Why Use NumPy?

Python lists are flexible but **slow** for numerical computing because they:

- Store elements as **pointers** instead of a continuous block of memory.
- Lack **vectorized operations**, relying on loops instead.
- Have significant **overhead** due to dynamic typing.

NumPy's Superpowers:

- **Faster** than Python lists (C-optimized backend)
- **Uses less memory** (efficient storage)
- **Supports vectorized operations** (no explicit loops needed)
- **Has built-in mathematical functions**

NumPy vs. Python Lists – Performance Test

Let's compare Python lists with NumPy arrays using a simple example.

Example 1: Adding Two Lists vs. NumPy Arrays

```
import numpy as np
import time

# Python list
size = 1_000_000
list1 = list(range(size))
list2 = list(range(size))

start = time.time()
result = [x + y for x, y in zip(list1, list2)]
end = time.time()
```

```
print("Python list addition time:", end - start)

# NumPy array
arr1 = np.array(list1)
arr2 = np.array(list2)

start = time.time()
result = arr1 + arr2  # Vectorized operation
end = time.time()
print("NumPy array addition time:", end - start)
```

Key Takeaway: NumPy is significantly **faster** because it performs operations in **C**, avoiding Python loops.

Creating NumPy Arrays

```
import numpy as np

# Creating a 1D NumPy array
arr1 = np.array([1, 2, 3, 4, 5])
print(arr1)

# Creating a 2D NumPy array
arr2 = np.array([[1, 2, 3], [4, 5, 6]])
print(arr2)

# Checking type and shape
print("Type:", type(arr1))
print("Shape:", arr2.shape)
```

◇ **NumPy stores data in a contiguous memory block**, making access faster than lists. ◇ `shape` shows the **dimensions** of an array.

Memory Efficiency – NumPy vs. Lists

Let's check memory consumption.

```
import sys

list_data = list(range(1000))
numpy_data = np.array(list_data)

print("Python list size:", sys.getsizeof(list_data) * len(list_data), "bytes")
print("NumPy array size:", numpy_data.nbytes, "bytes")
```

NumPy arrays use significantly less memory compared to Python lists.

Vectorization – No More Loops!

NumPy avoids loops by applying operations to entire arrays at once using SIMD (Single Instruction, Multiple Data) and other low-level optimizations. SIMD is a CPU-level optimization provided by modern processors.

Example 2: Squaring Elements

```
# Python list (loop-based)
list_squares = [x ** 2 for x in list1]

# NumPy (vectorized)
numpy_squares = arr1 ** 2
```

- NumPy is **cleaner** and **faster**!

Summary

- NumPy is **faster** than Python lists because it is optimized in **C**.

- It consumes **less memory** due to efficient storage.
- It provides **vectorized operations**, removing the need for slow loops.
- Essential for **data science** and **machine learning** workflows.

Exercises for Practice

- Create a NumPy array with values from **10 to 100** and print its shape.
- Compare the time taken to multiply **two Python lists** vs. **two NumPy arrays**.
- Find the **memory size** of a NumPy array with **1 million elements**.

Creating NumPy Arrays

Why NumPy Arrays?

NumPy arrays are the **core** of numerical computing in Python. They are:

- **Faster** than Python lists (C-optimized)
- **Memory-efficient** (store data in a contiguous block)
- **Support vectorized operations that support SIMD** (no slow Python loops)
- **Used in ML, Data Science, and AI**

---

1. Creating NumPy Arrays

From Python Lists:

```
import numpy as np


arr1 = np.array([1, 2, 3, 4, 5])  # 1D array
arr2 = np.array([[1, 2, 3], [4, 5, 6]])  # 2D array
```

```
print(arr1)  # [1 2 3 4 5]
print(arr2)
# [[1 2 3]
#  [4 5 6]]
```

