.astype(float)

print(float_arr)

print(float_arr.dtype)
```

Output:

[[1. 2. 3.]

[4. 5. 6.]]

float64

🛠️ NumPy Array Creation Routines

---

1. array()

👉 Creates an array from a Python list or tuple.
🧪 **Example:**

```
import numpy as np

a = np.array([1, 2, 3])

print(a)  # Output: [1 2 3]
```

---

2. zeros()

👉 Creates an array filled with zeros.
🧪 **Example:**

```
a = np.zeros((2, 3))

print(a)

# Output:

# [[0. 0. 0.]

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

---

3. ones()

👉 Creates an array filled with ones.
🧪 **Example:**

```
a = np.ones((2, 2))

print(a)

# Output:

# [[1. 1.]

#  [1. 1.]]
```

---

4. full()

👉 Creates an array filled with a specific value.
🧪 **Example:**

```
a = np.full((2, 3), 7)

print(a)

# Output:

# [[7 7 7]

#  [7 7 7]]
```

🧮 NumPy Math Operations on Arrays

Assume the array:

```
import numpy as np

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

🔹 Element-wise Operations:

```
print(arr + 5)
```

➤ Adds 5 to each element
Output:

```
[[15 25 35]
 [45 55 65]]
```

```
print(arr - 2)
```

➤ Subtracts 2 from each element
Output:

```
[[ 8 18 28]
 [38 48 58]]
```

```
print(arr * 5)
```

➢ Multiplies each element by 5

Output:

[[50 100 150]

[200 250 300]]

```
print(arr ** 5)
```

➢ Squares each element

Output:

[[100 400 900]

[1600 2500 3600]]

```
print(arr / 2)
```

➢ Divides each element by 2 (returns float values)

Output:

[[5. 10. 15.]

[20. 25. 30.]]

```
print(arr // 7)
```

➢ Floor division (integer division) by 7

Output:

[[1 2 4]

[5 7 8]]

Function	Description
np.sum()	Total of all elements
np.mean()	Average of all elements
np.min()	Minimum value
np.max()	Maximum value
np.std()	Standard deviation
np.var()	Variance (std deviation squared)

🎯 Indexing & Slicing in NumPy Arrays

---

1. Indexing (Access Elements)

👉 Use indexes to access individual elements.
🧪 **Example:**

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

print(a[2])  # Output: 30
```

---

2. 2D Indexing

👉 Access elements in 2D arrays using row and column.
🧪 **Example:**

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

print(b[1][0])  # Output: 3

print(b[1, 0])  # Output: 3 (same)
```

---

3. Slicing

👉 Get sub-arrays using the format [start:stop:step]
🧪 **Example:**

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

print(a[1:4])     # Output: [20 30 40]

print(a[::2])     # Output: [10 30 50]
```

---

4. 2D Slicing

👉 Slice rows and columns in 2D arrays.

array[row_start:row_end, column_start:column_end]

🧪 **Example:**

```
b = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

print(b[0:2, 1:])

# Output:

# [[2 3]

#  [5 6]]
```

🔹 Fancy Indexing in NumPy

Definition:
Fancy indexing allows you to access **multiple elements at once** using a list or array of indices.

🔸 Example:

```
import numpy as np

arr = np.array([10, 20, 30, 40, 50])
indices = [0, 2, 4]
result = arr[indices]
```

Output:

[10 30 50]

🟢 You can use fancy indexing for 2D arrays too:

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

result = arr2[[0, 2]]
```

Output:

[[1 2]
[5 6]]

🔹 Boolean Masking (Filtering)

Definition:
Boolean masking lets you filter elements **based on conditions**.

🔸 Example:

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

mask = arr > 25

result = arr[mask]
```

Output:

[30 40 50]

✅ Short form:

result = arr[arr > 25]

🔹 reshape() – Changing the Shape of an Array

Definition:
Used to **change the shape** (rows × columns) of an array **without changing data**.

🔸 Example:

```
import numpy as np

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

Output:

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

✅ You can also use -1 to let NumPy calculate the correct dimension:

```
arr.reshape(-1, 2)
```

🔹 Flattening Arrays

Used to **convert multi-dimensional arrays into a 1D array**.

🔸 flatten()

Returns a **copy** of the array as 1D.

```
arr2d = np.array([[1, 2, 3], [4, 5, 6]])
flat = arr2d.flatten()
```

Output:

[1 2 3 4 5 6]

🔸 ravel()

Returns a **view** of the array (no copy if not needed).

