NumPy

created by : The easylearn academ

oduction to NumPy

Py is a powerful Python library used for performing large-scale mathematical and scientific computa

ently, quickly, and flexibly.

onsidered indispensable in fields such as **Data Science**, **Machine Learning**, and **Scientific Comp**

o its performance and versatility.

Py provides data structures—most notably the **ndarray (n-dimensional array)**—which are simi

n lists but far more efficient in terms of memory usage and computational speed.

nensional array arrays enable fast and vectorized operations, making complex numerical

htforward and optimized.

ing NumPy, we can store and manipulate data in a way that is less resource-intensive compared to

n structures. It also supports various mathematical functions, linear algebra operations, and too

ating with other scientific libraries.

eature of NumPy

-Dimensional Arrays

Py offers a powerful n-dimensional array object called ndarray, which provides efficient storage and manipulation

ets in multiple dimensions.

nsive Mathematical Functions

udes a comprehensive collection of mathematical operations such as linear algebra, statistical functions, Fourier

orms, and random number generation.

dcasting

es arithmetic operations on arrays of different shapes and sizes without the need for explicit loops, making code

nore efficient.

Performance

Py is implemented in C, allowing operations to be executed much faster than standard Python code, especially wi

datasets.

nless Integration

y integrates with other key scientific and data libraries such as **Pandas**, **Matplotlib**, **Scikit-learn**, and **TensorFlov**

g it a cornerstone of the Python data ecosystem.

is NumPy Important?

ation for Scientific Computing in Python

y serves as the core library for numerical operations in Python. It provides the building blocks for more advanced libraries like Par TensorFlow, and Scikit-learn.

mance and Efficiency

y arrays are significantly faster and more memory-efficient than Python lists, especially for large datasets. This performance is cru dealing with high volumes of data in real-world applications.

es Vectorized Operations

umPy, operations can be applied to entire arrays without using loops. This makes code shorter, easier to read, and computationa — a key advantage in data analysis and machine learning workflows.

t Mathematical Tools

y provides a wide range of tools for linear algebra, statistics, random number generation, and more — all of which are essential fo e and scientific research.

perability and Ecosystem Support

y integrates seamlessly with other Python libraries used in data science, making it the glue that holds the Python scientific compu stem together.

-Disciplinary Applications

physics to finance, machine learning to image processing, NumPy is used across domains wherever numerical computation is inv

oduction to Jupyter Notebook

pyter Notebook is an open-source web-based interactive compu
ironment that allows you to:

- **Write and run code** (mainly Python) directly in your browser
- **Create visualizations**
- **Write formatted notes** using Markdown
- **Mix code, text, and output** in a single document (notebook)

widely used in **Data Science, Machine Learning, Education,** a
search because it supports interactive, step-by-step development
umentation.

can install jupyter extension from visual studio code

to Install NumPy

u can install NumPy using **pip**, which is the standard Python ckage manager.

en your terminal (Command Prompt, PowerShell, or terminal in Code or PyCharm).

n the following command:

```
pip install numpy
```

Verify Installation:

```
import numpy
print(numpy.__version__)
```

s confirms that NumPy is successfully installed and tells you ich version is active.

Array functions in numpy

Key Differences Between List and NumPy Array

ure	Python List	NumPy Array
Type	Can hold mixed types (int, float, str)	Must have the same data type (i float, etc.)
ormance	Slower (due to dynamic typing)	Faster (optimized for numerical operations)
ory Usage	Takes more memory	Uses less memory (efficient storage)
ematical ations	Need explicit loops (e.g., for loop)	Supports vectorized operations (faster calculations)
-in Methods	General-purpose functions (append(), sort(), etc.)	Specialized functions for math, stats, and linear algebra
-Dimensional ort	Only 1D (nested lists for more)	Supports multi-dimensional arra (e.g., 2D, 3D)
bility	More flexible, stores mixed data types	Optimized for numerical comput

When to Use Each?

Use **Python Lists** for:

- General-purpose programming.
- Heterogeneous or small data that doesn't require heavy numerical processing.

