NumPy

created by : The easylearn academy

Introduction to NumPy

- **NumPy** is a powerful Python library used for performing large-scale mathematical and scientific computations efficiently, quickly, and flexibly.
- It is considered indispensable in fields such as **Data Science**, **Machine Learning**, and **Scientific Computing** due to its performance and versatility.
- NumPy provides data structures—most notably the **ndarray (n-dimensional array)**—which are similar to Python lists but far more efficient in terms of memory usage and computational speed.
- **n-dimensional array** arrays enable fast and vectorized operations, making complex numerical tasks straightforward and optimized.
- By using NumPy, we can store and manipulate data in a way that is less resource-intensive compared to native Python structures. It also supports various mathematical functions, linear algebra operations, and tools for integrating with other scientific libraries.

Key feature of NumPy

1. **Multi-Dimensional Arrays**
   NumPy offers a powerful n-dimensional array object called ndarray, which provides efficient storage and manipulation of large datasets in multiple dimensions.
2. **Extensive Mathematical Functions**
   It includes a comprehensive collection of mathematical operations such as linear algebra, statistical functions, Fourier transforms, and random number generation.
3. **Broadcasting**
   Enables arithmetic operations on arrays of different shapes and sizes without the need for explicit loops, making code simpler and more efficient.
4. **High Performance**
   NumPy is implemented in C, allowing operations to be executed much faster than standard Python code, especially with large datasets.
5. **Seamless Integration**
   Easily integrates with other key scientific and data libraries such as **Pandas**, **Matplotlib**, **Scikit-learn**, and **TensorFlow**, making it a cornerstone of the Python data ecosystem.

Why is NumPy Important?

- **Foundation for Scientific Computing in Python**
  NumPy serves as the core library for numerical operations in Python. It provides the building blocks for more advanced libraries like Pandas, SciPy, TensorFlow, and Scikit-learn.

- **Performance and Efficiency**
  NumPy arrays are significantly faster and more memory-efficient than Python lists, especially for large datasets. This performance is crucial when dealing with high volumes of data in real-world applications.

- **Enables Vectorized Operations**
  With NumPy, operations can be applied to entire arrays without using loops. This makes code shorter, easier to read, and computationally faster — a key advantage in data analysis and machine learning workflows.

- **Robust Mathematical Tools**
  NumPy provides a wide range of tools for linear algebra, statistics, random number generation, and more — all of which are essential for data science and scientific research.

- **Interoperability and Ecosystem Support**
  NumPy integrates seamlessly with other Python libraries used in data science, making it the glue that holds the Python scientific computing ecosystem together.

- **Cross-Disciplinary Applications**
  From physics to finance, machine learning to image processing, NumPy is used across domains wherever numerical computation is involved.

Introduction to Jupyter Notebook

- **Jupyter Notebook** is an open-source web-based interactive computing environment 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)
- It's widely used in **Data Science, Machine Learning, Education, and Research** because it supports interactive, step-by-step development and documentation.
- You can install jupyter extension from visual studio code

How to Install NumPy

- You can install NumPy using **pip**, which is the standard Python package manager.
- Open your terminal (Command Prompt, PowerShell, or terminal in VS Code or PyCharm).
- Run the following command:
  - pip install numpy
- To Verify Installation:
  - import numpy
  - print(numpy.__version__)
- This confirms that NumPy is successfully installed and tells you which version is active.

Array functions in numpy

Key Differences Between List and NumPy Array

Feature	Python List	NumPy Array
Data Type	Can hold mixed types (int, float, str)	Must have the same data type (int, float, etc.)
Performance	Slower (due to dynamic typing)	Faster (optimized for numerical operations)
Memory Usage	Takes more memory	Uses less memory (efficient storage)
Mathematical Operations	Need explicit loops (e.g., for loop)	Supports vectorized operations (faster calculations)
Built-in Methods	General-purpose functions (append(), sort(), etc.)	Specialized functions for math, stats, and linear algebra
Multi-Dimensional Support	Only 1D (nested lists for more)	Supports multi-dimensional arrays (e.g., 2D, 3D)
Flexibility	More flexible, stores mixed data types	Optimized for numerical computing

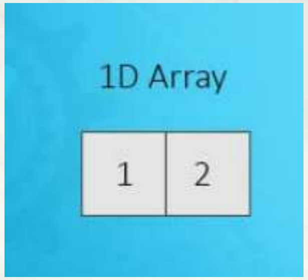
When to Use Each?