```
arr2d = np.array([[1, 2], [3, 4]])
flat_r = arr2d.ravel()
```

Output:

[1 2 3 4]

✅ Both work similarly, but:

Function	Returns	Editable
flatten()	Copy	No
ravel()	View	Yes (if possible)

📘 NumPy insert() – Notes

✅ Purpose:

To insert values into an existing NumPy array at a specified position.

🧠 Syntax:

```
np.insert(arr, index, value, axis=None)
```

🔸 **Parameters:**

Parameter	Description
arr	The original array
index	Position where value is inserted
value	Value(s) to insert
axis	Optional. If None, array is flattened. Use 0 for rows, 1 for columns (in 2D)

🔹 **Examples:**

👉 **Insert into 1D Array**

```
import numpy as np

arr = np.array([10, 20, 30])
new_arr = np.insert(arr, 1, 15)  # Insert 15 at index 1
# Output: [10 15 20 30]
```

👉 **Insert into 2D Array (Row-wise)**

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

👉 **Insert into 2D Array (Column-wise)**

```
arr = np.array([[1, 2], [3, 4]])
col = [9, 9]
new_arr = np.insert(arr, 1, col, axis=1)
# Output:
# [[1 9 2]
#  [3 9 4]]
```

⚠️ Note:

- If axis=None, the array is **flattened** first.
- np.insert() does **not** modify the original array; it returns a **new array**.

📘 NumPy append() and concatenate() – Notes

🔹 np.append()

✅ Purpose:

Appends values to the end of an array.

🧠 Syntax:

```
np.append(arr, values, axis=None)
```

🔸 **Parameters:**

- arr: Original array
- values: Values to append
- axis: By default None (flattens array before appending)

💡 Examples:

👉 1D Array:

```
arr = np.array([1, 2, 3])
new_arr = np.append(arr, [4, 5])
# Output: [1 2 3 4 5]
```

👉 2D Array (row-wise):

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

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

👉 2D Array (column-wise):

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

🔹 np.concatenate()

✅ **Purpose:**

Joins two or more arrays along an existing axis.

🧠 **Syntax:**

```
np.concatenate((arr1, arr2), axis=0)
```

🔸 **Parameters:**

- arr1, arr2: Arrays to join (as a tuple)
- axis: Axis along which to join (0 = rows, 1 = columns)

💡 **Examples:**

👉 1D Arrays:

```
a = np.array([1, 2])
b = np.array([3, 4])
result = np.concatenate((a, b))
# Output: [1 2 3 4]
```

👉 2D Arrays (row-wise):

```
a = np.array([[1, 2]])
b = np.array([[3, 4]])
result = np.concatenate((a, b), axis=0)
# Output:
```

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

👉 2D Arrays (column-wise):

```
a = np.array([[1], [2]])
b = np.array([[3], [4]])
result = np.concatenate((a, b), axis=1)
# Output:
# [[1 3]
#  [2 4]]
```

📘 NumPy delete() – Notes

✅ Purpose:

Deletes elements from a NumPy array along a specified axis.

🧠 Syntax:

```
np.delete(arr, obj, axis=None)
```

🔸 Parameters:

Parameter	Description
arr	The input array
obj	Index or list of indices to delete
axis	Axis to delete along (None = flatten first)

🔹 Examples

👉 **Delete from 1D Array**

```python
import numpy as np

arr = np.array([10, 20, 30, 40, 50])
new_arr = np.delete(arr, 2)
# Output: [10 20 40 50]
```

👉 **Delete Row in 2D Array**

```python
arr = np.array([[1, 2], [3, 4], [5, 6]])
new_arr = np.delete(arr, 1, axis=0)
# Deletes 2nd row (index 1)
# Output:
# [[1 2]
#  [5 6]]
```

👉 **Delete Column in 2D Array**

```python
arr = np.array([[1, 2, 3], [4, 5, 6]])
new_arr = np.delete(arr, 1, axis=1)
# Deletes 2nd column (index 1)
# Output:
# [[1 3]
#  [4 6]]
```

⚠️ Note:

- np.delete() **does not change** the original array.
- It returns a **new array** with the specified elements removed.

📘 NumPy vstack() and hstack() – Notes

🔹 np.vstack() → Vertical Stack

✅ **Purpose:**

Stacks arrays **vertically** (row-wise) — one on top of the other.

🧠 **Syntax:**

```
np.vstack((arr1, arr2))
```

💡 **Example:**

```
import numpy as np

a = np.array([1, 2])
b = np.array([3, 4])
result = np.vstack((a, b))
# Output:
# [[1 2]
#  [3 4]]
```

🔹 np.hstack() → Horizontal Stack

✅ **Purpose:**

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

🧠 **Syntax:**

```
np.hstack((arr1, arr2))
```

💡 **Example:**

```
a = np.array([1, 2])
b = np.array([3, 4])
result = np.hstack((a, b))
# Output: [1 2 3 4]
```

🔹 2D Array Example

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

v_result = np.vstack((a, b))
# Output:
# [[1 2]
#  [3 4]
#  [5 6]
```

```
#  [7 8]]

h_result = np.hstack((a, b))
# Output:
# [[1 2 5 6]
#  [3 4 7 8]]
```

✅ Summary

Function	Stacks	Direction
np.vstack()	Vertically	Top to bottom
np.hstack()	Horizontally	Side by side

📘 NumPy Split Functions – Notes

🔹 np.split()

✅ Purpose:

Splits an array into **equal parts**.