Use **NumPy Arrays** for:

- Mathematical computations.
- Large datasets where performance and memory efficiency are critical.
- Scientific computing, machine learning, and data analysis tasks.

ay

e `np.array()` to convert a list to a NumPy array. which is a powerful d
ucture for numerical computing.

eates Arrays from Lists or Tuples:

Converts Python lists, tuples, or other sequences into a NumPy array, ena
efficient numerical operations.

pports Multi-Dimensional Arrays:

Creates 1D, 2D (matrices), or higher-dimensional arrays for complex data.

tomatically converts elements to a common data type for consistency.

de Example:

```
nport numpy as np
rr = np.array([1, 2, 3])   #create 1d array
rint(arr)                  # Output: [1 2 3]
 = np.array([1,2,3,4.5])
rint(b)
```

rray

2D array in NumPy is a two-dimensional data structure,

ne should visualize it as a grid or matrix with rows and columns.

s used to represent tabular data, images, or mathematical matrices.

ch element in a 2D array can be accessed using two indices: one fo
v and one for the column.

ke all NumPy arrays, 2D arrays are memory-efficient and sup
ctorized operations.

here it is used?

ommonly used in data science, machine learning, and scientific compu
tasks like matrix operations, image processing, or storing datasets.

Array function on 2d array


```
arr_2d = np.array([[1, 2,3,4], [5,6,7,8]])
print(arr_2d)
print(arr_2d.shape)  # Output: (2,4)
```

Creating Arrays with Methods

nPy zeros()

e zeros() method creates a new array of given shape and type, filled with os.

mpy.zeros(shape, dtype = None, order = 'C')

e zeros() method takes three arguments:

- shape - desired shape of the new array (can be int or tuple of int)
- dtype (optional) - datatype of the new array
- order (optional) - specifies the order in which the zeros are filled

e zeros() method returns the array of given shape, order, and atype filled with 0s.

```
rt numpy as np
eate an array of 5 elements filled with 0s
y1 = np.zeros(5)
t(array1)  # Output: [0. 0. 0. 0. 0.]
```

nPy ones()

ones() method creates a new array of given shape and type, filled with ones.

npy.ones(shape, dtype = None, order = 'C')

ones() method takes three arguments:

- shape - desired new shape of the array (can be integer or tuple of integers)
- dtype (optional) - datatype of the returned array
- order (optional) - specifies the order in which the ones are filled

ones() method returns the array of given shape, order, and datatype filled with 1s.

```
rt numpy as np
eate a float array of 1s
y1 = np.ones(5)
('Float Array: ',array1) #Float Array:  [1. 1. 1. 1. 1.]
eate an int array of 1s
y2 = np.ones(5, dtype = int)
('Int Array: ',array2) #Int Array:  [1 1 1 1 1]
y3 = np.ones([2,3])
('n-d array:\n',array3)
. 1. 1.]
 1. 1.]]
```

nPy arange()

arange() method creates an array with evenly spaced elements as per the interval.

py.arange(start = 0, stop, step = 1, dtype = None)

e arange() method takes the following arguments:

start(optional)- the start value of the interval range (int or real)

stop- the end value of the interval range (exclusive) (int or real)

step(optional)- step size of the interval (int or real)

dtype(optional)- type of output array(dtype)

arange() method returns an array of evenly spaced values.

```
ort numpy as np
reate an array with first five elements
ay1 = np.arange(5) #[0 1 2 3 4]
reate an array with elements from 5 to 10(exclusive)
ay2 = np.arange(5, 10) #[5 6 7 8 9]
reate an array with elements from 5 to 15 with stepsize 2
ay3 = np.arange(5, 15, 2) #[ 5  7  9 11 13]
nt(array1,array2,array3)
```

nPy linspace()

e linspace() method creates an array with evenly spaced elements over an rval.

mpy.linspace(start, stop, num = 50, endpoint = True, retstep = False, dt ne, axis = 0)

- start- the start value of the sequence, **0** by default (can be array_like)
- stop- the end value of the sequence (can be array_like)
- num(optional)- number of samples to generate (int)
- endpoint(optional)- specifies whether to include end value (bool)
- retstep(optional)- if True, returns steps between the samples (bool)
- dtype(optional)- type of output array
- axis(optional)- axis in the result to store the samples(int)

e linspace() method returns an array of evenly spaced values.

