# **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:**

1. **Access elements:** arr\_4d[1, 0, 1, 0] → Retrieves a specific number.
2. **Transpose:** arr\_4d.transpose(1, 0, 2, 3) → Rearranges dimensions.
3. **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.rand(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**

NumPy arrays are homogenous, meaning all elements must have the same data type. The dtype attribute in NumPy defines the type of elements stored in an array, ensuring efficient memory usage and optimized computations

#### **Checking the Data Type of an Array**

You can check the data type of a NumPy array using .dtype

import numpy as np

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

print(arr.dtype) # Output: int64

The .dtype attribute returns the data type of the array elements

#### **Specifying Data Type while creating an array**

You can explicitly define the data type when creating an array

arr\_float = np.array([1,2,3], dtype=np.float32)

print (arr\_float) # Output: float32

#### **Common NumPy Data Types**

|  |  |
| --- | --- |
| **Data Type** | **Description** |
| int8, int16, int32, int64 | Integer types of different sizes |
| float16, float32, float64 | Floating-point numbers |
| bool\_ | Boolean values (True or False) |
| complex64, complex128 | Complex numbers |
| str\_, unicode\_ | String data types |

#### **Changing Data Type**

You can convert an array to a different data type using .astype()

arr = np.array([1.5,2.7,5.6])

arr\_int = arr.astype(np.int32)

print(arr\_int) # Output: [1,2,3]

print (arr\_int.dtype) # Output: int32

This converts floating point to integers, truncating the decimal part.

#### **Memory Efficiency with NumPy Data Types**

NumPy optimizes memory usage compared to Python lists. For example:

import sys

py\_list = [1, 2, 3, 4, 5]

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

print(sys.getsizeof(py\_list)) # Memory used by Python list

print(np\_array.nbytes) # Memory used by NumPy array

NumPy arrays are **more memory-efficient** than Python lists

#### **Structured Data Types**

NumPy allows defining **structured arrays** with multiple fields.

dt = np.dtype([('name', np.str\_, 16), ('age', np.int32)])

arr = np.array([('Alice', 25), ('Bob', 30)], dtype=dt)

print(arr['name']) # Output: ['Alice' 'Bob']

print(arr['age']) # Output: [25 30]

This is useful for handling tabular data

### **Changing array shape**

NumPy provides several ways to reshape arrays, allowing you to modify their dimensions without changing the underlying data. This is useful for organizing data efficiently in scientific computing and machine learning.

#### **.reshape()**

The .reshape() method allows you to change the shape of any array while maintaining the same number of elements.

import numpy as np

# Creating a 1D Array

arr = np.arange(12) Output: [0,1,2,3,4,5,6,7,8,9,10,11]

# Reshaping into 3 x 4 matrix

reshaped\_arr = arr.reshape(3,4)

print (reshaped\_arr)

Output:

[[ 0 1 2 3]

[ 4 5 6 7]

[ 8 9 10 11]]

* The total number of elements **must remain the same** (12 elements in both cases).
* .reshape(new\_shape) does **not** modify the original array; it returns a new view.

#### **.ravel() and .flatten()**

These methods convert mult-dimensional array into 1D arrays

# Flattening the reshaped array

flat\_arr = reshaped\_arr.ravel()

print(flat\_arr) # Output: [0 1 2 3 4 5 6 7 8 9 10 11]

* .ravel() returns a **view** (changes affect the original array).
* .flatten() returns a **copy** (changes do not affect the original array).

#### **.transpose()**

Swaps rows and column in a matrix

transposed\_arr = reshaped\_arr.transpose()

print(transposed\_arr)

Output:

[[ 0 4 8]

[ 1 5 9]

[ 2 6 10]

[ 3 7 11]]

* .transpose() flips the shape ((3,4) → (4,3)).

#### **.resize()**

Unlike .reshape(), .resize() modifies the original array

arr.resize(4,3)

print(arr)

Output:

[[ 0 1 2]

[ 3 4 5]

[ 6 7 8]

[ 9 10 11]]

* .resize(new\_shape) **changes the array in place**.
* If the new shape is larger, NumPy fills extra spaces with 0.

#### Using -1 for Automatic Reshaping

NumPy can infer one dimension automatically

When you specify -1 as a dimension, NumPy automatically calculates that dimension based on the total number of elements

It ensures that the total number of elements remains the same after reshaping

arr = np.arange(12)

# Reshaping into 2D with 6 columns

auto\_reshaped = arr.reshape(-1,6)

print(auto\_reshaped)

Output:

[[ 0 1 2 3 4 5]

[ 6 7 8 9 10 11]]

Here, NumPy automatically determines that **2 rows** are needed (12 / 6 = 2).

arr = np.arange(20) # 20 elements

reshaped\_array = arr.reshape(4, -1) # NumPy figures out the missing dimension

print(reshaped\_array.shape) # Output: (4, 5)

* Since arr has **20 elements**, and we specify 4 rows, NumPy infers 5 columns.

### **Indexing and slicing arrays**

NumPy provides powerful tools for indexing and slicing arrays, allowing efficient manipulation of data.

#### **Indexing in NumPy**

Indexing refers to accessing elements of an array using their position

##### **1D Array Indexing**

import numpy as np

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

print (arr[0]) # Output: 1

print (arr[-1]) #Output: 5

##### **2D Array Indexing**

import numpy as np

arr2d = np.array([[1,2,3],[4,5,6],[7,8,9]])

print (arr2d[0,1]) # Element at row index 0 and column index 1: Output: 2

print (arr2d[2,2]) # Element at row index 2 and column index 2:Output: 9

##### **Boolean Indexing**

arr =np.array([10,20,30,40,50,60])

print(arr[arr>25]) # Returns elements greater than 25: [30,40,50,60]

### **Slicing in NumPy**

Slicing allows extracting portions of an array

#### **1D Array Slicing**

arr = np.array([10,20,30,40,50,60,70,80])

print(arr[1:4]) # Elements from index to 3 Output: [20,30,40]

print(arr[:3]) # First three elements Output: [10,20,30]

print (arr[::2]) # Every second element Output: [10,30,50,70]

#### **2D Array Slicing**

arr2d = np.array([[1,2,3], [4,5,6], [7,8,9])

print(arr2d[:2, :2]) # First two rows and first two columns Output:[ [1,2],[4,5],[7,8]]

print(arr2d[:,1]): # All rows second column Output: [[2 ],[5],[6]]

print(arr2d[1:,:]) # All rows from first row and all columns Output: [[4,5,6],[7,8,9]]

#### **Step Slicing**

arr = np.array([10,20,30,40,50,60])

print(arr[1:5:2]) # Elements from index 1 to 4 with step 2: Output: [20,40]

#### **Modifying Slices**

Since NumPy Slices are views, modifying them affects the original array.

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

slice\_arr = arr[1:4]

slice\_arr[:] = 99 # Modifies original array

print(arr) #Output: [1,99,99,99,5]

To avoid modifying the original array, use .copy()

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

slice\_arr = arr[1:4].copy()

slice\_arr = 99

print(arr) # Output: [1,2,3,4,5]

#### **Fancy Indexing**

Fancy Indexing allows selecting multiple elements at once.

arr = np.array([10,20,30,40,50])

indices = [0,2,4]

print(arr[indices]) # Output: [10,30,50]

arr2d = np.array([[1,2,3],[4,5,6],[7,8,9]])

rows = [0,2]

cols = [1,2]

print(arr2d[rows, cols]) # Output: [2,9]

### **Copying vs View (memory optimization)**

In NumPy, copying and viewing refer to different was of handling array data:

#### **Copy**

A copy creates a new array with duplicated data

Changes made to the copy do not affect the original array

Copies are memory-intensive but necessary when independent modifications are required

You can explicitly create a copy using ndarray.copy()

Example:

import numpy as np

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

x = arr.copy()

arr[0] = 100

print(x) # Output: [1,2,3,4,5]

print(arr) # Output: [100,2,3,4,5]

#### **View**

A view provides a different perspective on the same data buffer.