🧠 Syntax:

```
np.split(array, sections, axis=0)
```

- sections: Number of parts to split into
- axis=0: Split by rows (default)

- axis=1: Split by columns

💡 **Example:**

```
arr = np.array([10, 20, 30, 40, 50, 60])
np.split(arr, 3)
# Output: [array([10, 20]), array([30, 40]), array([50, 60])]
```

🔹 np.hsplit() → Horizontal Split

✅ **Purpose:**

Splits a 2D array **horizontally** (column-wise)

🧠 **Syntax:**

```
np.hsplit(array, sections)
```

💡 **Example:**

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

🔹 np.vsplit() → Vertical Split

✅ **Purpose:**

Splits a 2D array **vertically** (row-wise)

🧠 **Syntax:**

```
np.vsplit(array, sections)
```

💡 **Example:**

```
arr = np.array([[1, 2], [3, 4], [5, 6], [7, 8]])
np.vsplit(arr, 2)
# Output:
# [array([[1, 2],
#         [3, 4]]),
```

```
#  array([[5, 6],
#         [7, 8]])]
```

✅ Summary Table

Function	Use Case	Direction
np.split()	Split equally by axis	Custom axis
np.hsplit()	Split horizontally	By columns
np.vsplit()	Split vertically	By rows

🧠 NumPy Notes: Broadcasting and Vectorization

🔹 What is Vectorization?

Vectorization refers to the process of performing operations on entire arrays (vectors/matrices) **without using explicit loops**.

✅ **Benefits:**

- Faster execution
- Cleaner and more readable code
- Efficient memory usage

🧾 **Example:**

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

# Vectorized addition (no loop)
c = a + b   # Output: [5 7 9]
```

🔹 What is Broadcasting?

Broadcasting is a technique that allows NumPy to perform operations on arrays of **different shapes** as if they had the same shape.

✅ **Key Rules:**

1. If arrays have different dimensions, NumPy adds 1s to the smaller array's shape (on the left) until dimensions match.
2. Dimensions are compatible when:
   - They are equal, or
   - One of them is 1

🧾 **Example 1: Adding scalar to array**

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

b = 5

result = a + b  # Output: [6 7 8]
```

🧾 **Example 2: 2D and 1D array**

```
A = np.array([[1, 2, 3], [4, 5, 6]])
B = np.array([10, 20, 30])
result = A + B
# Output:
# [[11 22 33]
#  [14 25 36]]
```

📌 Summary:

Feature	Description
Vectorization	Perform operations on arrays without loops
Broadcasting	Automatically expands smaller arrays to match dimensions for operations

🧠 NumPy Notes: Missing Values

🔹 What are Missing Values?

Missing values represent **unknown or undefined data** in an array. In NumPy, missing values are often represented by:

🔹 np.nan (Not a Number)

- Represents **missing or undefined values**.
- Used for placeholder when data is incomplete.

```
import numpy as np

a = np.array([1, 2, np.nan, 4])
print(np.isnan(a))
# Output: [False False  True False]
```

🔹 np.nan_to_num()

- Replaces **NaN, positive infinity (inf)**, and **negative infinity (-inf)** with **specified or default values**.
- Default replacements:
  - np.nan → 0.0
  - +inf → large finite number
  - -inf → large negative finite number

```
a = np.array([1, np.nan, np.inf, -np.inf])
b = np.nan_to_num(a)

print(b)
# Output: [  1.   0.  max_float -max_float ]
```

✅ **With custom values:**

```
np.nan_to_num(a, nan=0, posinf=999, neginf=-999)
# Output: [1.0, 0.0, 999.0, -999.0]
```

🔹 np.isinf()

- Checks for both **positive and negative infinity** in an array.

```
a = np.array([1, np.inf, -np.inf, 5])
print(np.isinf(a))
# Output: [False  True  True False]
```

🔹 np.PINF (Positive Infinity)

- A constant representing **positive infinity**.

print(np.PINF) # Output: inf

🔹 np.NINF (Negative Infinity)

- A constant representing **negative infinity**.

print(np.NINF) # Output: -inf