growthpr7@gmail.com

NumPy Full Course Notes (Beginner to Advanced) - Youtube Link

1. Introduction & Setup (NumPy)

What is NumPy?

NumPy (Numerical Python) is a powerful open-source Python library primarily used for numerical and scientific computing. It provides fast, efficient array operations, making it the foundation of the data science and machine learning ecosystem.

Whether you're working with big data, complex mathematical models, or simply optimizing Python performance, NumPy is your go-to tool for array and matrix manipulation. It serves as the backbone for other popular libraries like Pandas, TensorFlow, Scikit-learn, and OpenCV.

Key Features of NumPy

- **High Performance:** Written in C, ensuring faster execution.
- **Multidimensional Array Object:** Called `ndarray`, with tools for complex array creation and manipulation.

- **Broadcasting:** Enables arithmetic on arrays of different shapes.
- **Mathematical Functions:** Over 50+ functions for stats, linear algebra, Fourier transform, etc.
- **Memory Efficient:** Operates on homogeneous data types to reduce memory usage.
- **Integration with Other Libraries:** Works seamlessly with Pandas, Matplotlib, and more.

Installation & Setup

You can install NumPy using pip or conda:

Using pip:

```
pip install numpy
```

Using Anaconda:

```
conda install numpy
```

Import NumPy in Python:

```
import numpy as np
```

Here, np is the standard alias used throughout the Python ecosystem.

Why Use NumPy Over Lists?

Python lists are flexible but slow for numerical computation. NumPy arrays are:

- **Faster** due to vectorized operations

- **More compact** in memory usage
- **Convenient** with built-in mathematical functions

Example 1: Speed Difference

```
import numpy as np
import time
arr = np.arange(1000000)
start = time.time()
arr * 2
print("NumPy Time:", time.time() - start)

lst = list(range(1000000))
start = time.time()
[x * 2 for x in lst]
print("List Time:", time.time() - start)
```

Example 2: Memory Efficiency

```
import numpy as np
import sys

arr = np.array([1, 2, 3])
lst = [1, 2, 3]
print("NumPy array size:", arr.itemsize * arr.size)
print("List size:", sys.getsizeof(lst))
```

Real-World Applications of NumPy

- **Data Science & Analysis**: NumPy provides the base layer for processing and analyzing datasets.
- **Machine Learning**: Libraries like Scikit-learn and TensorFlow are built on top of NumPy.
- **Computer Vision**: OpenCV uses NumPy arrays for pixel-level image data.

- **Finance & Forecasting**: Ideal for working with time-series and numerical prediction models.

Feature	NumPy	Python List
Speed	Fast	Slower
Memory Usage	Lower	Higher
Broadcasting Support	Yes	No
Built-in Math Ops	Extensive	Minimal
Multi-dimensional	Supports (2D, 3D, etc.)	Only 1D via nesting

2. Creating Arrays in NumPy

What is an Array in NumPy?

An array in NumPy is a grid of values (all of the same type), indexed by a tuple of nonnegative integers. Arrays are the building blocks of NumPy, used to store data in a structured and efficient manner.

Arrays in NumPy are known as `ndarray` objects and can be 1D, 2D, or multi-dimensional. Creating arrays efficiently is the first step to mastering data handling with NumPy.

Methods to Create Arrays

1. From Python Lists or Tuples

Convert a standard list or tuple into a NumPy array.

```
import numpy as np
arr1 = np.array([10, 20, 30])
print(arr1)

tuple_array = np.array((5, 10, 15))
print(tuple_array)
```

2. Using Predefined Functions

NumPy provides powerful methods for generating arrays automatically.

a. `np.zeros()` – Creates an array filled with zeros

```
zeros_arr = np.zeros((2, 3))
print(zeros_arr)
```

b. `np.ones()` – Creates an array filled with ones

```
ones_arr = np.ones((3, 2))
print(ones_arr)
```

c. `np.full()` – Creates a constant value array

```
filled_arr = np.full((2, 2), 7)
print(filled_arr)
```

3. Generating Ranges

Use functions that create sequences of numbers:

a. `np.arange(start, stop, step)`

```
range_arr = np.arange(0, 10, 2)
print(range_arr)
```

b. `np.linspace(start, stop, num)` – Evenly spaced numbers

```
space_arr = np.linspace(0, 1, 5)
print(space_arr)
```

Use `dtype` argument to specify data type while creating an array:

```
np.array([1, 2, 3], dtype='float32')
```

- Use `ndmin` to specify minimum number of dimensions:

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

3. Array Indexing and Slicing

What is Indexing and Slicing in NumPy?