Changes made to a view reflect in the original array.

Views are efficient as they avoid unnecessary memory duplication

You can create a view using ndarray.view()

Example

import numpy as np

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

x = arr.view()

arr[0] = 5555

print (arr) # Output: [5555,2,3,4,5]

print(x) # Output: [5555,2,3,4,5]

### **Broadcasting in NumPy**

Broadcasting in NumPy refers to the ability of NumPy to perform element-wise operations on arrays of different shapes without the need for explicit looping. This feature allows for efficient computation and memory usage, making operations faster compared to traditional looping approaches.

#### **How Broadcasting works (Step-by-Step Explanation)**

1. **Understanding the Rules:** Broadcasting follows specific rules to match arrays of different shapes:

* The arrays are aligned starting from the rightmost dimension
* If dimensions differ, the smaller array is expanded by repeating its values.
* A dimension with size 1 can be stretched to match the corresponding dimension of the larger array

1. **Compatible Shapes:** Two arrays can be broadcast together only if:

* They have the same shape
* The shape of one of the arrays is 1 in a dimension that differs

1. **Applying Broadcasting:** Instead of manually reshaping and repeating arrays NumPy automatically extends the smaller array to match the larger array during operations.

Example:

import numpy as np

# Define a 2D array (3x3)

A = np.array([

[1,2,3],

[4,5,6],

[7,8,9]

])

# Define a 1D Array (1x3)

B = np.array([10,20,30])

# Perform element-wise addition

result = A + B

print (result)

#### **Advantages of Broadcasting**

Efficient – No explicit looping needed, making operations faster

Memory Optimized: Avoids unnecessary copying of data.

Intuitive Syntax: Makes mathematical operations simpler and more readable

#### **Examples on Variety of Broadcasting**

1. **Broadcasting with Scalar Addition**

A scalar (single value) can be broadcast across a NumPy array.

python

import numpy as np

# Define a 2D array

A = np.array([[1, 2, 3],

[4, 5, 6]])

# Scalar addition

result = A + 10

print(result)

**Output:**

[[11 12 13]

[14 15 16]]

**Explanation:** The scalar 10 is broadcast to every element in the array.

1. **Broadcasting with 1D and 2D Arrays**

python

B = np.array([[10], [20]]) # Shape (2,1)

C = np.array([1, 2, 3]) # Shape (1,3)

result = B + C # Broadcasting applied

print(result)

**Output:**

[[11 12 13]

[21 22 23]]

**Explanation:** B (2×1) expands to match C (1×3), resulting in a (2×3) matrix.

1. **Broadcasting with Element-wise Multiplication**

python

X = np.array([[1, 2, 3], [4, 5, 6]]) # Shape (2,3)

Y = np.array([10, 20, 30]) # Shape (3,)

result = X \* Y # Broadcasting applied

print(result)

**Output:**

[[10 40 90]

[40 100 180]]

**Explanation:** Y is expanded vertically to match X.

1. **Broadcasting Across Higher Dimensions**

python

D = np.ones((3, 1, 2)) # Shape (3,1,2)

E = np.array([5, 10]) # Shape (2,)

result = D \* E # Broadcasting applied

print(result)

**Output Shape:** (3,1,2)

[[[ 5 10]]

[[ 5 10]]

[[ 5 10]]]

**Explanation:** E expands to match the second dimension of D.

### **Aggregation Functions in NumPy**

Aggregation functions in NumPy allow you to perform statistical operations on array efficiently. These functions help summarize data by computing values such as sum, mean, max, min, standard deviation, variance and more

1. **Sum (np.sum)**

Computes the sum of all elements in an array

import numpy as np

A = np.array([

[1,2,3],

[4,5,6]

])

# Compute sum of all elements

total = np.sum(A)

# Compute sum along rows (axis = 1)

row\_sum = np.sum(A, axis=1)

# Compute sum along columns (axis = 0)

col\_sum = np.sum(A, axis=0)

print (f"Array A contains:\n {A}")

print (f"Total Sum: {total}")

print (f"Row Sum: {row\_sum}")

print (f"Col Sum: {col\_sum}")

1. **Mean (np.mean)**
   1. import numpy as np
   2. A = np.array([
   3. [1,2,3],
   4. [4,5,6]
   5. ])
   6. # Compute sum of all elements
   7. total\_mean = np.mean(A)
   8. # Compute sum along rows (axis = 1)
   9. row\_mean = np.mean(A, axis=1)
   10. # Compute sum along columns (axis = 0)
   11. col\_mean= np.mean(A, axis=0)
   12. print (f"Array A contains:\n {A}")
   13. print (f"Total Mean: {total\_mean}")
   14. print (f"Row Mean: {row\_mean}")
   15. print (f"Col Mean: {col\_mean}")
2. Maximum (np.max) and Minimum (np.min)
   1. import numpy as np
   2. A = np.array([
   3. [1,2,3],
   4. [4,5,6]
   5. ])
   6. # Compute max & min of all elements
   7. total\_max = np.max(A)
   8. total\_min = np.min(A)
   9. # Compute max & min along rows (axis = 1)
   10. row\_max = np.max(A, axis=1)
   11. row\_min = np.min(A, axis=1)
   12. # Compute max & min along columns (axis = 0)
   13. col\_max= np.max(A, axis=0)
   14. col\_min= np.min(A, axis=0)
   15. print (f"Array A contains:\n {A}")
   16. print (f"Total Max: {total\_max}")
   17. print (f"Total Min: {total\_min}")
   18. print (f"Row Max: {row\_max}")
   19. print (f"Row Min: {row\_min}")
   20. print (f"Col Max: {col\_max}")
   21. print (f"Col Min: {col\_min}")
3. **Standard Deviation (np.std) and Variance (np.var)**
   1. import numpy as np
   2. A = np.array([
   3. [1,2,3],
   4. [4,5,6]
   5. ])
   6. # Compute Std & Var of all elements
   7. total\_std = np.std(A)
   8. total\_var = np.var(A)
   9. # Compute std & var along rows (axis = 1)
   10. row\_std = np.std(A, axis=1)
   11. row\_var = np.var(A, axis=1)
   12. # Compute std & var along columns (axis = 0)
   13. col\_std= np.std(A, axis=0)
   14. col\_var= np.var(A, axis=0)
   15. print (f"Array A contains:\n {A}")
   16. print (f"Total STD: {total\_std}")
   17. print (f"Total VAR: {total\_var}")
   18. print (f"Row STD: {row\_std}")
   19. print (f"Row VAR: {row\_var}")
   20. print (f"Col STD: {col\_std}")
   21. print (f"Col VAR: {col\_var}")
4. **Median (np.median)**
   1. import numpy as np
   2. A = np.array([
   3. [1,2,3],
   4. [4,5,6]
   5. ])
   6. # Compute median of all elements
   7. total\_median = np.median(A)
   8. # Compute median along rows (axis = 1)
   9. row\_median = np.median(A, axis=1)
   10. # Compute median along columns (axis = 0)
   11. col\_median= np.median(A, axis=0)
   12. print (f"Array A contains:\n {A}")
   13. print (f"Total Median: {total\_median}")
   14. print (f"Row Median: {row\_median}")
   15. print (f"Col Median: {col\_median}")
5. **Cumulative Sum (np.cumsum)**
   1. A = np.array([1, 2, 3, 4, 5])
   2. # Compute Cumulative Sum
   3. total\_cumsum = np.cumsum(A)
   4. print (f"Values in Array A: {A}")
   5. print (f"Cumulative Sum of A: {total\_cumsum}")
   6. B = np.array([
   7. [1,2,3],
   8. [4,5,6]
   9. ])
   10. # Compute Cumulative Sum
   11. total\_cumsum = np.cumsum(B)
   12. # Row wise Cumulative Sum
   13. row\_cumsum = np.cumsum(B, axis=1)
   14. # Col wise Cumulative Sum
   15. col\_cumsum = np.cumsum(B, axis=0)
   16. print (f"Values in Array B: \n{B}")
   17. print (f"Cumulative Sum of B: \n{total\_cumsum}")
   18. print (f"Cumulative Row Sum of B: \n{row\_cumsum}")
   19. print (f"Cumulative Col Sum of B: \n{col\_cumsum}")
6. **Cumulative Product (np.cumprod)**
   1. X = np.array([1, 2, 3, 4, 5])
   2. # Compute Cumulative Sum
   3. total\_cumprod = np.cumprod(A)
   4. print (f"Values in Array X: {X}")
   5. print (f"Cumulative Product of X: {total\_cumprod}")
   6. P = np.array([
   7. [1,2,3],
   8. [4,5,6]
   9. ])
   10. # Compute Cumulative Product
   11. total\_cumprod = np.cumprod(P)
   12. # Row wise Cumulative Product
   13. row\_cumprod = np.cumprod(P, axis=1)
   14. # Col wise Cumulative Product
   15. col\_cumprod = np.cumprod(P, axis=0)
   16. print (f"Values in Array P: \n{P}")
   17. print (f"Cumulative Product of P: \n{total\_cumprod}")
   18. print (f"Cumulative Row Product of P: \n{row\_cumprod}")
   19. print (f"Cumulative Col Product of P: \n{col\_cumprod}")