Example of linespace function

```
mport numpy as np
 create an array of 5 elements between 2.0 and 3.0
rray1 = np.linspace(2.0, 3.0, num=5)
rint(array1) #Array1: [2.   2.25 2.5  2.75 3.  ]

 create an array of 5 elements between 2.0 and 3.0 excluding the endpoint
rray2 = np.linspace(2.0, 3.0, num=5, endpoint=False)
rint("Array2:", array2) # #Array2: [2.  2.2 2.4 2.6 2.8]

 create an array of 5 elements between 2.0 and 3.0 with the step size included
rray3, step_size = np.linspace(2.0, 3.0, num=5, retstep=True)
rint("Array3:", array3) #Array3: [2.   2.25 2.5  2.75 3.  ]
rint("Step Size:", step_size) #Step Size: 0.25
```

Key Differences Between arange and linspace

Both np.arange() and np.linspace() are NumPy functions used to generate numerical sequences, but they have some differences in their behavior.

arange() generates a sequence of values from start to stop with a given step size whereas linspace generates a sequence of num evenly spaced values from start to stop.

arange() excludes stop value whereas linspace includes stop value unless specified otherwise by endpoint = False

nPy copy()

copy() method returns an array copy of the given object.

py.copy(array)

e copy() method takes has 1 argument:

array - input data

arange() method returns an array of evenly spaced values.

```
numpy as np
/ an array from another array
 = np.arange(5)
 = np.copy(array0)
Array copied from Array: ',array1) # [0 1 2 3 4]
/ an array from a list
[1, 2, 3, 4, 5]
 = np.copy(list1) #[1 2 3 4 5]
Array copied from List: ',array2)
/ an array from another array
 = (1, 2, 3, 4, 5)
 = np.copy(tuple1)
Array copied from Tuple: ',array1) #[1 2 3 4 5]
```

3D ARRAY

A **3D array** in NumPy is essentially an array of 2D arrays — like a cube of numbers. It's also called a **"tensor"** with 3 axes:

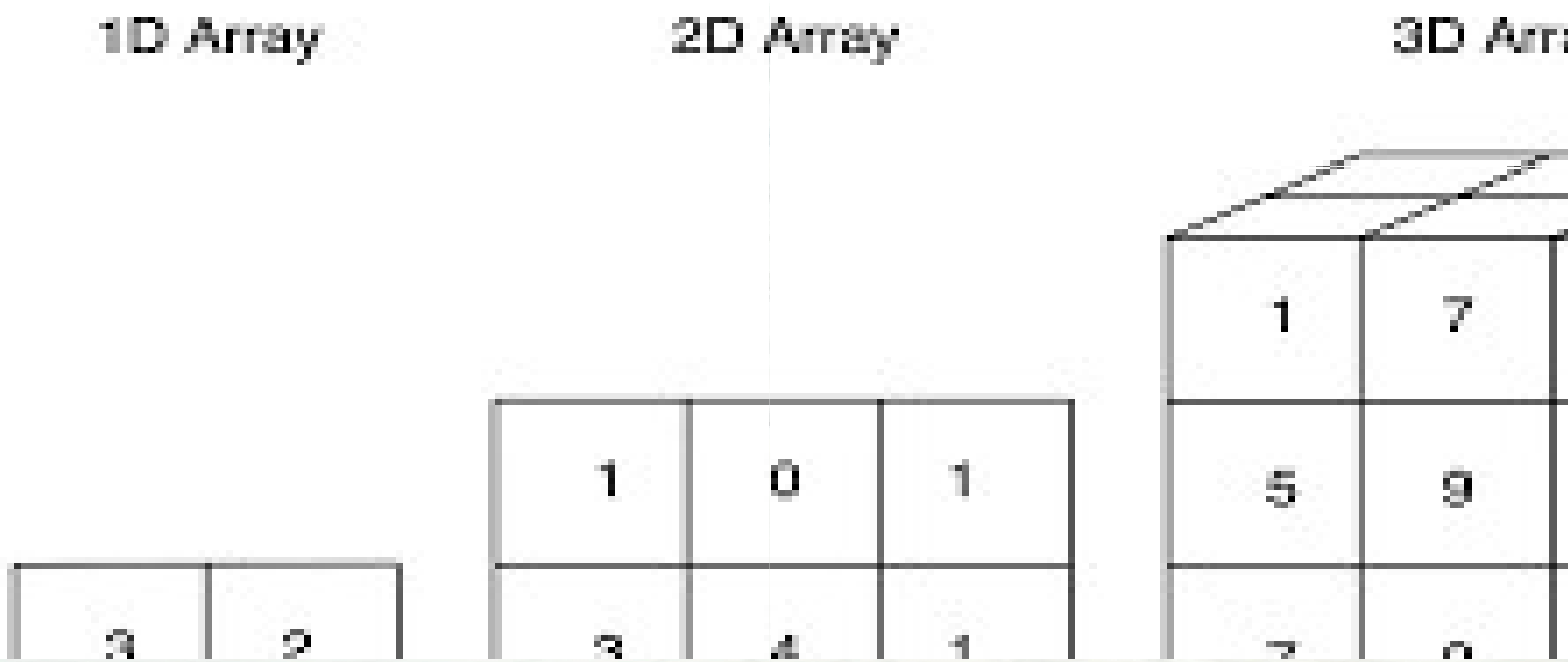
- **Axis 0** → depth (number of 2D arrays)
- **Axis 1** → rows
- **Axis 2** → columns

You can think of it like stacking **multiple 2D matrices** on top of each other.