- 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.

array

- Use `np.array()` to convert a list to a NumPy array. which is a powerful data structure for numerical computing.
- **Creates Arrays from Lists or Tuples**:
  - Converts Python lists, tuples, or other sequences into a NumPy array, enabling efficient numerical operations.
- **Supports Multi-Dimensional Arrays**:
  - Creates 1D, 2D (matrices), or higher-dimensional arrays for complex data.
- Automatically converts elements to a common data type for consistency.
- **Code Example:**

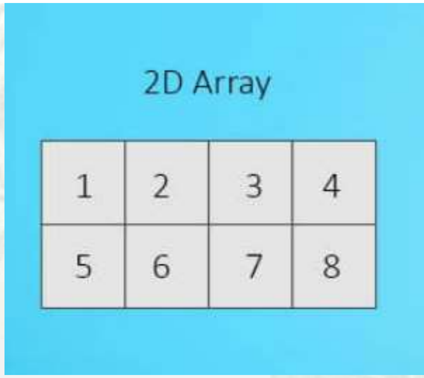
```
import numpy as np
arr = np.array([1, 2, 3])   #create 1d array
print(arr)                  # Output: [1 2 3]
b = np.array([1,2,3,4.5])
print(b)
```



2D array

- A 2D array in NumPy is a two-dimensional data structure,
- One should visualize it as a grid or matrix with rows and columns.
- It is used to represent tabular data, images, or mathematical matrices.
- Each element in a 2D array can be accessed using two indices: one for the row and one for the column.
- Like all NumPy arrays, 2D arrays are memory-efficient and support vectorized operations.
- **Where it is used?**
- Commonly used in data science, machine learning, and scientific computing for 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

NumPy zeros()

- The zeros() method creates a new array of given shape and type, filled with zeros.
- **numpy.zeros(shape, dtype = None, order = 'C')**
- The 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
- The zeros() method returns the array of given shape, order, and datatype filled with 0s.

```
import numpy as np
# create an array of 5 elements filled with 0s
array1 = np.zeros(5)
print(array1)  # Output: [0. 0. 0. 0. 0.]
```

NumPy ones()

- The ones() method creates a new array of given shape and type, filled with ones.
- **numpy.ones(shape, dtype = None, order = 'C')**
- he 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
- The ones() method returns the array of given shape, order, and datatype filled with 1s.

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

NumPy arange()

- The arange() method creates an array with evenly spaced elements as per the interval.
- **numpy.arange(start = 0, stop, step = 1, dtype = None)**
- The 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)
- The arange() method returns an array of evenly spaced values.

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

NumPy linspace()

- The linspace() method creates an array with evenly spaced elements over an interval.
- **numpy.linspace(start, stop, num = 50, endpoint = True, retstep = False, dtype = None, 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)
- The linspace() method returns an array of evenly spaced values.