### **Boolean Masking and Filtering in NumPy**

Boolean masking and filtering in NumPy allows you to select, modify or filter elements in an array based on a specific conditions. This technique is widely used in data analysis, machine learning and scientific computing.

#### **Understanding Boolean Masks**

A Boolean mask is a NumPy array containing True/False values that correspond to each element in another array. The True values indicate which elements should be selected.

Example:

A = np.array([1,5,10,25,190,50, 71,20])

A\_bool = A > 50

print (f"Elements for A : {A}")

print (f"Masking elements of A that are > 50 {A\_bool}")

#### **Filtering Elements Using Boolean Mask**

You can use the Boolean mask to filter elements from the array

Example:

A = np.array([1,5,10,25,190,50, 71,20])

A\_bool = A > 50

print (f"Elements for A : {A}")

print (f"Filtering elements of A that are > 50 : {A[A % 2 != 0]}")

### **Sorting and Filtering Arrays in NumPy**

NumPy provides the numpy.sort() function to sort elements in an array.

**Example:**

**Basic Sorting**

import numpy as np

# Create an array

arr = np.array([4,61,2,51,5])

# Sorting the array

sorted\_arr = np.sort(arr)

print(sorted\_arr)

The np.sort() functions returns a new sorted array without modifying the original one.

**Sorting a Multi-Dimensional Array**

arr\_2D = np.array([[3,2,1], [6,5,4]])

# Sorting along rows (default axis=1)

sorted\_rows = np.sort(arr\_2D, axis = 1)

print (sorted\_rows)

# Sortnig along columns

sorted\_columns = np.sort(arr\_2D, axis = 0)

print(sorted\_columns)

axis = 1 sorts each row individually, and axis = 0 sorts each column.

### **Searching for elements in Arrays using NumPy**

Searching involves finding specific values or indices.

#### **Finding an Element’s Index**

arr = np.array([10, 20, 30, 40, 50])

# Find index of element 30

index = np.where(arr == 30)

print(index) # Output: (array([2]),)

The np.where() function returns the index where the element is found.

#### **Searching with Conditions**

arr = np.array([10, 25, 30, 45, 50])

# Find all elements greater than 25

indices = np.where(arr > 25)

print(arr[indices]) # Output: [30 45 50]

You can use conditions to filter elements.

#### **Checking for an Element's Existence**

arr = np.array([5, 10, 15, 20])

# Check if 15 is in the array

exists = np.isin(arr, 15)

print(exists) # Output: [False False True False]

np.isin() checks for the presence of a value.

### **Vectorized Computations & Efficiency Improvements using NumPy**

Vectorized computations in NumPy allow operations to be performed on entire arrays without explicit loops, making them significantly faster and more efficient. This is because NumPy leverages optimized C-based implementations under the hood.

#### **Why Vectorization Matters?**

**Improves Performance:** Eliminates slow Python loops

**Simplifies Code:** Makes it cleaner and more readable

**Handles Large Data Efficiently:** Ideal for big datasets in Machine Learning and Scientific Computing.

##### **Example-1: Element-wise operations without loops**

import numpy as np

# Create an array

arr = np.array([2,4,6,8,10])

# Vectorized addition

result = arr + 2

print (result) # Output: [4,6,8,10,12]

##### **Example-2: Vectorized Matrix Multiplication**

import numpy as np

# Define two matrices

A = np.array([[1,2],[3,4]])

B = np.array([[5,6],[7,8]])

# Matrix multiplication using np.dot()

result = np.dot(A, B)

print(result)

##### **Example-3: Logical Operations on Array**

arr = np.array([10,20,30])

# Check which elements are greater than 15

result = arr > 15

print (result) # Output: [False, True, True]

##### **Example-4: Applying Custom Functions with np.vectorize()**

If you have a custom function, np.vectorize() helps apply it efficiently

import numpy as np

# Define a Custom Function

def custom\_fun(x):

return x\*\*2 + 2\*x + 1

# Create and Array

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

# Apply function using np.vectorize()

vectorized\_func = np.vectorize(custom\_func)

result = vectorized\_fun(arr)

print(result) # Output: [4, 9, 16, 25]

**Summary**

* **Vectorization eliminates loops**, making computations faster.
* **NumPy operations are optimized** for performance.
* **Matrix operations, logical conditions, and custom functions** can be vectorized.

### **Handling missing values using NumPy**

#### **Understanding NaN and inf in NumPy**

* NaN (Not a Number): Represents undefined or missing values, often encountered in datasets with incomplete records
* Inf (infinity): Represents infinite values, usually resulting from division by zero or mathematical operations with very large numbers

#### **Detecting NaN and inf in an array**

NumPy provides np.isnan() and np.isinf() to identify these values in an array.

**Example:**

import numpy as np

# Creating an array with NaN and inf values

arr = np.array([1,2,np.nan,4, np.inf, 6])

# Detect NaN values

nan\_mask = np.isnan(arr)

print(“NaN mask:”, nan\_mask)

# Detect inf values

Inf\_mask = np.isinf(arr)

print(“Inf mask:”, inf\_mask)

# Detect both NaN and inf values

nan\_inf\_mask = np.isnan(arr) | np.isinf(arr)

print("NaN or Inf mask:", nan\_inf\_mask)

**Output:**

NaN mask: [False False True False False False]

Inf mask: [False False False False True False]

NaN or Inf mask: [False False True False True False]

#### **Replacing NaN and inf values**

You can replace missing values with meaningful values like zeros, mean, or median using np.nan\_to\_num()

clean\_arr = np.nan\_to\_num(arr, nan=0, posinf=10, neginf=-10)

print("Replaced array:", clean\_arr)

Replaced array: [ 1. 2. 0. 4. 10. 6.]

#### **Removing NaN and inf values**

Filtering out invalid values can be helpful when you want a clean dataset.

**Example: Removing NaN and inf**

filtered\_arr = arr[~nan\_inf\_mask] # Keep only valid numbers

print("Filtered array:", filtered\_arr)

**Output:**

Filtered array: [1. 2. 4. 6.]

#### **Handling NaN values in calculations**

Many NumPy functions have special versions that ignore NaN values.

**For example:**

* np.nanmean(), np.nanmedian(), np.nanmax() → Compute statistics **ignoring NaN**.
* np.nansum() → Sum **excluding NaN**.

**Example: Computing mean while ignoring NaN**

mean\_val = np.nanmean(arr)

print("Mean ignoring NaN:", mean\_val)

#### **NumPy Structured Array**

NumPy structured arrays are a powerful feature that allow you to create arrays with named fields, similar to columns in a table or attributes in a struct. They are useful when you want to store heterogeneous data types in a single array, making NumPy function like a lightweight database.