- **Unlike lists, all elements must have the same data type.**

Creating Arrays from Scratch:

```
np.zeros((3, 3))    # 3x3 array of zeros
np.ones((2, 4))     # 2x4 array of ones
np.full((2, 2), 7)  # 2x2 array filled with 7
np.eye(4)           # 4x4 identity matrix
np.arange(1, 10, 2) # [1, 3, 5, 7, 9] (like range)
np.linspace(0, 1, 5) # [0. 0.25 0.5 0.75 1.] (evenly spaced)
```

Key Takeaway: NumPy offers powerful shortcuts to create arrays **without loops!**

---

2. Checking Array Properties

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

print("Shape:", arr.shape)   # (2, 3) → 2 rows, 3 columns
print("Size:", arr.size)     # 6 → total elements
print("Dimensions:", arr.ndim) # 2 → 2D array
print("Data type:", arr.dtype) # int64 (or int32 on Windows)
```

- **NumPy arrays are strongly typed**, meaning all elements share the same data type.

---

3. Changing Data Types

```
arr = np.array([1, 2, 3], dtype=np.float32)  # Explicit type
print(arr.dtype)  # float32

arr_int = arr.astype(np.int32)  # Convert float to int
print(arr_int)  # [1 2 3]
```

- **Efficient memory usage** by choosing the right data type.

---

4. Reshaping and Flattening Arrays

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

reshaped = arr.reshape((3, 2))  # Change shape
print(reshaped)
# [[1 2]
#  [3 4]
#  [5 6]]

flattened = arr.flatten()  # Convert 2D → 1D
print(flattened)  # [1 2 3 4 5 6]
```

Indexing and slicing

Lets now learn about indexing and slicing in Numpy

Indexing (Same as Python Lists)

```
arr = np.array([10, 20, 30, 40])
print(arr[0])  # 10
print(arr[-1]) # 40
```

Slicing (Extracting Parts of an Array)

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

print(arr[1:4])  # [20 30 40] (slice from index 1 to 3)
print(arr[:3])   # [10 20 30] (first 3 elements)
print(arr[::2])  # [10 30 50] (every 2nd element)
```

Slicing returns a view, not a copy! Changes affect the original array.**

This might seem counterintuitive since Python lists create copies when sliced. But in NumPy, slicing returns a view of the original array. Both the sliced array and the original array share the same data in memory, so changes in the slice affect the original array.

Why does this happen?

- Memory Efficiency: Avoids unnecessary copies, making operations faster and saving memory.
- Performance: Enables faster access and manipulation of large datasets without duplicating data.

```
sliced = arr[1:4]
sliced[0] = 999
print(arr)  # [10 999 30 40 50]
```

- Use `.copy()` if you need an independent copy.

---

6. Fancy Indexing & Boolean Masking

Fancy Indexing (Select Multiple Elements)

```
arr = np.array([10, 20, 30, 40, 50])
idx = [0, 2, 4]  # Indices to select
print(arr[idx])  # [10 30 50]
```

Boolean Masking (Filter Data)

```
arr = np.array([10, 20, 30, 40, 50])
mask = arr > 25  # Condition: values greater than 25
print(arr[mask])  # [30 40 50]
```

This is a powerful way to filter large datasets efficiently!

Summary

- **NumPy arrays** are faster, memory-efficient alternatives to lists.
- You can create arrays using `np.array()`, `np.zeros()`, `np.ones()`, etc.
- **Indexing & slicing** allow efficient data manipulation.
- **Reshaping & flattening** change array structures without copying data.
- **Fancy indexing & boolean masking** help filter and access specific data.

Exercises for Practice

- Create a **3×3** array filled with random numbers and print its shape.
- Convert an array of floats `[1.1, 2.2, 3.3]` into integers.
- Use **fancy indexing** to extract **even numbers** from `[1, 2, 3, 4, 5, 6]`.
- Reshape a **1D array of size 9** into **a 3×3 matrix**.

- Use **boolean masking** to filter numbers **greater than 50** in an array.

Multidimensional Indexing and Axis

NumPy allows you to efficiently work with **multidimensional arrays**, where indexing and axis manipulation play a crucial role. Understanding how indexing works across multiple dimensions is essential for data science and machine learning tasks.

---

1. Understanding Axes in NumPy

Each dimension in a NumPy array is called an **axis**. Axes are numbered starting from **0**.

For example:

- **1D array** → 1 axis (axis 0)
- **2D array** → 2 axes (axis 0 = rows, axis 1 = columns)
- **3D array** → 3 axes (axis 0 = depth, axis 1 = rows, axis 2 = columns)

Example: Axes in a 2D Array