Uses of 3D Arrays

1. Image processing (color channels: RGB)
2. Scientific simulations
3. Deep learning (multi-dimensional data)
4. Time-series over 2D grids

Different type of array



at are the array attributes?

bute	Description	Example Output
ray.ndim	Number of dimensions (axes)	2 for a 2D array
ray.shape	Tuple representing the size of each dimension	(3, 4) for 3 rows, 4 columns
ray.size	Total number of elements in the array	12 for a 3×4 array
ray.dtype	Data type of the array elements	int32, float64, etc.
ray.itemsize	Size (in bytes) of each element	4 bytes for int32
ray.nbytes	Total memory (in bytes) consumed by the array	48 for 12 elements × 4 bytes
ray.T	Transposed array (rows become columns and vice versa)	Flips axes

example

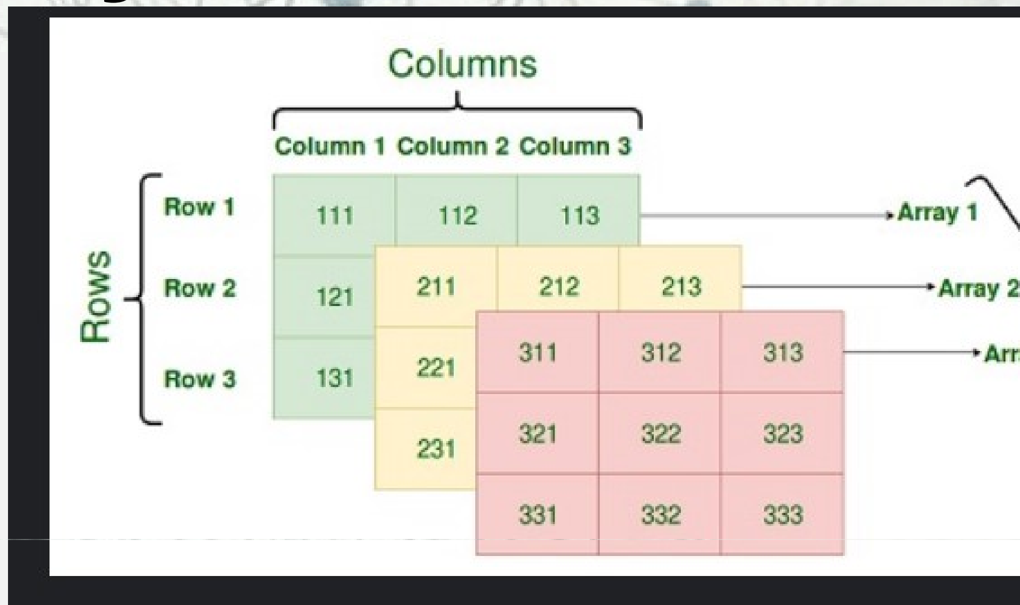
```
import numpy as np
arr = np.array([[1, 2, 3], [4, 5, 6]])
print("Array:\n", arr)
print("Dimensions (ndim):", arr.ndim) # 2
print("Shape:", arr.shape) #(2,3)
print("Size:", arr.size) # 6
print("Data type:", arr.dtype) # int64
print("Item size (bytes):", arr.itemsize) #8
print("Total bytes:", arr.nbytes) #64
print("Transpose:\n", arr.T)
```

ow to Create a 3D Array?

```
numpy as np
array([[[311, 312, 313],
  [321, 322, 323],
  [331, 332, 333]],

 [[211, 212, 213],
  [221, 222, 223],
  [231, 232, 233]],

  [[111, 112, 113],
  [121, 122, 123],
  [131, 132, 133]]])
)
```

Output of 3d array

```
'''
[[[311 312 313]
  [321 322 323]
  [331 332 333]]

 [[211 212 213]
  [221 222 223]
  [231 232 233]]

 [[111 112 113]
  [121 122 123]
  [131 132 133]]]
'''
```



NumPy Array Data Types

tegory	Examples	Description
eger	int8, int16, int32, int64	Signed integers of different sizes
signed Int	uint8, uint16, uint32, uint64	Non-negative integers
at	float16, float32, float64	Real numbers (decimals)
mplex	complex64, complex128	Complex numbers (real + imaginary)
olean	bool_	True / False
ing	str_, unicode_	Fixed-length strings
ject	object_	Any Python object (slower, less efficient)

Numpy Random

In NumPy, we have a module called random which provides functions for generating random numbers.