**Step-1: Import NumPy**

import numpy as np

**Step-2: Define a Structured Data Type**

Structured arrays allow you to define custom data types with multiple fields. Here is an example of defining a structured data type for a student record.

student\_dtype = np.dtype([

(‘name’, ‘U20’), # Unicode string with max length 20

(‘age’, ‘i4’), # 4-byte integer

(‘grade’, ‘f4’) # 4-byte integer

])

**Step-3: Create a Structured Array**

Now, lets create a NumPy array with this structure data type

students = np.array([

(‘Alice’, 20, 85.5),

(‘Bob’, 22, 90.2),

(‘Charlie’, 21, 78.8)

], dtype=student\_dtype)

**Step-4: Accessing Data**

You can access individual elements or specific fields easily

print(students[0]) # Access first student’s record

print(students[‘name’]) # Get all names

print(students[‘age’][1]) # Get the age of second student (Bob)

**Step-5: Modifying Values**

Structure arrays allow modifications:

students[1][‘grade’] = 95.0 # Update Bob’s grade

print(students)

**Example-2: Employee Records**

Let’s create a structured array to store employee information (Name, ID,Salary)

employee\_dtype = np.dtype([

(‘name’, ‘u20’),

(‘id’, ‘i4’),

(‘salary’, ‘f8’)

])

employees = np.array([

(‘John’, 101, 50000.0),

(‘Luke’, 102, 60000.0),

(‘Mark’, 103, 70000.0)

], dtype=employee\_dtype)

**Access employee Salaries**

print(employees['salary'])

Sort by Salary

sorted\_employees = np.sort(employees, order=’salary’)

print(sorted\_employees)

##### **Why use Structured Arrays?**

* Efficient storage of heterogeneous data types
* Easy field-based access
* Supports sorting and filtering based on specific fields

### **Dot Product (Matrix Multiplication)**

The dot product of two matrices is their matrix multiplication. In NumPy, you can use np.dot() or @ (Python's matrix multiplication operator).

import numpy as np

# Define two matrices

A = np.array( [[1,2], [3,4]] )

B = np.array( [[5,6],[7,8]] )

# Compute dot product

dot\_product = np.dot(A, B)

# Alternative way using @ operator

dot\_product\_alt = (A @ B)

print(“Matrix A:\n”, A)

print(“Matrix B:\n”, B)

print(“Dot product (Matrix Multiplication):\n”, dot\_prod)

### **Transpose of Matrix**

The **transpose** flips the matrix over its diagonal—rows become columns and vice versa. In NumPy, we use .T or np.transpose().

C = np.array([[1, 2, 3], [4, 5, 6]])

# Transpose using .T

transpose\_C = C.T

# Alternative way

transpose\_C\_alt = np.transpose(C)

print("Original Matrix:\n", C)

print("Transpose of Matrix:\n", transpose\_C)

### **Inverse of Matrix**

The **inverse** of a square matrix A is another matrix A⁻¹ such that A \* A⁻¹ = I (where I is the identity matrix). In NumPy, we use np.linalg.inv().

D = np.array([[4, 7], [2, 6]])

# Compute the inverse

inverse\_D = np.linalg.inv(D)

print("Original Matrix:\n", D)

print("Inverse of Matrix:\n", inverse\_D)

**Explanation:**

* The inverse exists only if the matrix is **non-singular** (i.e., its determinant is non-zero).
* The inverse satisfies the property D @ D⁻¹ = I.

### **Linear Equations**

Solving linear equations is a fundamental application of linear algebra, and NumPy provides a powerful function, numpy.linalg.solve(), to solve systems of linear equations efficiently.

**Understanding Linear Equations**

A system of linear equations can be written in matrix form as:

Ax=B

where:

* A is the coefficient matrix,
* x is the unknown variable vector,
* B is the constant matrix.

To find x, we use:

x= A^{-1} B

However, instead of computing the inverse explicitly (which can be computationally expensive), NumPy provides numpy.linalg.solve() to directly solve for xx.

**Example 1: Solving a Simple System**

Let's solve the system:

x + 2y = 1

3x+5y=2

**Step-by-Step Solution**

import numpy as np

# Define coefficient matrix A

A = np.array([[1,2],[3,5]])

# Define constant matrix B

B = np.array([1,2])

# Solve for x

x = np.linalg.solve(A, B)

print(“Solution for x and y:”, x)

**Explanation:**

* The coefficient matrix A contains the coefficients of x and y
* The Constant matrix B contains the right-hand side values
* np.linalg.solve(A,B) computes the exact solution without explicitly finding A^-1

**Example 2: Solving a 3-variable system**

Let’s solve:

2x + y – z = 8

-3x + 2y + 4z = -11

-2x + y + 2z = -3

**Step-by-Step Solution**

# Define coefficient matrix A

A = np.array([[2,1,-1], [-3,2,4], [-2,1,2]])

# Define constant matrix B

B = np.array([8, -11, -3])

# Solve for x, y, and z

solution = np.linalg.solve(A,B)

**Explanation**

* The coefficient matrix AA represents the system of equations.
* The constant matrix BB contains the right-hand side values.
* np.linalg.solve(A, B) efficiently computes the solution.

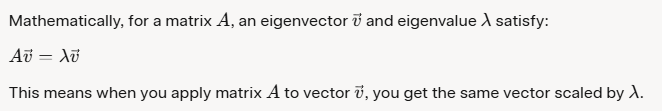
### **Eigenvalues and Eigenvectors**

**What are Eigenvalues and Eigenvectors?**

Imagine a matrix as a machine that takes a vector (a list of numbers, like an arrow in space) and transforms it by stretching, shrinking or rotating it. An Eigenvector is a special vector that, when the matrix transforms it, doesn’t change its direction – it only gets scaled (stretched or shrunk) by a number called eigenvalue.

Eigenvector: A Non-Zero vector that stays along the same line after the matrix transformation.

Eigenvalue: The number that tells you how much the eigenvector is scaled (stretched or shrunk).



**Why are they useful?**

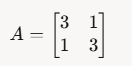
Eigenvalues and eigenvectors help us understand how matrices work in things like computer graphics, physics, or even Google’s Search Algorithms. They simplify complex transformations by finding directions that behave predictably.

#### **Step-by-Step Guide to Calculate Eigenvalues and Eigenvectors**

We will use a simple 2x2 matrix as an example, since its easier to handle, and walk through the calculations step by step. You will be able to deo this with just basic algebra!

**Example Matrix**

Let’s take the matrix:



We want to find its eigenvalues and eigenvectors

**Step-1: Set Up the Eigenvalue Equation**

To find the eigenvalues, we use the equation 

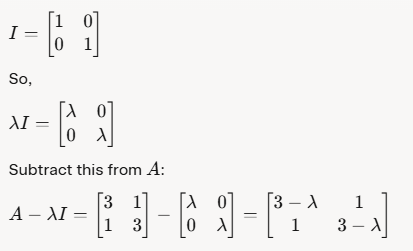
Rewrite it as:



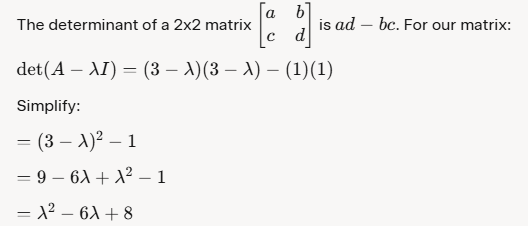
Here, *I* is the identity matrix (same size as *A,* with1s on the diagonal and 0s else where), an  is the eigenvector. For a non-zero eigenvector, the matrix  must have a determinant of zero:



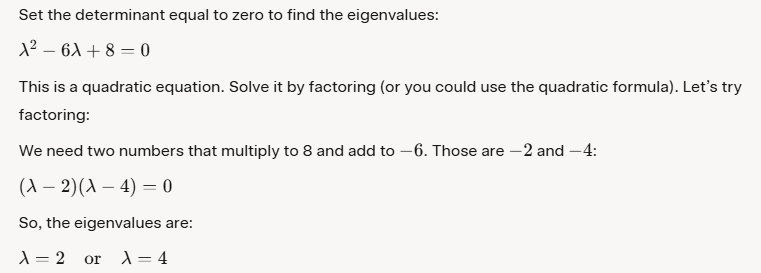
For our 2 x 2 matrix, the identity matrix is:



