NumPy

What is NumPy?

NumPy (short form of **Numerical Python**) is an open-source Python library used for numerical computing. It provides tools for working efficiently with arrays, matrices, and mathematical functions.

NumPy introduces a new data structure called the **ndarray** (n-dimensional array), which is much faster and more memory-efficient than Python's built-in lists for numerical operations.

Important Features

- **Multidimensional Arrays:**
  Efficient storage and manipulation of large datasets (e.g., 1D, 2D, 3D arrays).
- **Mathematical Functions:**
  A wide range of mathematical operations — linear algebra, Fourier transforms, random number generation, etc.
- **Vectorization:**
  Operations on entire arrays without explicit loops, making code cleaner and faster.
- **Interoperability:**
  Works well with other libraries like Pandas, Matplotlib, TensorFlow, PyTorch etc.

Usage

```python
import numpy as np

# Create & Output an array
arr = np.array([1, 2, 3, 4, 5])
print(arr)
```

NumPy Arrays vs Python Lists

1. NumPy arrays are faster than Python lists (implemented in C, not pure Python).
2. They store elements in **contiguous memory** (better cache performance).
3. They are also homogenous i.e. all elements have same type.
4. They use **vectorized operations** (no slow Python loops).

Performance Comparison

```
import numpy as np
import time

size = 10_000_000 # large data set of 10 million numbers

# Python Lists
python_list = list(range(size))
start = time.time()

list_squared = [x**2 for x in python_list] # square of all nums

end = time.time()
print("Python list time:", end - start, "seconds")

# NumPy Arrays
np_array = np.array(python_list)
start = time.time()

array_squared = np_array ** 2 # vectorized operation
end = time.time()

print("NumPy array time:", end - start, "seconds")
```

Memory Usage Comparison

```
# Memory Usage comparison
import sys

print("Python list size:", sys.getsizeof(python_list) * len(python_list))
print("NumPy array size:", np_array.nbytes)
```

Creating NumPy Arrays

There are multiple ways of creating NumPy arrays, most common of which are:

1. From Python Lists

```
# Creating NumPy Arrays - from lists
arr = np.array([1, 2, 3, 4])
print(arr, type(arr))

arr2 = np.array([1, 2, 3, 4, "prime", 3.14])
print(arr2, type(arr2))

# 2D Arrays - Matrix
arr3 = np.array([[1, 2, 3], [4, 5, 6]])
print(arr3, arr3.shape)
```

Note - All elements in `arr2` in above code will have same type (homogenous) unlike lists.

2. Using built-in Functions

```
# Creating NumPy Arrays - from scratch
arr1 = np.zeros((3, 4)) # 3x4 array of 0s
print(arr1, arr1.shape)

arr2 = np.ones((3, 3)) # 3x3 array of 1s
print(arr2, arr2.shape)

arr3 = np.full((2, 3), 5) # 2x3 array of 5s
print(arr3, arr3.shape)

arr4 = np.eye(3) # Identity matrix of 3x3
print(arr4, arr4.shape)

arr5 = np.arange(1, 20, 2) # Elements in range(1, 20)
print(arr5, arr5.shape)

arr6 = np.linspace(0, 10, 5) # Evenly spaced array
print(arr6, arr6.shape)
```

NumPy Array Properties

Array properties helps you understand and manipulate data in arrays efficiently.

```
# Useful Attributes
arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]])

print(arr.shape) # Dimensions - (4 x 3)
print(arr.size)  # Total elements - (12)
print(arr.ndim)  # Number of dimensions - 2
print(arr.dtype) # Data type object - int64
print(arr.itemsize) # Size of each element in bytes - 8 for int64
```

We can also explicitly change the `dtype` for our arrays.

```
# Specify dtype at creation
str_arr = np.array([1, 2, 3], dtype="U")
print(str_arr, str_arr.dtype)

float_arr = np.array([1, 2, 3], dtype="float64")
print(str_arr, float_arr.dtype)

# Creating new array with a specific type from existing array
int_arr = float_arr.astype(np.int64)
print(int_arr, int_arr.dtype)
```

Operations on Arrays

There are a lot of useful operations that we can perform on our arrays.

