# **Learn Numpy & Pandas**

# What is Numpy?

NumPy (Numerical Python) is a powerful library in Python for numerical computing. It provides support for working with large, multi-dimensional arrays and matrices, along with mathematical functions to operate on these data structures efficiently.

**Key Features of NumPy:**

* **N-dimensional array:** Supports efficient storage and manipulation of multi-dimensional arrays.
* **Mathematical functions:** Includes operations like linear algebra, Fourier transform and statistical computations
* **Broadcasting:** Enables performing operations on arrays of different shapes seamlessly
* **Performance Optimization:** Written in C, making computations significantly faster than Python’s built-in list operations
* **Integration:** Works well with other scientific computing libraries like Pandas, SciPy, and Matplotlib.

**Why use NumPy?**

**Speed:** Much faster compared to Python lists when dealing with numerical data.

**Memory Efficiency:** Uses less memory than Python lists for storing data.

**Ease of Computation:** Allows vectorized operations, eliminating the need for explicit loops.

**Interoperability:** Compatible with various data formats and scientific computing frameworks.

## **Installation of NumPy**

Installing NumPy is straightforward and can be done using pip or conda, depending on your preference. Here’s a step-by-step guide for Windows, Linux, and macOS:

### **Windows Installation:**

1. Open Command Prompt
2. Check Python Installation

python – version

If Python is not installed, download and install it from python.org

1. Install NumPy using pip

pip install numpy

1. Verify Installation

Python -c “import numpy; print(numpy.\_\_version\_\_)”

### **Linux Installation**

1. Open Terminal
2. Check Python Installation

python3 –version

If Python is missing, install it using

sudo apt install python3

1. Install pip (if not installed)

sudo apt install python3-pip

1. Install Numpy

pip3 install numpy

1. Verify Installation

Python3 -c “import numpy; print(numpy.\_\_version\_\_)”

### **macOS**

1. Open Terminal
2. Check Python Installation

python3 – version

If Python is missing, install it using

brew install python

1. Install pip (if not installed)

python3 -m ensurepip

1. Install Numpy

pip install numpy

1. Verify installation

python3 -c “import numpy; print(numpy.\_\_version\_\_)”

Alternatively, you can install NumPy using conda

conda install numpy

This method is useful if you are using Anaconda or Miniconda

## **Creating and Manipulating Arrays**

### **1D Array (One-Dimensional)**

**A 1D Array is simply a list-like array with single dimension.**

**Creation:**

import numpy as np

# Creating a 1D array

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

print(arr\_id)

**Manipulation of Array:**

**Access Elements:** arr\_id[2]

**Slicing:** arr\_id[1:3]

**Mathematical Operations:** arr\_id \* 2

### **2D Array (Two-Dimensional)**

A 2D Array represents rows and column, like a table or matrix

**Creation:**

arr\_2d = np.array([[1,2,3],[4,5,6]])

print(arr\_2d)

**Manipulation of Array:**

**Access Elements:** arr\_2d[1,2]

**Row slicing:** arr\_2d[0,:]

**Column slicing:** arr\_2d[:,-1]

**Matrix Operations:** np.dot(arr\_2d, np.array([[1], [2], [3]]))

### **3D Array (Three-Dimensional)**

A **3D array** adds depth, like multiple 2D arrays stacked.

**Creation:**

python

arr\_3d = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])

print(arr\_3d)

**Manipulation:**

* **Access elements:** arr\_3d[1, 0, 1] → Retrieves 6.
* **Slicing:** arr\_3d[:, :, 0] → Extracts all first elements from each depth layer.
* **Reshape:** arr\_3d.reshape(4, 2) → Changes shape while preserving data.

### **4D Array (Four-Dimensional)**

A **4D array** is useful for representing complex datasets like images or time-series data.

**Creation:**

python

arr\_4d = np.random.randint(1, 10, (2, 2, 2, 2)) # Creates a random 4D array

print(arr\_4d)

**Manipulation:**

* **Access elements:** arr\_4d[1, 0, 1, 0] → Retrieves a specific number.
* **Transpose:** arr\_4d.transpose(1, 0, 2, 3) → Rearranges dimensions.
* **Reshape:** arr\_4d.reshape(8, 2) → Flattens some dimensions while keeping structure.

**Summary**

* **1D**: Simple list-like array.
* **2D**: Table/matrix-like format.
* **3D**: Stack of 2D arrays.
* **4D**: Complex structured data, useful for advanced computing.

## **Difference between NumPy Array and Python List**

NumPy arrays and Python Lists might seem similar at first glance, but they have key differences:

**Speed & Performance**

* NumPy arrays are significantly faster than Python lists because they use contiguous memory allocation and vectorized operations.
* Python lists store references to objects, making them slower when performing numerical computations
* print ("""
* ######################################################################
* #####     Speed Comparison between NumPy vs Python Lists         #####
* ######################################################################
* """)
* import numpy as np
* import time
* # Creating large list and NumPy array
* list\_data = list(range(1000000))
* numpy\_array = np.arange(1000000)
* # Timing list operations
* start = time.time()
* list\_result = [x \* 2 for x in list\_data] # using list comprehension
* end = time.time()
* print(f"List computation time: {end - start:.5f} seconds")
* # Timing NumPy array operations
* start = time.time()
* numpy\_result = numpy\_array \* 2 # vectorized operation
* end = time.time()
* print(f"NumPy computation time: {end - start: .5f} seconds")
* print ("+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++")