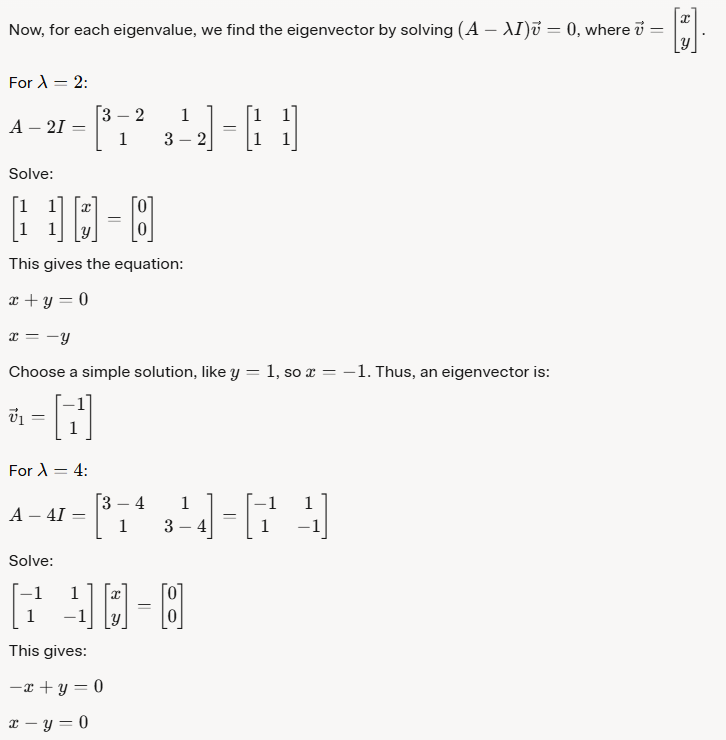
**Step-2: Calculate the Determinant**

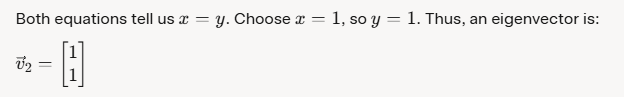


**Step-3: Solve for Eigenvalues**

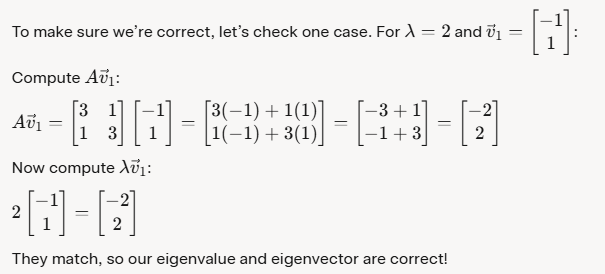


**Step-4: Find Eigenvectors for Each Eigenvalues**

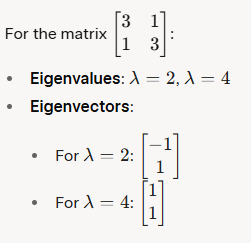




**Step-5: Verify (Optional)**



**Summary of Results**

****

**Notes**

1. NumPy normalizes eigenvectors to have unit length (norm of 1).
2. For complex eigenvalues, the eigenvectors will also be complex.
3. The eigenvalues are not necessarily ordered.
4. For larger matrices, you might want to use np.linalg.eigh() for Hermitian or real symmetric matrices, which is more efficient.

**Handling Special Cases**

**Diagonal Matrix**

For a diagonal matrix, the eigenvalues are simply the diagonal elements, and the eigenvectors are the standard basis vectors.

**Repeated Eigenvalues**

When eigenvalues are repeated, there may be multiple linearly independent eigenvectors (geometric multiplicity equals algebraic multiplicity) or fewer (geometric multiplicity less than algebraic multiplicity).

**Non-Square Matrices**

Remember that eigenvalues and eigenvectors are only defined for square matrices. If you try to compute them for a non-square matrix, NumPy will raise a LinAlgError.

# **What is Pandas?**

Pandas is a powerful, open-source data analysis and manipulation library for Python. It provides easy-to-use data structure and functions designed to make working with structured (tablular, multidimensional, or time-series) data fast and intuitive.

**Key Features of Pandas**

**Data Structures**

* **DataFrame:** A 2D labeled data structure (like a spreadsheet or SQL Table) with columns of different types
* **Series:** A 1D labled array capable of holding any data type

**Data handling:**

* Read and write data from various formats (CSV, Excel, SQL, JSON, etc).
* Handle missing data with dropna(), fillna() and interpolation methods

**Data Manipulation**

* Filter, sort and reshape data
* Merge, join, and concatenate datasets
* Group data using groupby() for aggregation and transformations

**Time Series Support**

* Powerful data/time functionality for resampling, shifting and frequency conversion.

**Performance Optimization**

* Built on NumPy for efficient computation
* Supports vectorized operations for faster execution

**Why Use Pandas?**

* **Ease of Use:** Intuitive syntax for complex operations
* **Versatility:** Works well with other Python Libraries (NumPy, Matplotlib, Scikit-learn)
* **Efficient Data Cleaning:** Simplifies handling missing or inconsistent data.
* **Powerful analysis:** Supports statistical, financial and scientific computations.
* **Integration:** Works seamlessly with databases, Excel and Big Data tools

## **Installation of Pandas**

Pandas is a Python library, so you need Python installed first. Below are the installation steps for different operating systems.

**Prerequisites:**

1. **Python (3.7 or newer)** – Download from [python.org](https://www.python.org/downloads/).
2. **pip** (Python package manager) – Usually comes with Python

### **Windows Installation:**

**Method-1: Using pip (recommended)**

1. Open Command Prompt (CMD)

Prest Win + R, type cmd and hit Enter

1. Check Python & pip installation

python –version

pip --version

1. Install Pandas

pip install pandas

1. Verify Installation

python -c “import pandas as pd; print(pd.\_\_version\_\_)”

**Method-2: Using Anaconda (For Data Science)**

1. Download Anaconda from anaconda.com
2. Open Anaconda Prompt and run:

conda install pandas

### **Linux (Ubuntu/Debian, Fedora etc) Installation**

**Method-1: Using pip**

1. Open terminal (Ctrl + Alt + T)
2. Install Python & pip (if missing)

**For Ubuntu/Debian**

sudo apt update

sudo apt install python3 python3-pip

**For Fedora/RHEL**

sudo dnf install python3 python3-pip

1. Install Pandas

pip3 install pandas

1. Verify Installation

python3 -c “import pandas as pd; print(pd.\_\_version\_\_)”

**Method-2: Using Conda (Alternative)**

1. Install Miniconda/Anaconda (Linux installer)
2. Run

conda install pandas

### **macOS**

**Method-1: Using pip**

1. Open terminal (cmd + space, type terminal)
2. Check Python and pip

python3 – version

pip3 – version

(if missing, install Python from python.org)

1. Install Pandas

pip3 install pandas

1. Verify installation

python3 -c “import pandas as pd; print(pd.\_\_version\_\_)”

**Method-2: Using Homebrew (Alternative)**

1. Install Homebrew (if not installed)

/bin/bash -c "$(curl -fsSL <https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh>)"

1. Install Python and Panda

brew install python

pip3 install pandas

**Method-3: Using Conda**

* 1. Install Miniconda/Anaconda (macOS Installer)
  2. Run

conda install pandas

## **Pandas Data Structure**

Pandas provides two primary data structures that make data manipulation and analysis efficient.

