📘 NumPy Notes

📌 What is NumPy?

NumPy (Numerical Python) is a **powerful library** for numerical computations in Python.
It provides:

- **Multidimensional arrays** (ndarray)
- **Mathematical functions** (like mean, sum, sqrt, etc.)
- **Linear algebra**, **Fourier transform**, **random number generation**, and more.

☑️ Why Use NumPy?

Feature	Benefit
Faster	NumPy uses C under the hood, making array operations **much faster** than regular Python loops or lists.
Memory Efficient	Stores data in **contiguous memory** locations.
Convenient	Supports broadcasting, vectorization, slicing, reshaping, etc.
Rich Ecosystem	Used in **Pandas**, **Matplotlib**, **Scikit-learn**, **TensorFlow**, etc.

⚖️ Difference Between NumPy Arrays and Python Data Structures

Feature	NumPy Array	Python List / Tuple
Speed	Very fast (uses C)	Slower
Memory	Less memory	More memory
Operations	Vectorized (e.g., a + 1)	Requires loops
Data Type	Homogeneous (all elements same type)	Heterogeneous
Functionality	Mathematical operations	Limited math support

NumPy Array Creation Methods

---

1. np.array()

Creates an array from a **Python list or tuple**.

```
import numpy as np

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

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

---

2. dtype (Data Type of Array Elements)

Specify or check the **data type** of elements in an array.

```
arr = np.array([1, 2, 3], dtype=float)

print(arr)        # [1. 2. 3.]

print(arr.dtype)   # float64
```

---

3. np.arange(start, stop, step)

Creates a range of numbers with a given step.

```
arr = np.arange(0, 10, 2)

print(arr)  # [0 2 4 6 8]
```

---

4. np.ones(shape)

Creates an array filled with **1s**.

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

print(arr)

# [[1. 1. 1.]

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

---

5. np.zeros(shape)

Creates an array filled with **0s**.

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

print(arr)

# [[0. 0.]

#  [0. 0.]

#  [0. 0.]]
```

---

6. np.eye(N)

Creates an **identity matrix** of size N x N.

```
arr = np.eye(3)

print(arr)

# [[1. 0. 0.]

#  [0. 1. 0.]

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

---

7. np.random.rand(shape)

Generates an array of **random numbers** between 0 and 1.

```
arr = np.random.rand(2, 3)

print(arr)
```

---

8. np.linspace(start, stop, num)

Generates num **evenly spaced values** between start and stop.

```
arr = np.linspace(0, 1, 5)

print(arr)  # [0.   0.25 0.5  0.75 1.  ]
```

NumPy Array Attributes

These attributes help you understand the **structure and memory** details of a NumPy array.

---

1. 🔹 ndim – Number of Dimensions

Returns how many dimensions (axes) the array has.

```
import numpy as np


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

print(a.ndim)  # Output: 2
```

---

2. 🔹 shape – Shape of the Array

Returns a **tuple** showing the size of each dimension (rows, columns).

```
print(a.shape)  # Output: (2, 3) → 2 rows, 3 columns
```

---

3. 🔹 size – Total Number of Elements

Returns the **total number of elements** in the array.

```
print(a.size)  # Output: 6 → because 2 × 3 = 6 elements
```

---

4. 🔹 itemsize – Memory Size of One Element (in Bytes)

Tells how many **bytes each element** takes in memory.

```
print(a.itemsize)  # Output: 4 (if dtype is int32)
```

---

5. 🔹 dtype – Data Type of Array Elements

Returns the **data type** (int32, float64, etc.) of array elements.