Indexing refers to accessing individual elements, while slicing allows us to extract subarrays. NumPy provides powerful and flexible ways to extract values from arrays, making data manipulation easier and faster.

You can access values using indices like in Python lists, but with more capabilities — especially in multi-dimensional arrays.

Indexing Examples

1. Basic Indexing

Accessing elements by their position:

```
import numpy as np
arr = np.array([10, 20, 30, 40, 50])
print(arr[2])  # Output: 30
```

2. 2D Array Indexing

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

Note: `matrix[row, column]`

Slicing Examples

✅ **1. 1D Array Slicing**

```
arr = np.array([10, 20, 30, 40, 50])
print(arr[1:4])  # Output: [20 30 40]
```

✅ **2. 2D Array Slicing**

```
matrix = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(matrix[:2, 1:])
# Output:
# [[2 3]
#  [5 6]]
```

Boolean Indexing

Select elements that satisfy a condition.

```
arr = np.array([5, 10, 15, 20])
print(arr[arr > 10])  # Output: [15 20]
```

Fancy Indexing

Use arrays of indices to access multiple elements.

```
arr = np.array([100, 200, 300, 400, 500])
print(arr[[1, 3, 4]])  # Output: [200 400 500]
```

4. Array Shape and Reshape

What is Shape in NumPy?

The shape of a NumPy array refers to the number of elements along each dimension (rows, columns, etc.). It's a tuple that tells you how your data is structured in memory.

What is Reshaping?
Reshaping means changing the structure of an array without changing its data. For example, converting a 1D array to a 2D matrix.

📏 Checking the Shape of Arrays

```
import numpy as np
arr = np.array([[1, 2, 3], [4, 5, 6]])
print(arr.shape)  # Output: (2, 3)

arr1D = np.array([1, 2, 3, 4])
print(arr1D.shape)  # Output: (4,)
```

Reshaping Arrays

1. Using `reshape()`

```
arr = np.arange(6)
reshaped = arr.reshape(2, 3)
print(reshaped)
```

Output:

```
[[0 1 2]
 [3 4 5]]
```

2. Converting Multi-Dimensional to 1D

```
matrix = np.array([[1, 2], [3, 4]])
flat = matrix.reshape(-1)
print(flat)  # Output: [1 2 3 4]
```

Transpose Arrays

Swap rows and columns using `.T`

```
matrix = np.array([[1, 2], [3, 4]])
print(matrix.T)
```

Output:

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

5. Mathematical Operations

What Are Mathematical Operations in NumPy?
NumPy enables fast, element-wise operations on entire arrays using simple syntax. These operations are vectorized, meaning they're much faster than looping through individual elements.

You can perform addition, subtraction, multiplication, division, power, and use a wide variety of built-in universal functions (ufuncs) such as `np.sqrt()`, `np.exp()`, `np.log()` etc.

Basic Arithmetic Operations

All arithmetic operations work element-wise:

```
import numpy as np
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])

print(arr1 + arr2)   # [5 7 9]
print(arr1 - arr2)   # [-3 -3 -3]
print(arr1 * arr2)   # [4 10 18]
print(arr2 / arr1)   # [4.0 2.5 2.0]
```

Mathematical Functions (ufuncs)

1. Square Root

```
arr = np.array([4, 9, 16])
print(np.sqrt(arr))  # [2. 3. 4.]
```

2. Exponentiation

```
arr = np.array([1, 2, 3])
print(np.exp(arr))  # [ 2.718  7.389  20.085 ]
```

3. Logarithm

```
arr = np.array([1, np.e, np.e**2])
print(np.log(arr))  # [0. 1. 2.]
```

Aggregate Functions

Useful for summarizing data:

```
arr = np.array([1, 2, 3, 4])
print(arr.sum())      # 10
print(arr.mean())     # 2.5
print(arr.min())      # 1
print(arr.max())      # 4
print(arr.std())      # Standard deviation
```

6. Broadcasting

What is Broadcasting in NumPy?

Broadcasting is a powerful feature that allows NumPy to perform arithmetic operations on arrays of different shapes and sizes. Instead of manually reshaping arrays to match, broadcasting automatically expands them to a compatible shape.

This is especially useful in data science and machine learning when applying operations across rows, columns, or vectors without writing loops.

Broadcasting Rules

Broadcasting compares array shapes element-wise from the end:

1. If dimensions match or one of them is 1, they are compatible.

2. If they don't match and neither is 1, an error is raised.

Example 1: Add Scalar to Array

```
import numpy as np
arr = np.array([10, 20, 30])
print(arr + 5)
```

Output: `[15 25 35]`

Explanation: The scalar `5` is broadcast to each element.