### **Series (1-Dimensional Data)**

A Series is a one-dimensional labelled array capable of holding any data type (integers, strings, floats, python objects, etc.,)

**Key Features of a Series:**

* Similar to a column in an Excel sheet or a list with indices
* Contains data + an index (like row labels)
* Homogeneous (all elements are of the same type)

**Creating a Series**

import pandas as pd

# From a list (default index: 0,1,2…)

s1 = pd.Series([10,20,30,40,50])

print (s1)

Output:

0 10

1 20

2 30

3 40

dtype: int64

Custom Index:

s2 = pd.Series([1,2,3], index=[‘a’,’b’,’c’])

print(s2)

Output:

a 1

b 2

c 3

dtype: int64

**Accessing Series Data**

**By Label (index name)**

print(s2['b']) *# Output: 2*

**By Position (integer index)**

print(s2[1]) *# Output: 2*

### **DataFrame (2-Dimensional Data)**

A DataFrame is a 2D Table (like an Excel sheet or a SQL table) with rows and columns

**Key Features of a DataFrame**

* Rows = Observations (records)
* Columns = variables (features)
* Can hold different data types in different columns.
* Supports label-based indexing (loc) and position-based indexing (iloc)

#### **Creating a DataFrame**

Data = {

“Name”: [“Alice”, “Bob”, “Charlie”],

“Age”: [25,30,35],

“City”: [“New York”, “London”, “Paris”]

}

Output:

Name Age City

0 Alice 25 New York

1 Bob 30 London

2 Charlie 35 Paris

#### **Custom Index**

df = pd.DataFrame(data, index=["P1", "P2", "P3"])

print(df)

Output:

Name Age City

P1 Alice 25 New York

P2 Bob 30 London

P3 Charlie 35 Paris

#### **Accessing DataFrame Data**

##### **Select a column (returns a Series)**

print(df["Name"])

Output:

P1 Alice

P2 Bob

P3 Charlie

Name: Name, dtype: object

##### **Select multiple columns**

print(df[["Name", "Age"]])

##### **Select rows by label (loc):**

print(df.loc["P1"])

##### **Select rows by position (iloc)**

print(df.iloc[0]) # First Row

### **Creating DataFrame (2-Dimensional Data)**

Methods to create a DataFrame

#### **From a Dictionary**

The most common method – keys become column header and values become column data.

import pandas as pd

data = {

“name”: [“Allen”, “Kane”, “Lambert”],

“section”: [“C”,”D”,”E”],

“city”: [“Bangalore”,”Columbo”,”Beijing”]

}

df = pd.DataFrame(data)

print(df)

Output:

name section city

0 Allen C Bangalore

1 Kane D Colombo

2 Lambert E Beijing

**Custom Index:**

df.index(“R1”,”R2”,”R3”)

print(df)

#### **From a List**

list = [

    ["Allen", "C", "Bangalore"],

    ["Kane", "D", "Colombo"],

    ["Lambert", "E", "Beijing"]

]

df = pd.DataFrame(list, columns=["Name", "Section", "City"])

print(f"The newly created DataFrame contains: \n{df}")

df.index = ["L1","L2","L3"]

print(f"DataFrame with Row Headers:\n{df}")

**Output:**

The newly created DataFrame contains:

Name Section City

0 Allen C Bangalore

1 Kane D Colombo

2 Lambert E Beijing

DataFrame with Row Headers:

Name Section City

L1 Allen C Bangalore

L2 Kane D Colombo

L3 Lambert E Beijing

#### **From a List of Dictionaries**

data = [

    {"Name": "Alice", "Age": 25, "City": "New York"},

    {"Name": "Bob", "Age": 30, "City": "Paris"},

    {"Name": "Charlie", "Age": 35, "City": "London"}

]

df = pd.DataFrame(data)

print(f"The newly created DataFrame contains: \n{df}")

df.index = ["LD1","LD2","LD3"]

print(f"DataFrame with Row Headers:\n{df}")

**Output:**

The newly created DataFrame contains:

Name Age City

0 Alice 25 New York

1 Bob 30 Paris

2 Charlie 35 London

DataFrame with Row Headers:

Name Age City

LD1 Alice 25 New York

LD2 Bob 30 Paris

LD3 Charlie 35 London

#### **From a CSV or Excel File**

Pandas can read external files directly into DataFrame

df = pd.read\_csv("data.csv") *# Assumes 'data.csv' exists*

print(df.head())

#### **From a NumPy Array**

Useful when working with numerical data

import numpy as np

data = np.array([

[1, "Alice", 25],

[2, "Bob", 30],

[3, "Charlie", 35]

])

df = pd.DataFrame(data, columns=["ID", "Name", "Age"])

print(df)

**Output:**

ID Name Age

0 1 Alice 25

1 2 Bob 30

2 3 Charlie 35

#### **From a SQL Database**

Pandas can fetch data from SQL database

import sqlite3

conn = sqlite3.connect("database.db")

df = pd.read\_sql("SELECT \* FROM users", conn)

print(df.head())

## **Data Manipulation**

Data manipulation and cleaning are critical steps in data analysis, and Python's

Pandas library is a powerful tool for these tasks.

### **Data Cleaning**

Data cleaning involves handling missing values, removing duplicates, correcting

data types, and dealing with inconsistencies.

#### **Handling Missing Values**

##### **Identifying Missing Values**

Use is.na() or is.null() to detect missing values.

In **Python**, NaN (Not a Number) and None serve different purposes, but both can indicate missing values depending on the context.

**Key Differences:**

|  |  |  |
| --- | --- | --- |
| **Attribute** | **NaN (Not a Number)** | **None** |
| **Type** | float (from NumPy/Pandas) | NoneType |
| **Usage** | Used in numerical computations | Represents a null or missing value |
| **Behavior in Pandas** | Recognized as a missing value | Also treated as missing, but converted to NaN in numerical columns |
| **Mathematical Operations** | Propagates (e.g., NaN + 5 = NaN) | Throws an error (e.g., None + 5 raises TypeError) |
| **Checking for Missing Values** | Use pd.isna() or np.isnan() | Use is None or pd.isna() |

###### **is.na()**

**Example-1**

import pandas as pd

import numpy as np

# Sample DataFrame

data = {

'Name': ['Alice', 'Bob', 'Charlie', 'David', None],

'Age': [25, np.nan, 30, 22, 28],

'Salary': [50000, 60000, np.nan, 45000, 52000],

'Department': ['HR', 'IT', 'IT', 'Marketing', 'HR']

}

df = pd.DataFrame(data)

print("Original DataFrame:")

print(df)

**Output:**

Name Age Salary Department

0 Alice 25.0 50000.0 HR

1 Bob NaN 60000.0 IT

2 Charlie 30.0 NaN IT

3 David 22.0 45000.0 Marketing

4 None 28.0 52000.0 HR

**# Check for missing values**

print("Missing Values:")

print(df.isna())

**Output:**

Name Age Salary Department

0 False False False False

1 False True False False

2 False False True False

3 False False False False

4 True False False False

###### **is.null()**

**Example-2**

print(f"Original DataFrame: \n{df}")

print(f"\nDataFrame with is.null\n{df.isnull()}")

**Output:**

#######################################################################################################

Example-2: Handling Missing Values

Using: isnull()

#######################################################################################################

Original DataFrame:

Name Age Salary Department

0 Alice 25.0 50000.0 HR

1 Bob NaN 60000.0 IT

2 Charlie 30.0 NaN IT

3 David 22.0 45000.0 Marketing

4 None 28.0 52000.0 HR

DataFrame with is.null

Name Age Salary Department

0 False False False False

1 False True False False

2 False False True False

3 False False False False

4 True False False False

##### **Dropping Missing Values**

Use dropna() to remove rows or columns with missing values.

**# Drop rows with any missing values**

df\_dropped = df.dropna()

print("After dropping rows with missing values:")

print(df\_dropped)

You can also specify conditions, e.g., drop rows where specific columns have missing values:

# Drop rows where 'Age' is missing

df\_dropped\_age = df.dropna(subset=['Age'])

print("After dropping rows with missing Age:")

print(df\_dropped\_age)

**Output:**

Original DataFrame:

Name Age Salary Department

0 Alice 25.0 50000.0 HR

1 Bob NaN 60000.0 IT

2 Charlie 30.0 NaN IT

3 David 22.0 45000.0 Marketing

4 None 28.0 52000.0 HR

After dropping rows with missing values:

Name Age Salary Department

0 Alice 25.0 50000.0 HR

3 David 22.0 45000.0 Marketing

##### **Filling Missing Values**

###### **fillna()**

Use fillna() to replace missing values with a specific value, mean, median or forward/backward fill.