```
import numpy as np

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

print(arr)
```

Output:

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

- **Axis 0 (rows)** → Operations move **down** the columns.
- **Axis 1 (columns)** → Operations move **across** the rows.

Summing along axes:

```
print(np.sum(arr, axis=0))  # Sum along rows (down each column)
print(np.sum(arr, axis=1))  # Sum along columns (across each row)
```

Output:

```
[12 15 18]  # Column-wise sum
[ 6 15 24]  # Row-wise sum
```

---

2. Indexing in Multidimensional Arrays

You can access elements using **row and column indices**.

```
# Accessing an element
print(arr[1, 2])  # Row index 1, Column index 2 → Output: 6
```

You can also use **slicing** to extract parts of an array:

```
print(arr[0:2, 1:3])  # Extracts first 2 rows and last 2 columns
```

Output:

```
[[2 3]
 [5 6]]
```

3. Indexing in 3D Arrays

For **3D arrays**, the first index refers to the "depth" (sheets of data).

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

# Output of arr3D.shape is → (depth, rows, columns)
print(arr3D.shape)  # Output: (2, 2, 3)
```

Accessing elements in 3D:

```
# First sheet, second row, third column
print(arr3D[0, 1, 2])  # Output: 6

print(arr3D[:, 0, :])   # Get the first row from both sheets
```

4. Practical Example: Selecting Data Along Axes

```
# Get all rows of the first column
first_col = arr[:, 0]
print(first_col)  # Output: [1 4 7]
```

```
# Get the first row from each "sheet" in a 3D array
first_rows = arr3D[:, 0, :]
print(first_rows)
```

Output:

```
[[ 1  2  3]
 [ 7  8  9]]
```

5. Changing Data Along an Axis

```
# Replace all elements in column 1 with 0
arr[:, 1] = 0
print(arr)
```

Output:

```
[[1 0 3]
 [4 0 6]
 [7 0 9]]
```

6. Summary

- Axis 0 = rows (vertical movement), Axis 1 = columns (horizontal movement)
- Indexing works as `arr[row, column]` for 2D arrays and `arr[depth, row, column]` for 3D arrays
- Slicing allows extracting subarrays
- Operations along axes help efficiently manipulate data without loops

Data Types in NumPy

Let's learn about NumPy's **data types** and explore how they affect memory usage and performance in your arrays.

1. Introduction to NumPy Data Types

NumPy arrays are **homogeneous**, meaning that they can only store elements of the same type. This is different from Python lists, which can hold mixed data types. NumPy supports various **data types** (also called **dtypes**), and understanding them is crucial for optimizing memory usage and performance.

Common Data Types in NumPy:

- `int32`, `int64`: Integer types with different bit sizes.
- `float32`, `float64`: Floating-point types with different precision.
- `bool`: Boolean data type.
- `complex64`, `complex128`: Complex number types.
- `object`: For storing objects (e.g., Python objects, strings).

You can check the dtype of a NumPy array using the `.dtype` attribute.

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])
print(arr.dtype)  # Output: int64 (or int32 depending on the system)
```

---

2. Changing Data Types

You can **cast** (convert) the data type of an array using the `.astype()` method. This is useful when you need to change the type for a specific operation or when you want to reduce memory usage.

Example: Changing Data Types

```
arr = np.array([1.5, 2.7, 3.9])
print(arr.dtype)  # Output: float64

arr_int = arr.astype(np.int32)  # Converting float to int
print(arr_int)    # Output: [1 2 3]
print(arr_int.dtype)  # Output: int32
```

Example: Downcasting to Save Memory

```
arr_large = np.array([1000000, 2000000, 3000000], dtype=np.int64)
arr_small = arr_large.astype(np.int32)  # Downcasting to a smaller dtype
```