Example 2: Add 1D Array to 2D Array

```
matrix = np.array([[1, 2, 3], [4, 5, 6]])
row = np.array([10, 20, 30])
print(matrix + row)
```

Output:

```
[[11 22 33]
 [14 25 36]]
```

Explanation: The row vector `[10, 20, 30]` is broadcast to each row of the matrix.

Example 3: Broadcasting with Columns

```
col = np.array([[10], [20]])
print(matrix + col)
```

Output:

```
[[11 12 13]
 [24 25 26]]
```

Explanation: Column vector is broadcast to each column.

7. Aggregate Functions

What Are Aggregate Functions in NumPy?

Aggregate functions perform summary operations across elements in an array. They are useful for computing statistics such as sum, average, min, max, and standard deviation.

These operations can be applied to entire arrays or along specific axes (rows or columns).

Common Aggregate Functions

1. Sum and Mean

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
print(arr.sum())   # Output: 15
print(arr.mean())  # Output: 3.0
```

2. Min, Max, and Standard Deviation

```
print(arr.min())   # Output: 1
print(arr.max())   # Output: 5
print(arr.std())   # Output: 1.4142...
```

Aggregation on Multi-Dimensional Arrays

You can perform aggregation row-wise or column-wise using the `axis` parameter:

1. Column-wise (axis=0)

```
matrix = np.array([[1, 2], [3, 4]])
print(matrix.sum(axis=0))  # Output: [4 6]
```

2. Row-wise (axis=1)

```
print(matrix.mean(axis=1))  # Output: [1.5 3.5]
```

8. Array Joining and Splitting

What is Array Joining?

Joining is the process of combining two or more arrays into a single array. NumPy provides several functions like `concatenate()`, `stack()`, `vstack()`, and `hstack()` for this purpose.

What is Array Splitting?

Splitting is the opposite of joining — it involves breaking a single array into multiple smaller arrays using functions like `split()`, `hsplit()`, and `vsplit()`.

Joining Arrays

1. Using `np.concatenate()`

```
import numpy as np
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
joined = np.concatenate((arr1, arr2))
print(joined)  # Output: [1 2 3 4 5 6]
```

2. Vertical and Horizontal Stacking

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

print(np.vstack((a, b)))
```

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

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

Splitting Arrays

1. Using `np.split()`

```
arr = np.array([10, 20, 30, 40, 50, 60])
split_arr = np.split(arr, 3)
print(split_arr)
```

Output:

```
[array([10, 20]), array([30, 40]), array([50, 60])]
```

2. Horizontal and Vertical Splits

```
mat = np.array([[1, 2, 3], [4, 5, 6]])
print(np.hsplit(mat, 3))  # Split columns
print(np.vsplit(mat, 2))  # Split rows
```

9. Boolean Indexing

What is Boolean Indexing in NumPy?

Boolean indexing is a powerful way to filter and select elements from an array using boolean conditions. Instead of manually looping through elements, NumPy allows you to directly extract elements that satisfy a specific condition.

Example 1: Basic Boolean Filtering

```
import numpy as np
arr = np.array([10, 20, 30, 40, 50])
filtered = arr[arr > 25]
print(filtered)  # Output: [30 40 50]
```

Example 2: Combining Conditions

You can use logical operators like &, |, and ~.

```
arr = np.array([5, 10, 15, 20, 25])
print(arr[(arr > 10) & (arr < 25)])  # Output: [15 20]
```

Example 3: Boolean Masks

You can create a mask array of booleans and use it to filter:

```
mask = arr > 10
print(mask)          # Output: [False False  True  True  True]
print(arr[mask])      # Output: [15 20 25]
```

10. Real Life Use Case Example

Use Case: Curving Student Grades
Let's say we want to curve student grades so that the class average becomes 80, but no student gets a score lower than their original or higher than 100. NumPy makes this operation simple and efficient.

Step-by-Step Solution

Step 1: Define Grades

```
import numpy as np
grades = np.array([72, 35, 64, 88, 51])
```

Step 2: Calculate the Mean and Curve

```
avg = grades.mean()
adjusted_grades = grades + (80 - avg)
```

Step 3: Clip the Values

We want to ensure no grades go below the original or above 100.

```
final_grades = np.clip(adjusted_grades, grades, 100)
print(final_grades)
```

Output: Adjusted and bounded grades as per our requirement.

Why It Works

- `grades.mean()` finds the average.
- `np.clip()` limits the adjusted values within a specified range.
- Entire array operation is vectorized—no loops needed!

Real-World Relevance

- Adjusting employee scores in performance reviews
- Scaling exam marks or feedback scores
- Simulating bonus distributions or inflation-adjusted values