*# Fill missing Age with the mean*

df['Age'] = df['Age'].fillna(df['Age'].mean())

print("After filling missing Age with mean:")

print(df)

*# Fill missing Name with 'Unknown'*

df['Name'] = df['Name'].fillna('Unknown')

print("\nAfter filling missing Name with 'Unknown':")

print(df)

**Output:**

After filling missing Age with mean:

Name Age Salary Department

0 Alice 25.0 50000.0 HR

1 Bob 26.25 60000.0 IT

2 Charlie 30.0 NaN IT

3 David 22.0 45000.0 Marketing

4 None 28.0 52000.0 HR

After filling missing Name with 'Unknown':

Name Age Salary Department

0 Alice 25.00 50000.0 HR

1 Bob 26.25 60000.0 IT

2 Charlie 30.00 NaN IT

3 David 22.00 45000.0 Marketing

4 Unknown 28.00 52000.0 HR

##### **Interpolating Missing Values**

###### **interpolate()**

Use interpolate() for linear interpolation of numeric data.

*# Interpolate missing Salary values*

df['Salary'] = df['Salary'].interpolate()

print("After interpolating missing Salary:")

print(df)

**Output:**

text

Copy

Name Age Salary Department

0 Alice 25.00 50000.0 HR

1 Bob 26.25 60000.0 IT

2 Charlie 30.00 52500.0 IT

3 David 22.00 45000.0 Marketing

4 Unknown 28.00 52000.0 HR

#### **Removing Duplicates**

##### **drop\_duplilcates()**

Use drop\_duplicates() to remove duplicate rows.

*# Add a duplicate row for demonstration*

df = pd.concat([df, df.iloc[[0]]], ignore\_index=True)

print("DataFrame with duplicate row:")

print(df)

*# Remove duplicates*

df\_no\_duplicates = df.drop\_duplicates()

print("\nAfter removing duplicates:")

print(df\_no\_duplicates)

**Output:**

DataFrame with duplicate row:

Name Age Salary Department

0 Alice 25.00 50000.0 HR

1 Bob 26.25 60000.0 IT

2 Charlie 30.00 52500.0 IT

3 David 22.00 45000.0 Marketing

4 Unknown 28.00 52000.0 HR

5 Alice 25.00 50000.0 HR

After removing duplicates:

Name Age Salary Department

0 Alice 25.00 50000.0 HR

1 Bob 26.25 60000.0 IT

2 Charlie 30.00 52500.0 IT

3 David 22.00 45000.0 Marketing

4 Unknown 28.00 52000.0 HR

#### **Correcting Data Types**

Use astype() or pd.to\_numeric() to ensure correct data types.

##### **astype()**

*# Convert Age to integer*

df['Age'] = df['Age'].astype(int)

print("After converting Age to integer:")

print(df)

**Output:**

text

Copy

Name Age Salary Department

0 Alice 25 50000.0 HR

1 Bob 26 60000.0 IT

2 Charlie 30 52500.0 IT

3 David 22 45000.0 Marketing

4 Unknown 28 52000.0 HR

5 Alice 25 50000.0 HR

### **Data Manipulation Methods**

Data manipulation involves filtering, sorting, grouping, merging, and reshaping data.

#### **Filtering Data**

Use boolean indexing or query() to filter rows based on conditions.

*# Filter employees with Salary > 50000*

high\_salary = df[df['Salary'] > 50000]

print("Employees with Salary > 50000:")

print(high\_salary)

*# Using query*

high\_salary\_query = df.query('Salary > 50000')

print("\nUsing query method:")

print(high\_salary\_query)

**Output:**

Employees with Salary > 50000:

Name Age Salary Department

1 Bob 26 60000.0 IT

2 Charlie 30 52500.0 IT

4 Unknown 28 52000.0 HR

Using query method:

Name Age Salary Department

1 Bob 26 60000.0 IT

2 Charlie 30 52500.0 IT

4 Unknown 28 52000.0 HR

#### **Sorting Data**

Use sort\_values() to sort by one or more columns.

*# Sort by Salary in descending order*

sorted\_df = df.sort\_values(by='Salary', ascending=False)

print("Sorted by Salary (descending):")

print(sorted\_df)

**Output:**

text

Copy

Name Age Salary Department

1 Bob 26 60000.0 IT

2 Charlie 30 52500.0 IT

4 Unknown 28 52000.0 HR

0 Alice 25 50000.0 HR

5 Alice 25 50000.0 HR

3 David 22 45000.0 Marketing

#### **Grouping and Aggregating**

Use groupby() to group data and apply aggregation functions like mean(), sum(), or count().

*# Group by Department and calculate average Salary*

grouped = df.groupby('Department')['Salary'].mean().reset\_index()

print("Average Salary by Department:")

print(grouped)

**Output:**

text

Copy

Department Salary

0 HR 51000.0

1 IT 56250.0

2 Marketing 45000.0

#### **Merging and Joining**

Use merge() or concat() to combine DataFrames.

*# Create another DataFrame*

data2 = {

'Department': ['HR', 'IT', 'Marketing'],

'Location': ['New York', 'San Francisco', 'Chicago']

}

df2 = pd.DataFrame(data2)

*# Merge DataFrames on Department*

merged\_df = pd.merge(df, df2, on='Department', how='left')

print("Merged DataFrame:")

print(merged\_df)

**Output:**

Name Age Salary Department Location

0 Alice 25 50000.0 HR New York

1 Bob 26 60000.0 IT San Francisco

2 Charlie 30 52500.0 IT San Francisco

3 David 22 45000.0 Marketing Chicago

4 Unknown 28 52000.0 HR New York

5 Alice 25 50000.0 HR New York

#### **Reshaping Data**

Use pivot(), melt(), or stack() to reshape data.

##### **Pivoting**

Create a pivot table to summarize data.

*# Pivot table: Average Salary by Department and Name*

pivot\_table = df.pivot\_table(values='Salary', index='Department', columns='Name', aggfunc='mean')

print("Pivot Table:")

print(pivot\_table)

**Output:**

Name Alice Bob Charlie David Unknown

Department

HR 50000.0 NaN NaN NaN 52000.0

IT NaN 60000.0 52500.0 NaN NaN

Marketing NaN NaN NaN 45000.0 NaN

##### **Melting**

Convert wide-format data to long format using melt().

*# Melt DataFrame*

melted = pd.melt(df, id\_vars=['Name'], value\_vars=['Age', 'Salary'], var\_name='Metric', value\_name='Value')

print("Melted DataFrame:")

print(melted)

**Output:**

Name Metric Value

0 Alice Age 25.0

1 Bob Age 26.0

2 Charlie Age 30.0

3 David Age 22.0

4 Unknown Age 28.0

5 Alice Age 25.0

6 Alice Salary 50000.0

7 Bob Salary 60000.0

8 Charlie Salary 52500.0

9 David Salary 45000.0

10 Unknown Salary 52000.0

11 Alice Salary 50000.0

##### **Applying Custom Transformations**

Use apply() or map() to apply custom functions.

###### **apply()**

The apply function in pandas is used to apply a function along an axis of a DataFrame or on a Series. It’s a flexible way to perform custom operations on rows, columns, or individual elements that aren’t easily vectorized.

**Key Points**

* **DataFrame.apply**: Applies a function along an axis (rows or columns) of a DataFrame.
  + axis=0 (default): Apply the function to each column.
  + axis=1: Apply the function to each row.
* **Series.apply**: Applies a function to each element in a Series.
* Useful for custom functions when built-in pandas methods don’t suffice.
* Returns a DataFrame, Series, or scalar depending on the function and input.