1. Reshaping

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

reshaped = arr.reshape((2, 3)) # converts (1x6) => (2x3)
print(reshaped, reshaped.shape)

flattened = reshaped.flatten() # converts 2D => 1D
print(flattened, flattened.shape)
```

2. Indexing

```
# Indexing for 1D array
arr = np.array([1, 2, 3, 4, 5])
print(arr[0])

# Indexing for 2D array
arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]) # 2D array
print(arr[0][1]) # 2
print(arr[1][2]) # 6
```

Apart from simple indexing, we can also use Fancy & Boolean indexing.

Fancy indexing means accessing array elements using integer arrays/lists of indices rather than plain slices (`:`).

```
# Fancy Indexing
arr = np.array([1, 2, 3, 4, 5])
idx = [0, 1, 4]
print(arr[idx])        # print nums at given indices
```

Boolean masking(or indexing) means using a Boolean array (`True`/`False`) to select elements from another array.

```
# Boolean Indexing
print(arr[arr > 2])          # print nums greater than 2
print(arr[arr % 2 == 0])     # print even num
```

3. Slicing

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

print(arr[2:6]) # [3, 4, 5, 6]
print(arr[:6])  # [1, 2, 3, 4, 5, 6]
print(arr[3:])  # [4, 5, 6, 7]
print(arr[::2]) # [1, 3, 5, 7]
```

Copy v/s View

Slicing a list returns a copy but slicing a NumPy array returns a view - for efficiency. View is like a shallow copy that shares the same data as the original array, so no duplication happens here.

- **Views** are **fast** and **memory-efficient** (no data duplication).
- **Copies** are **safe** but **slower** and use **more memory**.

```
# Sliced List is a COPY
py_list = [1, 2, 3, 4, 5]
copy_list = py_list[1:4] # [2, 3, 4]
copy_list[1] = 333

print(copy_list)
print(py_list) # [1, 2, 3, 4, 5] - remains same

# Sliced Array is a VIEW
np_arr = np.array([1, 2, 3, 4, 5])
view_arr = np_arr[1:4] # [2, 3, 4]
view_arr[1] = 333

print(view_arr)
print(np_arr) # [1, 2, 333, 4, 5] - changes

# Creating a COPY for Array
copy_arr = np_arr[1:4].copy() # [2, 3, 4]
copy_arr[2] = 444
print(copy_arr)
print(np_arr) # [1, 2, 3, 4, 5] - remains same
```

Multi-dimensional Arrays

Multi-dimensional arrays in NumPy are the foundation of most scientific and machine-learning work.
A NumPy array can have **any number of dimensions** (1D, 2D, 3D & so on). Each dimension is called an **axis**.

- 1D array has 1 axis (axis0).
- 2D array has 2 axes (axis0 = rows, axis1 = columns)
- 3D array has 3 axes (axis0 = depth/layer, axis1 = rows in each layer, axis2 = columns in each layer)

```
# 1D array
arr1D = np.array([1, 2, 3])
print(arr1D.ndim) # 1

# 2D array (matrix)
arr2D = np.array([[1, 2, 3],
                  [4, 5, 6]])
print(arr2D.ndim) # 2

# 3D array (tensor)
arr3D = np.array([[[1, 2, 3],
                   [4, 5, 6]],
                  [[7, 8, 9],
                   [10, 11, 12]]])
print(arr3D.ndim) # 3
```

Usage of multi-dimensional arrays in Practice

We'll look into a practical example of dealing with real-world images data. If we have a ML/DL model working with images, then we can feed this data as 2D or 3D arrays.

1. **Grayscale Image** as 2D array
   A grayscale image has **height × width** pixels. Each pixel has a single intensity value (0–255 for 8-bit images).
2. **Color Images** as 3D array
   A color image(RGB) has **height × width × channels**. The 3 channels are → RGB (Red, Green, Blue) & each channel is a 2D array of pixel intensities (0-255).

We'll cover practical implementation in later chapters.