Example of linespace function

```
import numpy as np
# create an array of 5 elements between 2.0 and 3.0
array1 = np.linspace(2.0, 3.0, num=5)
print(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
array2 = np.linspace(2.0, 3.0, num=5, endpoint=False)
print("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
array3, step_size = np.linspace(2.0, 3.0, num=5, retstep=True)
print("Array3:", array3) #Array3: [2.   2.25 2.5  2.75 3.  ]
print("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

NumPy copy()

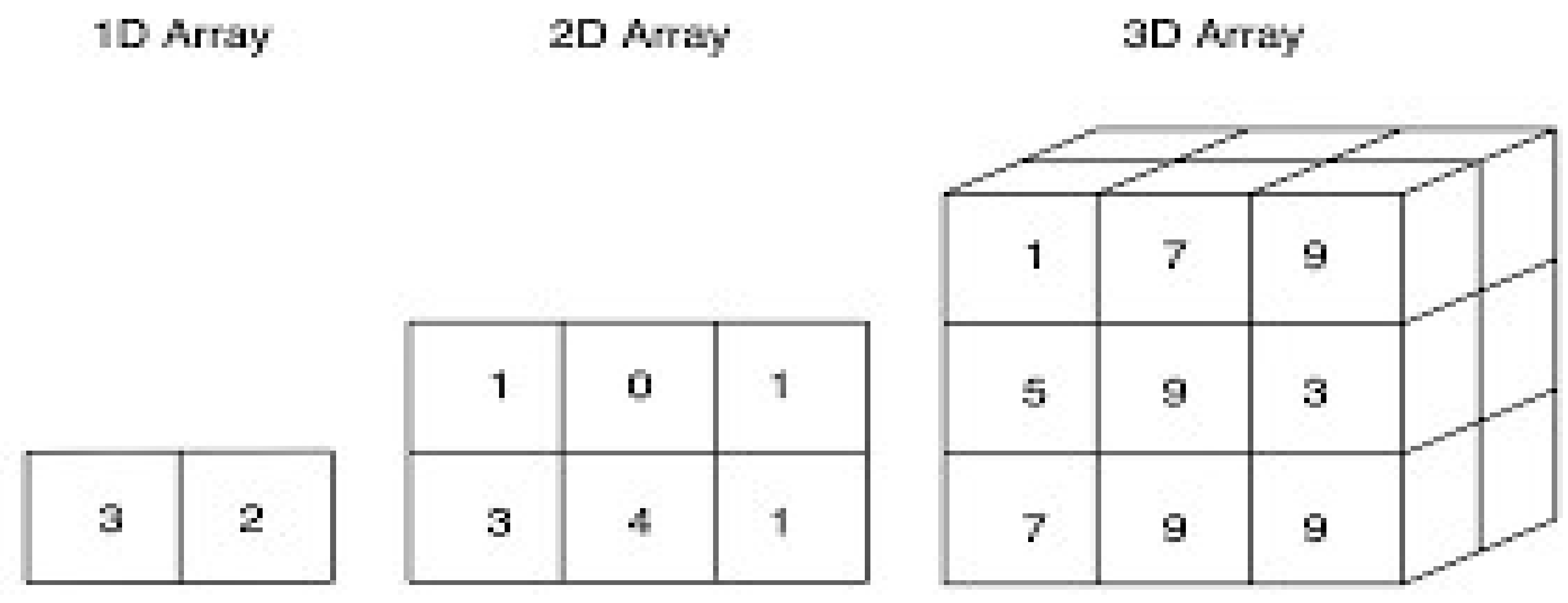
- The copy() method returns an array copy of the given object.
- **numpy.copy(array)**
- The copy() method takes has 1 argument:
  - array - input data
- The arange() method returns an array of evenly spaced values.

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



what are the array attributes?

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

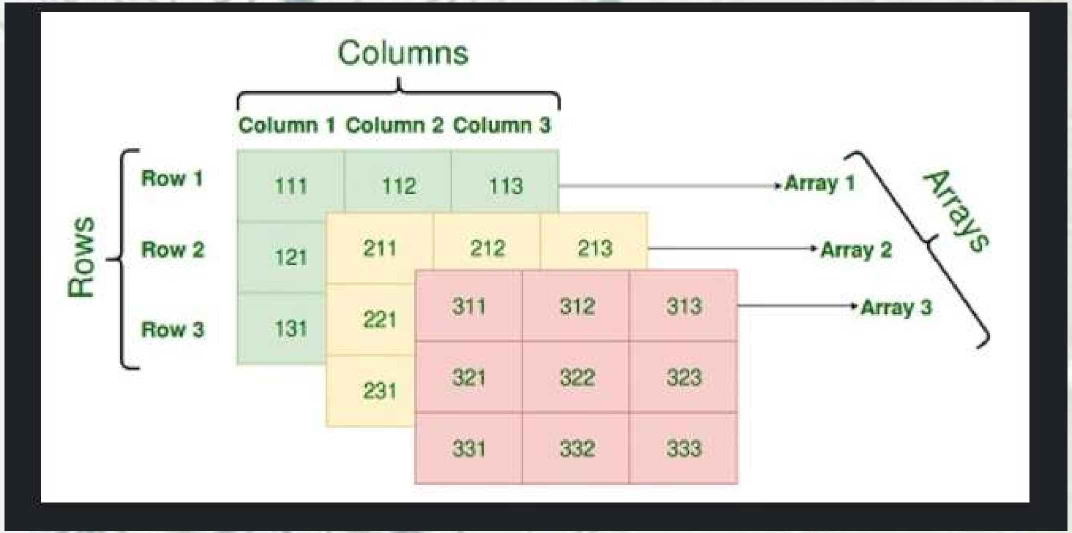
example

```
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)
```

How to Create a 3D Array?

```
import numpy as np
a = 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]]])
print(a)
```

```
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

Category	Examples	Description
Integer	int8, int16, int32, int64	Signed integers of different sizes
Unsigned Int	uint8, uint16, uint32, uint64	Non-negative integers
Float	float16, float32, float64	Real numbers (decimals)
Complex	complex64, complex128	Complex numbers (real + imaginary)
Boolean	bool_	True / False
String	str_, unicode_	Fixed-length strings
Object	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.