```
print(a.dtype)  # Output: int32 or int64 (depends on your system)
```

🔁 All Together Example

```
a = np.array([[10, 20, 30], [40, 50, 60]], dtype=np.int32)


print("ndim:", a.ndim)       # 2
print("shape:", a.shape)     # (2, 3)
print("size:", a.size)       # 6
print("itemsize:", a.itemsize)  # 4 bytes
print("dtype:", a.dtype)     # int32
```

🔄 Changing Data Type in NumPy Arrays

NumPy allows you to **change or convert** the data type (dtype) of array elements using several methods.

✅ 1. Using astype() Method

The astype() function is used to **explicitly convert** an array to a different data type.

📌 Syntax:

```
array.astype(new_dtype)
```

📘 Example:

```
import numpy as np
arr = np.array([1, 2, 3])
new_arr = arr.astype(float)
print(new_arr)      # [1. 2. 3.]
print(new_arr.dtype) # float64
```

✅ 2. Converting from Float to Integer

This **truncates** the decimal values.

```
arr = np.array([1.5, 2.7, 3.2])
int_arr = arr.astype(int)
```

```
print(int_arr)      # [1 2 3]
print(int_arr.dtype) # int32 or int64
```

✅ 3. Converting to String Type

```
arr = np.array([10, 20, 30])
str_arr = arr.astype(str)

print(str_arr)      # ['10' '20' '30']
print(str_arr.dtype) # <U2 (Unicode string)
```

✅ 4. Converting Boolean to Integer or Float

```
bool_arr = np.array([True, False, True])

int_arr = bool_arr.astype(int)
float_arr = bool_arr.astype(float)

print(int_arr)   # [1 0 1]
print(float_arr) # [1. 0. 1.]
```

✅ 5. Checking the Data Type Before and After Conversion

```
arr = np.array([1, 2, 3])
print("Before:", arr.dtype)