Operations along Axes

We can also perform certain **operations along a specific axis** in an array.

```
arr2D = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

print(np.sum(arr2D)) # sum of entire array - 45

sum_of_columns = np.sum(arr2D, axis = 0)
print(sum_of_columns) # [12 15 18]

sum_of_rows = np.sum(arr2D, axis = 1)
print(sum_of_rows) # [6 15 24]

# Slicing
print(arr2D[0:3, 1:3]) # slice rows(0, 1, 2) x cols(1, 2)
# [[2 3]
# [5 6]
# [8 9]]
```

Operations on 3D arrays

Let's look at how do we work with **3D** arrays:

```
arr3D = np.array([[[1, 2],[3, 4],[5, 6]], [[7, 8],[9, 10],[11, 12]]])

print(arr3D, arr3D.shape)

# Indexing
print(arr3D[0][1][1]) # 4
print(arr3D[1][2][1]) # 12

print(arr3D[:, :, 0])  # first col from both layers
print(arr3D[:, 0, :])  # first row from both layers

# Manipulating data
arr3D[:, 0, :] = 99    # change first row to store 99
print(arr3D)
```

Data Types in NumPy

We have already discussed how every NumPy array has a **single data type** (homogeneous arrays) & it is stored in the `.dtype` attribute.

Now, let's look into some of the most **common data types** in NumPy:

- Integer literals : `int32` , `int64`
- Floating literals : `float32` , `float64`
- Boolean : `bool`

- Complex numbers : `complex64` , `complex128`
- String : `S` (byte-str) & `U` (unicode-str)
- Object : generic python objects – `object`

```
# Common Data Types
arr = np.array([1, 2, 3, 4, 5])
arr2 = np.array([1.0, 2.0, 3.0])
arr3 = np.array(["hello", "world", "prime", "ai/ml"])

print(arr.dtype) # int64
print(arr2.dtype) # float64
print(arr3.dtype) # U

# Complex Numbers
arr1 = np.array([2 + 3j])
arr2 = np.array([5 + 8j])

print(arr1, arr1.dtype)
print(arr1 + arr2)
print(arr2 - arr1)

# Objects
arr = np.array (["hello", {1, 2, 3}, 3.14])
print(arr, arr.dtype)
```

We can also change the data type, either by explicitly typecasting at array creation using `dtype` attribute or by using the `astype` method while creating a new array from existing one.

```
# Changing the data type
new_arr = arr.astype("float64")
print(new_arr, new_arr.dtype)

new_arr = np.array([1, 2, 3, 4, 5], dtype="float64")
print(new_arr, new_arr.dtype)
```

Why does dtype matter?

- **Memory efficiency**
  - `np.int8` uses 1 byte per element, `np.int64` uses 8 bytes.
- **Performance**
  - Smaller types = faster computations.
- **Compatibility**
  - Images often use `np.uint8`
  - ML libraries expect `float32`

Note - In some case it is useful to do **Downcasting** i.e. converting type to a smaller data type. This is done to reduce memory usage & improve performance.

Example - Suppose you have a dataset of 1 million people's ages. Storing them as `int64` wastes memory because ages are small numbers (0–120). So we can downcast these values to `Int8`.

Vectorization & Broadcasting in NumPy

Vectorization and **Broadcasting** in NumPy are two of the most powerful features for fast numerical computations.

Vectorization

Vectorization means performing **operations on entire arrays at once** without explicit Python loops.

- NumPy uses **C-level implementations** internally → much faster than Python loops.
- Makes code **shorter, cleaner, and faster**.

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

sq_arr = arr ** 2        # Square of all nums
print(sq_arr)

arr2 = np.array([6, 7, 8, 9, 10])
print(arr + arr2)        # Sum of 2 arrays
```

Broadcasting

Broadcasting allows NumPy to **automatically expand arrays of different shapes** so that arithmetic operations can be performed. It's basically scaling arrays without using extra memory.

- No need to manually reshape arrays.
- Useful for **combining arrays of different dimensions**.

Broadcasting Rules

Broadcasting can only take place when the arrays are of compatible shape. So NumPy compares shapes of arrays **from right to left**. For the array to be compatible, all dimensions must either be:

- **Equal**, or
- **1**, or
- **Missing** (smaller array can be “stretched”).