**Syntax**

* **DataFrame.apply**: df.apply(func, axis=0, \*\*kwargs)
* **Series.apply**: series.apply(func, \*\*kwargs)

*# Add a new column with Salary increased by 10%*

df['Salary\_Increased'] = df['Salary'].apply(lambda x: x \* 1.1)

print("DataFrame with increased Salary:")

print(df)

**Output:**

text

Copy

Name Age Salary Department Salary\_Increased

0 Alice 25 50000.0 HR 55000.0

1 Bob 26 60000.0 IT 66000.0

2 Charlie 30 52500.0 IT 57750.0

3 David 22 45000.0 Marketing 49500.0

4 Unknown 28 52000.0 HR 57200.0

5 Alice 25 50000.0 HR 55000.0

**Additional Tips for Data Cleaning**

* **String Operations:** Use str methods for cleaning text data, e.g., df['Name'].str.upper().
* **Replacing Values:** Use replace() to correct specific values, e.g., df['Department'].replace('HR', 'Human Resources').
* **Outlier Detection:** Use statistical methods or IQR to identify and handle outliers.
* **Datetime Handling:** Convert strings to datetime using pd.to\_datetime() for time-series data.

**Notes**

* **Performance**: apply can be slower than vectorized operations (e.g., df['A'] \* 2). Use vectorized methods when possible for better performance.
* **Alternatives**: Consider numpy operations, map for Series, or applymap (for element-wise operations on DataFrames, deprecated in favor of map in newer pandas versions).
* **Flexibility**: apply is ideal for complex logic that can’t be easily vectorized.

###### **map()**

The map function in pandas is used to apply a function to each element in a Series. It’s similar to the apply function for a Series but is specifically designed for element-wise transformations and is often more efficient. Unlike DataFrame.apply, which can operate on rows or columns, map is limited to Series and is typically used for simple transformations or mapping values to new ones using a dictionary or function.

**Key Points**

* **Series.map**: Applies a function or mapping (e.g., dictionary) to each element in a Series.
* Commonly used for:
  + Replacing values using a dictionary.
  + Applying a simple function to each element.
* Returns a new Series with transformed values.
* More efficient than apply for Series operations.
* Does not work on DataFrames (use DataFrame.apply or DataFrame.map for element-wise operations on DataFrames in newer pandas versions).

**Syntax**

Series.map(func, na\_action=None)

* + func: A function, dictionary, or Series to apply.
  + na\_action: Can be None (default, applies function to NaN values) or 'ignore' (skips NaN values).

**Example 1: Using map with a Dictionary**

Suppose you want to replace values in a Series using a dictionary.

import pandas as pd

# Sample Series

data = pd.Series(['A', 'B', 'C', 'A'])

# Dictionary for mapping

grade\_map = {'A': 'Excellent', 'B': 'Good', 'C': 'Average'}

# Apply map to replace values

result = data.map(grade\_map)

print(result)

**Output:**

0 Excellent

1 Good

2 Average

3 Excellent

dtype: object

**Explanation:** Each value in the Series is replaced with the corresponding value from the grade\_map dictionary. If a value isn’t found in the dictionary, it becomes NaN unless specified otherwise.

**Example 2: Using map with a Function**

Let’s apply a function to transform numerical values in a Series.

import pandas as pd

# Sample Series

scores = pd.Series([85, 60, 95])

# Function to categorize scores

def grade(score):

if score >= 90:

return 'A'

elif score >= 70:

return 'B'

else:

return 'C'

# Apply the grade function using map

result = scores.map(grade)

print(result)

**Output:**

0 B

1 C

2 A

dtype: object

**Explanation:** The grade function is applied to each element in the scores Series, returning a new Series with letter grades.

Example 3: Handling NaN Values with na\_action

If your Series contains missing values, you can control how map handles them.

import pandas as pd

import numpy as np

# Sample Series with NaN

data = pd.Series([1, 2, np.nan, 4])

# Function to square a number

def square(x):

return x \*\* 2

# Apply map with na\_action='ignore'

result = data.map(square, na\_action='ignore')

print(result)

**Output:**

0 1.0

1 4.0

2 NaN

3 16.0

dtype: float64

**Explanation:** The square function is applied only to non-NaN values because na\_action='ignore'. If na\_action=None, attempting to apply the function to NaN would raise an error or produce NaN.

**Example 4: Using map for String Operations**

You can use map to transform strings in a Series.

import pandas as pd

# Sample Series

names = pd.Series(['alice', 'bob', 'charlie'])

# Use map to capitalize each name

result = names.map(str.capitalize)

print(result)

Output:

0 Alice

1 Bob

2 Charlie

dtype: object

**Explanation:** The str.capitalize function is applied to each element in the names Series, capitalizing the first letter of each name.

**Notes**

* **Performance**: map is generally faster than apply for Series because it’s optimized for element-wise operations.
* **Limitations**: map only works on a Series, not a DataFrame. For DataFrames, use DataFrame.apply (for row/column operations) or DataFrame.map (for element-wise operations, available in pandas 2.0+).
* **Dictionary Mapping**: Using a dictionary with map is a common way to perform lookups or replacements.
* **Alternatives**: For vectorized operations, consider pandas’ built-in methods (e.g., str methods for strings) or numpy functions for numerical data, as they’re often faster.

**Comparison with apply**

* map is for Series only; apply works on both Series and DataFrames.
* map supports dictionary mapping; apply does not.
* For Series, map is usually faster than apply for simple transformations.

###### **applymap()**

The applymap() function in pandas is used to apply a function to every single element in a DataFrame. It's similar to map() for Series, but works on entire DataFrames.

Key Points about applymap():

* Operates element-wise (on each individual cell)
* Returns a new DataFrame with transformed values
* Works with functions that take a single value and return a single value
* Slower than vectorized operations, so use with caution on large DataFrames

import pandas as pd

import numpy as np

# Create a sample DataFrame

data = {

'A': [1.23, 4.56, 7.89],

'B': [9.87, 6.54, 3.21],

'C': [0.12, 3.45, 6.78]

}

df = pd.DataFrame(data)

print("Original DataFrame:")

print(df)

**Output:**

Original DataFrame:

A B C

0 1.23 9.87 0.12

1 4.56 6.54 3.45

2 7.89 3.21 6.78

**Example 1: Round all numbers to 1 decimal place**

rounded\_df = df.applymap(lambda x: round(x, 1))

print("\nRounded DataFrame:")

print(rounded\_df)

Output:

Rounded DataFrame:

A B C

0 1.2 9.9 0.1

1 4.6 6.5 3.5

2 7.9 3.2 6.8

**Example 2: Apply a custom function to each element**

def double\_if\_positive(x):

return x \* 2 if x > 0 else x

transformed\_df = df.applymap(double\_if\_positive)

print("\nTransformed DataFrame:")

print(transformed\_df)

**Output:**

Transformed DataFrame:

A B C

0 2.46 19.74 0.24

1 9.12 13.08 6.90

2 15.78 6.42 13.56

**When to Use applymap() vs Other Methods:**

* Use applymap() when you need to transform every element in the DataFrame
* Use apply() when you need to operate on entire rows or columns
* For vectorized operations (like multiplying all values by 2), prefer direct operations (df \* 2) as they're much faster

**Performance Consideration:**

For large DataFrames, applymap() can be slow because it operates element-by-element. Consider using vectorized operations or NumPy functions when possible for better performance.

**Summary of Key Methods**

| **Task** | **Method** | **Purpose** |
| --- | --- | --- |
| Identify Missing Values | isna(), isnull() | Detect NaN or None values |
| Drop Missing Values | dropna() | Remove rows/columns with missing values |
| Fill Missing Values | fillna(), interpolate() | Replace missing values |
| Remove Duplicates | drop\_duplicates() | Remove duplicate rows |
| Filter Data | Boolean indexing, query() | Select rows based on conditions |
| Sort Data | sort\_values() | Sort by column(s) |
| Group and Aggregate | groupby(), agg() | Group data and apply aggregation |
| Merge DataFrames | merge(), concat() | Combine multiple DataFrames |
| Reshape Data | pivot(), melt() | Transform data structure |
| Custom Transformations | apply(), map() | Apply custom functions to data |