Important function in random module

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

Example of function in random module.

```
import numpy as np

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

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

```
import 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)
```

Accessing array

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

```
import numpy as np
arr = np.array([[10, 20, 30], [40, 50, 60]])
#Access element at 1st row, 2nd column:
print(arr[0, 1])  # Output: 20
# Slicing Arrays like list in python
print(arr[:, 1])    # All rows, 2nd column: [20 50]
print(arr[1, :])    # 2nd row, all columns: [40 50 60]
print(arr[0:2, 1:3])  # Subarray from row 0–1 and column 1–2
'''
[[20 30]
 [50 60]]
'''
#Boolean Indexing (can not use and and or operator)
print(arr[arr > 30])  # Output: [40 50 60]
#Fancy Indexing
print(arr[[0, 1], [1, 2]])  # Elements at (0,1) and (1,2): [20 60]
```

how to change value in array?

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

Arithmetic Operations (Element-wise)

- NumPy allows addition, subtraction, multiplication, and division to be performed directly between arrays of the same shape, applying the operation to each corresponding element.
- **Scalar Operations:**
  - You can perform operations between an array and a single number (scalar), and the operation is applied to every element in the array.
- **Mathematical Functions:**
  - NumPy provides built-in functions like np.sqrt(), np.exp(), np.log(), and np.sin() which work element-wise on arrays to perform common mathematical computations.
- **Comparison Operations:**
  - You can compare arrays using operators like >, <, ==, etc., and NumPy will return a boolean array showing the result for each element.
- **Aggregate Functions:**
  - Functions such as np.sum(), np.mean(), np.min(), and np.max() help you compute summaries like total, average, minimum, or maximum across all elements in an array.
- **Matrix Operations:**
  - For true matrix mathematics (not element-wise), you can use @ or np.dot() for matrix multiplication and .T to transpose arrays.

Let see example 1 of 2

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

Let see example 2 of 2

```
import numpy as np
a = np.array([10, 20, 30])
b = np.array([1, 2, 3])
#Comparison Operations
print(a > 15)   # [False  True  True]
# Matrix Operations (Not Element-wise)
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])
print(A @ B)        # Matrix multiplication
print(np.dot(A, B)) # Same result
# Aggregate Functions
arr = np.array([10, 20, 30, 40, 50])
print("Sum: ", np.sum(arr))       # Output: 150
print("Mean: ", np.mean(arr))     # Output: 30.0
print("Min: ", np.min(arr))       # Output: 10
print("Max: ", np.max(arr))       # Output: 50
```

Array Comparison in NumPy

- **Element-wise Comparison**:
  Using == or != compares each element of one array with the corresponding element in another array.
- **Result is a Boolean Array**:
  The output is a new array containing True or False for each comparison result.
- **a == b**:
  Returns True where elements of a and b are equal, False otherwise.
- **a != b**:
  Returns True where elements are different, False where they are the same.
- **np.all(condition)**:
  Returns True only if **all** elements in the condition are True.
- **np.any(condition)**:
  Returns True if **at least one** element in the condition is True.
- **Useful for Validation and Filtering**:
  These comparisons help in filtering arrays or checking conditions efficiently.

example

```python
import numpy as np
a = np.array([1, 2, 3])
b = np.array([1, 0, 3])
# Comparing Two Arrays (Element-wise)
print(a == b)      # [ True False  True]
print(a != b)      # [False  True False]
#Check if All or Any Elements Match
print(np.all(a == b))   # False (not all elements match)
print(np.any(a == b))   # True (at least one match)
```

array_equal()

- The array_equal() method in NumPy is used to check if two arrays have the same shape and elements, i.e.,
- it returns True if the two arrays are equal, and False otherwise.
- It compares both the data and the shape of the arrays.
- Syntax
- numpy.array_equal(a, b) #both a and b must be array
- Let see example

example

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

logical operations

- logical operations on NumPy arrays allow you to perform **element-wise** logical operations such as AND, OR, and NOT on boolean arrays or arrays containing numerical values.
- These operations are useful when filtering or selecting data based on conditions.
- **Logical 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.
- These operations return boolean arrays where each element is the result of applying the logical operation on corresponding elements of the input arrays.
- Let us see example

example

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