```
# Broadcasting with a Scalar
arr_mul10 = arr * 10 # Multiply by 10 to all nums
print(arr_mul10)

# Broadcasting with a Vector
arr1D = np.array([1, 2, 3])
arr2D = np.array([[1, 2, 3], [4, 5, 6]])
print(arr1D + arr2D)
```

A quite common example of broadcasting in **Vector Normalization**. This is very common in machine learning and data preprocessing.

Let’s take an example of **Standard Vector Normalization** i.e. transforming an array such that it has:

- mean = 0
- standard deviation = 1

For each element x_i in vector, $x_i^{\text{normalized}} = (x_i - \mu)/\sigma$

Where:

- μ = mean of the vector (or column)
- σ = standard deviation

```
# Standard Vector Normalization
arr = np.array([[1, 2], [3, 4]])
mean = np.mean(arr)
std_dev = np.std(arr)
normalized_arr = (arr - mean) / std_dev

print(normalized_arr)

# Column wise Normalization
arr = np.array([[1, 2], [3, 4], [5, 6]])
mean = np.mean(arr, axis = 0)
std_dev = np.std(arr, axis = 0)
print((arr - mean) / std_dev)
```

Extra references: understand Standard Deviation and Variance.

Useful Mathematical Functions

NumPy is massive & provides a **wide range of built-in mathematical functions** that are highly optimized and can operate **element-wise on arrays**. Let's have a look at some of them:

Aggregation Functions

These are functions that take an array and reduce it to a single value (or smaller array) by combining elements.

1. `sum()` - returns sum of all elements
2. `prod()` - returns product of all elements
3. `min()` - returns minimum value
4. `max()` - returns maximum value
5. `argmin()` - returns index of min value
6. `argmax()` - returns index of max value
7. `mean()` - returns mean (average)
8. `median()` - returns median
9. `std()` - returns standard deviation
10. `var()` - returns variance

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

print(np.sum(arr))      # 15
print(np.prod(arr))     # 120
print(np.min(arr))      # 1
print(np.argmin(arr))   # 0
print(np.max(arr))      # 5
print(np.argmax(arr))   # 4
print(np.mean(arr))     # 3.0
print(np.median(arr))   # 3.0
print(np.std(arr))      # 1.41
print(np.var(arr))      # 2.0
```

Power Functions

1. `square()` - returns square of all elements
2. `sqrt()` - returns square root of all elements
3. `pow(a, b)` - returns a^b

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

print(np.square(arr))  # [1, 4, 9, 16, 25]
print(np.sqrt(arr))    # [1, 1.41, 1.73, 2, 2.23]
print(np.pow(arr, 3))  # [1, 8, 27, 64, 125]
```

Log & Exponential Functions

1. `log()` - returns natural log
2. `log10()` - returns log base 10
3. `log2()` - returns log base 2
4. `exp(x)` - returns e^x

```python
print(np.log(arr))
print(np.log10(arr))
print(np.log2(arr))
print(np.exp(arr))
```

Rounding Functions

1. `round()` - rounds off to nearest value
2. `ceil()` - rounds up
3. `floor()` - rounds down
4. `trunc(x)` - truncates(removes) the fractional part

```python
print(np.round(2.678)) # 3.0
print(np.floor(2.678)) # 2.0
print(np.ceil(2.678))  # 3.0
print(np.trunc(2.678)) # 2.0
```

Miscellaneous Functions

1. `abs()` - returns absolute value
2. `sort()` - returns sorted(arranges in increasing order) values
3. `unique()` - returns unique values

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

print(np.abs(arr))     # [1 2 5 3 8 4 2 5]
print(np.sort(arr))    # [-5 -4 1 2 2 3 5 8]
print(np.unique(arr))  # [-5 -4 1 2 3 5 8]
```

There are many other built-in functions that we will use and learn about in later chapters.

> *Keep Learning & Keep Exploring!*