arr = arr.astype('float32')
print("After:", arr.dtype)
```

🧠 Why Change Data Types?

Purpose	Example
Save memory	Use float32 instead of float64
Prepare data for ML models	Convert strings/booleans to numbers
Avoid errors during calculations	Convert strings to numeric types

NumPy Array Operations

1. Scalar Operations

These are operations where a single scalar value is applied to each element of the array.

Let:
a = np.array([1, 2, 3, 4])

Arithmetic with Scalar:

- a + 5 → [6, 7, 8, 9]
- a - 2 → [-1, 0, 1, 2]
- a * 3 → [3, 6, 9, 12]
- a / 2 → [0.5, 1.0, 1.5, 2.0]
- a ** 2 → [1, 4, 9, 16]

Relational with Scalar:

- a > 2 → [False, False, True, True]
- a < 4 → [True, True, True, False]
- a == 3 → [False, False, True, False]
- a != 1 → [False, True, True, True]
- a >= 2 → [False, True, True, True]
- a <= 3 → [True, True, True, False]

These operations are **element-wise**, meaning they apply to each value in the array.

---

2. Vector (Element-wise) Arithmetic

These operations are performed between two arrays of the same shape.

Let:
x = np.array([1, 2, 3])
y = np.array([4, 5, 6])

Arithmetic:

- x + y → [5, 7, 9]
- x - y → [-3, -3, -3]
- x * y → [4, 10, 18]
- x / y → [0.25, 0.4, 0.5]
- x ** y → [1, 32, 729]

---

3. Vector Relational Operations

These comparisons are done element by element between two arrays.

Let:
x = np.array([10, 20, 30])
y = np.array([15, 20, 25])

Relational:

- x > y → [False, False, True]
- x < y → [True, False, False]
- x == y → [False, True, False]
- x != y → [True, False, True]
- x >= y → [False, True, True]
- x <= y → [True, True, False]

These return **Boolean arrays** used for filtering or logical operations.

NumPy Array Functions

==========

🔹 **Max, Min, Sum, Prod**

These functions return **summary values** from the array.

✅ **Basic Usage:**

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

- np.max(arr) → 4
- np.min(arr) → 1
- np.sum(arr) → 10
- np.prod(arr) → 24

✅ **With axis:**

- np.sum(arr, axis=0) → [4, 6] → (column-wise sum)
- np.sum(arr, axis=1) → [3, 7] → (row-wise sum)

Explanation:

- axis=0: Works **vertically** (down columns)
- axis=1: Works **horizontally** (across rows)

🔹 **Mean, Median, Std, Var (Statistics Functions)**

✅ **Basic Usage:**

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

- np.mean(arr) → 2.5
- np.median(arr) → 2.5
- np.std(arr) → 1.118
- np.var(arr) → 1.25

✅ **With axis:**

- np.mean(arr, axis=0) → [2.0, 3.0]
- np.mean(arr, axis=1) → [1.5, 3.5]

🔹 **Trigonometric Functions**

These work **element-wise** on angle values in **radians**.

Let arr = np.array([0, np.pi/2])

- np.sin(arr) → [0.0, 1.0]
- np.cos(arr) → [1.0, 0.0]
- np.tan(arr) → [0.0, undefined]

🔹 **Dot Product**

Used for **matrix multiplication** or **inner product** of vectors.

✅ **Example:**

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

- np.dot(a, b) → 11 (1×3 + 2×4)

For matrices:
Let
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])

- np.dot(A, B) →

lua

CopyEdit

[[19, 22],

[43, 50]]

🔹 **Logarithm & Exponential Functions**

Let arr = np.array([1, 2])

- np.log(arr) → [0.0, 0.693]
- np.exp(arr) → [2.718, 7.389]

🔹 **Rounding Functions**

Let arr = np.array([1.4, 2.6, 3.5])

- np.round(arr) → [1.0, 3.0, 4.0]

- np.floor(arr) → [1.0, 2.0, 3.0]
- np.ceil(arr) → [2.0, 3.0, 4.0]

NumPy Indexing and Slicing

==========

🔹 1D Array

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

✅ **Indexing:**

- arr[0] → 10
- arr[-1] → 50 (last element)
- arr[2] → 30

✅ **Slicing:**

- arr[1:4] → [20, 30, 40]
- arr[:3] → [10, 20, 30]
- arr[2:] → [30, 40, 50]
- arr[::2] → [10, 30, 50] (every 2nd element)

🔹 2D Array

Let:
arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

✅ **Indexing:**

- arr[0][1] or arr[0, 1] → 2
- arr[2][2] or arr[2, 2] → 9
- arr[-1, -1] → 9 (last row, last column)

✅ **Row Access:**

- arr[0] → [1, 2, 3] (first row)

- arr[1, :] → [4, 5, 6] (second row, all columns)

✅ **Column Access:**

- arr[:, 0] → [1, 4, 7] (first column)
- arr[:, 2] → [3, 6, 9] (third column)

✅ **Slicing:**

- arr[0:2, 1:3] →

[[2, 3],

[5, 6]]

- arr[:2, :] → first 2 rows
- arr[:, 1:] → all rows, columns from index 1 onward

🔹 **3D Array**

Let:

arr = np.array([

[[1, 2], [3, 4]],

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

])

Shape: (2, 2, 2)

✅ **Indexing:**

- arr[0, 0, 0] → 1
- arr[1, 1, 1] → 8
- arr[0] →

[[1, 2],

[3, 4]]

✅ **Slicing:**

- arr[0, :, :] → 2D slice from first block
  → [[1, 2], [3, 4]]
- arr[:, 1, :] → second row from all blocks
  → [[3, 4], [7, 8]]

- arr[:, :, 0] → first column from every 2x2
  → [[1, 3], [5, 7]]

NumPy Iteration (1D, 2D, 3D Arrays)

==========

🔹 1D Array Iteration

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

✅ **Iterating:**

```
for i in arr:
    print(i)
```

Output:
10
20
30

Each element is directly accessed one by one.

🔹 2D Array Iteration

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

✅ **Iterating Row by Row:**

```
for row in arr:
    print(row)
```

Output:
[1 2 3]
[4 5 6]

✅ **Iterating Each Element (Nested Loop):**