These functions can be useful for generating random inputs for testing algorithms.

It has following important functions.

mportant function in random module

unction	Description
and()	Uniform distribution over [0, 1). Accepts shape as positional arguments.
andn()	Standard normal distribution (mean=0, std=1). For normally distributed data
andint(low, high, size)	Random integers from low (inclusive) to high (exclusive).
andom()	Random floats in [0.0, 1.0). Similar to rand() but shape passed as a tuple.
hoice()	Randomly selects elements from a given array.
shuffle()	Randomly shuffles elements in a one-dimensional array in place.

Example of function in random module.

```
mport numpy as np

p.random.seed(42)                 # Ensures reproducibility
rint(np.random.rand(2, 2))        # 2x2 array of random floats
rint(np.random.randint(1, 10, 5))    # 5 random integers from 1 to 9
rint(np.random.normal(0, 1, 3))      # create 1d array of float type
rint(np.random.choice([1, 2, 3]))    # Randomly picks one element

rr = np.array([10, 20, 30, 40, 50])  # Define array
p.random.shuffle(arr)                # Shuffles array in-place
rint(arr)
```

Creating Complex Numbers in NumPy

You can use numpy.array() to create an array of complex numbers, just like any other NumPy array.

Here difference is that you define your numbers with a j to indicate the imaginary part.

```
mport numpy as np

# Creating a complex number array
complex_num = np.array([3 + 4j, 1 - 2j])
print("Complex Array:", complex_num)

# Checking the data type
print("Data Type:", complex_num.dtype)
```

ccessing array

umPy, you can access elements, rows, columns, or sub-arrays using indexing and
g similar to Python lists, but with more power

```
t numpy as np
 np.array([[10, 20, 30], [40, 50, 60]])
ss element at 1st row, 2nd column:
(arr[0, 1])  # Output: 20
cing Arrays like list in python
(arr[:, 1])    # All rows, 2nd column: [20 50]
(arr[1, :])    # 2nd row, all columns: [40 50 60]
(arr[0:2, 1:3])  # Subarray from row 0–1 and column 1–2

30]
60]]

ean Indexing (can not use and and or operator)
(arr[arr > 30])  # Output: [40 50 60]
y Indexing
(arr[[0, 1], [1, 2]])  # Elements at (0,1) and (1,2): [20 60]
```

now to change value in array?

```
mport numpy as np
rr = np.array([10, 20, 30])
rr[1] = 99
rint(arr)  # Output: [10 99 30]
 Change a Value in a 2D Array
rr2d = np.array([[1, 2], [3, 4]])
rr2d[1, 0] = 100
rint(arr2d)
 Change Multiple Values Using Slicing
rr[0:2] = [111, 222]
rint(arr)  # Output: [111 222  30]
 Change Values Based on a Condition
rr[arr > 100] = 0
rint(arr)  # Output: [  0   0  30]
```

metic Operations (Element-wise)

Py allows addition, subtraction, multiplication, and division to be performed directly be
s of the same shape, applying the operation to each corresponding element.

ar Operations:

ou can perform operations between an array and a single number (scalar), and the opera
pplied to every element in the array.

ematical Functions:

lumPy provides built-in functions like np.sqrt(), np.exp(), np.log(), and np.sin() which
lement-wise on arrays to perform common mathematical computations.

parison Operations:

ou can compare arrays using operators like >, <, ==, etc., and NumPy will return a bo
rray showing the result for each element.

egate Functions:

unctions such as np.sum(), np.mean(), np.min(), and np.max() help you compute sumr
ke total, average, minimum, or maximum across all elements in an array.

x Operations:

or true matrix mathematics (not element-wise), you can use @ or np.dot() for
nultiplication and .T to transpose arrays.