**Memory Efficiency**

* NumPy arrays consume less memory than lists because they store elements as a homogenous type (all elements must be of the same data type)
* Lists, on the other hand, store elements as individual objects, leading to higher memory usage
* print("""
* ######################################################################
* #####  Memory utilization Comparison between NumPy vs Python List ####
* ######################################################################
* """)
* import numpy as np
* import sys
* # Creating a Python list and a NumPy array with the same elements
* python\_list = [i for i in range(1000)]
* numpy\_array = np.arange(1000)
* # Checking memory usage
* python\_list\_size = sys.getsizeof(python\_list) + sum(sys.getsizeof(i) for i in python\_list)
* numpy\_array\_size = numpy\_array.nbytes
* print(f"Memory used by Python list: {python\_list\_size} bytes")
* print(f"Memory used by NumPy array: {numpy\_array\_size} bytes")
* print ("+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++")

**Functionality & Operations**

* NumPy supports element-wise operations, meaning you can perform arithmetic operations directly on arrays.
* Python lists require explicit looping or list comprehensions to achieve similar operations.

**Type Consistency**

* NumPy arrays enforce a single data type (e.g., all elements must be integers or floats)
* Lists can hold mixed data types, such as integers, strings, or even other lists

**Built-in Methods**

* NumPy provides numerous powerful mathematical and statistical functions
* List rely on Python’s built-in methods (append(), sort(), remove() etc.,) which are more general-purpose

*Bottom Line:* If you're working with numerical data or large datasets, NumPy arrays are the way to go for efficiency and speed. If you need flexibility and mixed data types, Python lists might be better.

## **NumPy Arrays**

### **Creating NumPy arrays**

NumPy provides several ways to create arrays, each suited to different use cases.

#### **np.array**

This is the most basic way to create a NumPy array using Python list or tuples

Import numpy as np

# Creating a NumPy array from a Python list

list\_data = [1,2,3,4]

numpy\_array = np.array(list\_data)

print(numpy\_array)

print(type(numpy\_array))

* np.array() converts a list or tuple into NumPy array
* It automatically determines the data type (dtype) of elements

#### **np.arange**

This function creates arrays with evenly spaced values

# Creating an array with values from 0 to 9

numpy\_array2 = np.arange(10)

print(numpy\_array2)

# Creating an array with a specific step

numpy\_array3 = np.arange(1,10,2)

print(numpy\_array3)

#### **np.linspace**

Generates a specified number of evenly spaced values between a given range.

# Creating an array from 0 to 10 with 5 elements

numpy\_array4 = np.linspace(0,10, 5)

print(numpy\_array4) # Output: [0. 2.5 5. 7.5 10.]

* np.linspace(start, stop, num\_elements) ensures precision when dividing a range into equal parts
* Often used in plotting or mathematical computations

#### **np.zero() and np.ones()**

Create arrays filled with zeros or ones

# Creating a 1D array with five zeros

zeros\_array = np.zeros(5)

print(zeros\_array) # Output: [ 0. 0. 0. 0. 0. ]

# Creating a 2x3 matrix filled with ones

ones\_array = np.ones((2,3))

print (ones\_array)

output:

[[1. 1. 1.]

[1. 1. 1.]]

* np.zeros(shape) creates an array filled with zeros
* np.ones(shape) creates an array filled with ones

#### **np.full()**

Creates an array filled with specific values

# Creating a 3 x 3 array filled with value 7

full\_array = np.full((3,3),7)

print (full\_array)

output:

[[7 7 7]

[7 7 7]

[7 7 7]]

* np.full(shape, value) creates an array with a predefined value
* handy for defining constant matrices

#### **np.eye()**

Generates an identity matrix

# Create a 4 x 4 identity matrix

identity\_matrix = np.eye(4)

print(identity\_matrix)

output:

[[1. 0. 0. 0.]

[0. 1. 0. 0.]

[0. 0. 1. 0.]

[0. 0. 0. 1.]]

* Identity matrices are often used in linear algebra operations

#### **np.random()**

Creates arrays filled with random values

# Creating a 3 x 3 array with random numbers between 0 and 1

random\_array = np.random(3,3)

print (random\_array)

Output:

[[0.342 0.892 0.102]

[0.423 0.015 0.923]

[0.712 0.831 0.531]]

* np.random.rand(shape) generates uniform random numbers between 0 and 1
* np.random.randint(low, high, shape) generates random integers

#### **np.empty()**

Creates an uninitialized array (contains garbage values)

# Creating an empty array

empty\_array = np.empty((2,3))

print(empty\_array)

'''

Output (values may vary):

[[4.66706551e-310 2.14156671e-316 0.00000000e+000]

[0.00000000e+000 5.02034658e+175 5.56417452e-309]]

'''

* np.empty(shape) allocates memory but does not initialize values.
* Faster than np.zeros() and np.ones() when initial values are irrelevant.

### **Understanding array data types**

dtype

### **Changing array shape**

Reshape

Flatten

Ravel

### **Indexing and slicing arrays**

Copying vs View (memory optimization)