```
for row in arr:
```

```
    for item in row:
        print(item)
```

Output:
1 2 3 4 5 6

✅ **Using .flat to Flatten:**

```
for item in arr.flat:
    print(item)
```

🔹 **3D Array Iteration**

Let:

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

Shape: (2, 2, 2)

✅ **Iterating Block by Block:**

```
for block in arr:
    print(block)
```

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

✅ **Nested Loop for All Elements:**

```
for block in arr:
    for row in block:
        for item in row:
            print(item)
```

Output:
1 2 3 4 5 6 7 8

✅ **Flattened Iteration:**

```
for item in arr.flat:
    print(item)
```

✅ **Note on .nditer()**

NumPy also provides np.nditer() for efficient and flexible iteration over arrays of any shape.

```
for x in np.nditer(arr):
    print(x)
```

Works for **1D, 2D, 3D** arrays and simplifies nested loops.

NumPy: reshape, transpose, and ravel

==========

🔹 **1. reshape()**

Used to change the **shape** of the array **without changing data**.

✅ **Syntax:**

```
array.reshape(new_shape)
```

✅ **Example:**

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

Output:

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

- The **total number of elements** must remain the same.
- You can use -1 to auto-calculate one dimension:
- arr.reshape(3, -1) → shape will be (3, 2)

🔹 2. transpose()

Used to **swap the axes** of an array. Commonly used to **convert rows to columns and vice versa**.

✅ **Syntax:**

array.transpose()

✅ **Example:**

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

transposed = arr.transpose()

Output:

[[1, 3],

[2, 4]]

For 2D: switches rows and columns.
For 3D: can rearrange multiple axes using:

arr.transpose(1, 0, 2)

🔹 3. ravel()

Returns a **flattened 1D array** from any n-dimensional array. Unlike reshape, ravel() is often a **view** (no copy).

✅ **Syntax:**

array.ravel()

✅ **Example:**

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

flattened = arr.ravel()

Output:

[1, 2, 3, 4]

- Similar to flatten(), but ravel() is **faster** and uses less memory.

NumPy: vstack, hstack, vsplit, hsplit

==========

🔹 1. np.vstack() – Vertical Stacking

Stacks arrays **vertically** (along rows). The arrays are stacked **one on top of the other**.

✅ **Syntax:**

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

✅ **Example:**

arr1 = np.array([1, 2, 3])

arr2 = np.array([4, 5, 6])

stacked = np.vstack((arr1, arr2))

Output:

[[1, 2, 3],

[4, 5, 6]]

- The **number of columns** must be the same.

🔹 2. np.hstack() – Horizontal Stacking

Stacks arrays **horizontally** (along columns). The arrays are stacked **side by side**.

✅ **Syntax:**

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

✅ **Example:**

arr1 = np.array([1, 2, 3])

arr2 = np.array([4, 5, 6])

stacked = np.hstack((arr1, arr2))

Output:

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

- The **number of rows** must be the same.

🔹 3. np.vsplit() – Vertical Splitting

Splits the array **vertically** (along rows). The array is split into **multiple sub-arrays** along the first axis (rows).

✅ **Syntax:**

np.vsplit(arr, indices_or_sections)

✅ **Example:**

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

split_arr = np.vsplit(arr, 3)

Output:

[array([[1, 2, 3]]),

array([[4, 5, 6]]),

array([[7, 8, 9]])]

- **Divides the array** into the specified number of **rows**.

🔹 4. np.hsplit() – Horizontal Splitting

Splits the array **horizontally** (along columns). The array is split into **multiple sub-arrays** along the second axis (columns).

✅ **Syntax:**

np.hsplit(arr, indices_or_sections)

✅ **Example:**

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

split_arr = np.hsplit(arr, 3)

Output:

[array([[1], [4], [7]]),

array([[2], [5], [8]]),

array([[3], [6], [9]])]

- **Divides the array** into the specified number of **columns**.