Let see example 1 of 2

```
mport numpy as np
 = np.array([10, 20, 30])
 = np.array([1, 2, 3])
basic maths operations
 Both arrays must be of same shape, or be broadcast-compatibl
rint(a + b)   # [11 22 33]
rint(a - b)   # [ 9 18 27]
rint(a * b)   # [10 40 90]
rint(a / b)   # [10. 10. 10.]
Scalar Operations
rint(a + 5)   # [15 25 35]
rint(a * 2)   # [20 40 60]
Mathematical Functions
rint(np.sin(a))       # Sine of each element
rint(np.sqrt(a))      # Square root
rint(np.exp(a))       # Exponential (e^x)
rint(np.log(a))       # Natural log
```

Let see example 2 of 2

```
mport numpy as np
 = np.array([10, 20, 30])
 = np.array([1, 2, 3])
Comparison Operations
rint(a > 15)   # [False  True  True]
 Matrix Operations (Not Element-wise)
 = np.array([[1, 2], [3, 4]])
 = np.array([[5, 6], [7, 8]])
rint(A @ B)         # Matrix multiplication
rint(np.dot(A, B)) # Same result
```

Aggregate Functions

```
rr = np.array([10, 20, 30, 40, 50])
rint("Sum: ", np.sum(arr))          # Output: 150
rint("Mean: ", np.mean(arr))        # Output: 30.0
rint("Min: ", np.min(arr))          # Output: 10
rint("Max: ", np.max(arr))          # Output: 50
```

ay Comparison in NumPy

ement-wise Comparison:

ng == or != compares each element of one array with the corresponding element in another array.

sult is a Boolean Array:

e output is a new array containing True or False for each comparison result.

= b:

turns True where elements of a and b are equal, False otherwise.

= b:

turns True where elements are different, False where they are the same.

all(condition):

turns True only if **all** elements in the condition are True.

any(condition):

turns True if **at least one** element in the condition is True.

eful for Validation and Filtering:

ese comparisons help in filtering arrays or checking conditions efficiently.

example

```
mport numpy as np
 = np.array([1, 2, 3])
 = np.array([1, 0, 3])
```

Comparing Two Arrays (Element-wise)

```
rint(a == b)      # [ True False  True]
rint(a != b)      # [False  True False]
Check if All or Any Elements Match
rint(np.all(a == b))   # False (not all elements match)
rint(np.any(a == b))   # True (at least one match)
```

ray_equal()

- he array_equal() method in NumPy is used to check if two arrays ave the same shape and elements, i.e.,
- returns True if the two arrays are equal, and False otherwise.
- compares both the data and the shape of the arrays.
- yntax
- umpy.array_equal(a, b) #both a and b must be array
- et see example

xample

```
mport numpy as np
 Two arrays with the same shape and elements
rr1 = np.array([1, 2, 3])
rr2 = np.array([1, 2, 3])
 Check if they are equal
rint(np.array_equal(arr1, arr2))  # Output: True
 Arrays with different elements
rr3 = np.array([1, 2, 4])
rint(np.array_equal(arr1, arr3))  # Output: False
 Arrays with different shapes
rr4 = np.array([[1, 2, 3]])
rint(np.array_equal(arr1, arr4))  # Output: False
```

gical operations

gical operations on NumPy arrays allow you to perform **element-wis** gical operations such as AND, OR, and NOT on boolean arrays or array ontaining numerical values.

hese operations are useful when filtering or selecting data based o onditions.

ogical Operators in NumPy:

1. **np.logical_and()**: Performs element-wise logical AND.
2. **np.logical_or()**: Performs element-wise logical OR.
3. **np.logical_not()**: Performs element-wise logical NOT.
4. **np.logical_xor()**: Performs element-wise logical XOR.

hese operations return boolean arrays where each element is the result o pplying the logical operation on corresponding elements of the input arrays.

et us see example

ample

```
t numpy as np
ample arrays
= np.array([1, 2, 3, -1, 5])
= np.array([4, 0, 2, 7, 3])
. Logical AND (True where both conditions are True)
t_and = np.logical_and(arr1 > 0, arr2 < 5)
("Logical AND:", result_and)
put: [ True False  True False  True]
. Logical OR (True where at least one condition is True)
t_or = np.logical_or(arr1 > 0, arr2 < 5)
("Logical OR:", result_or)
put: [ True  True  True  True  True]
. Logical NOT (True where the condition is False)
t_not = np.logical_not(arr1 > 0)
("Logical NOT:", result_not)
put: [False False False  True False]
. Logical XOR (True where only one of the conditions is True, not both)
t_xor = np.logical_xor(arr1 > 0, arr2 < 5)
("Logical XOR:", result_xor)
put: [ True  True False  True False]
```