```
print(arr_small)  # Output: [1000000 2000000 3000000]
print(arr_small.dtype)  # Output: int32
```

---

3. Why Data Types Matter in NumPy

The choice of data type affects: - **Memory Usage**: Smaller data types use less memory. - **Performance**: Operations on smaller data types are faster due to less data being processed. - **Precision**: Choosing the appropriate data type ensures that you don't lose precision (e.g., using `float32` instead of `float64` if you don't need that extra precision).

Example: Memory Usage

```
arr_int64 = np.array([1, 2, 3], dtype=np.int64)
arr_int32 = np.array([1, 2, 3], dtype=np.int32)

print(arr_int64.nbytes)  # Output: 24 bytes (3 elements * 8 bytes each)
print(arr_int32.nbytes)  # Output: 12 bytes (3 elements * 4 bytes each)
```

---

4. String Data Type in NumPy

Although NumPy arrays typically store numerical data, you can also store strings by using the `dtype='str'` or `dtype='U'` (Unicode string) format. However, working with strings in NumPy is **less efficient** than using lists or Python's built-in string types.

Example: String Array

```
arr = np.array(['apple', 'banana', 'cherry'], dtype='U10')  # Unicode string array
print(arr)
```

---

5. Complex Numbers

NumPy also supports **complex numbers**, which consist of a real and imaginary part. You can store complex numbers using `complex64` or `complex128` data types.

Example: Complex Numbers

```
arr = np.array([1 + 2j, 3 + 4j, 5 + 6j], dtype='complex128')
print(arr)
```

---

6. Object Data Type

If you need to store mixed or complex data types (e.g., Python objects), you can use `dtype='object'`. However, this type sacrifices performance, so it should only be used when absolutely necessary.

Example: Object Data Type

```
arr = np.array([{'a': 1}, [1, 2, 3], 'hello'], dtype=object)
print(arr)
```

---

7. Choosing the Right Data Type

Choosing the correct data type is essential for: - **Optimizing memory**: Using the smallest data type that fits your data. - **Improving performance**: Smaller types generally lead to faster operations. - **Ensuring precision**: Avoid truncating or losing important decimal places or values.

Summary:

- **NumPy arrays are homogeneous**, meaning all elements must be of the same type.
- Use `.astype()` to **change data types** and optimize memory and performance.

- The choice of data type affects **memory usage**, **performance**, and **precision**.
- Be mindful of **complex numbers** and **object data types**, which can increase memory usage and reduce performance.

Broadcasting in NumPy

Now, we'll explore how to make your code faster with **vectorization** and **broadcasting** in NumPy. These techniques are key to boosting performance in numerical operations by avoiding slow loops and memory inefficiency.

1. Why Loops Are Slow

In Python, loops are typically slow because: - **Python's interpreter**: Every iteration of the loop requires Python to interpret the loop logic, which is inherently slower than lower-level, compiled code. - **High overhead**: Each loop iteration in Python involves additional overhead for function calls, memory access, and index management.

While Python loops are convenient, they don't take advantage of the **optimized memory and computation** that libraries like **NumPy** provide.

Example: Looping Over Arrays in Python

```
import numpy as np

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

# Using a loop to square each element (slow)
for num in arr:
    result.append(num ** 2)

print(result)  # Output: [1, 4, 9, 16, 25]
```

This works, but it's not efficient. Each loop iteration is slow, especially with large datasets.

---

2. Vectorization: Fixing the Loop Problem

Vectorization allows you to perform operations on entire arrays **at once**, instead of iterating over elements one by one. This is made possible by **NumPy's optimized C-based backend** that executes operations in compiled code, which is much faster than Python loops.

Vectorized operations are also **more readable** and compact, making your code easier to maintain.

Example: Vectorized Operation

```
arr = np.array([1, 2, 3, 4, 5])
result = arr ** 2  # Vectorized operation
print(result)  # Output: [1 4 9 16 25]
```

Here, the operation is applied to all elements of the array simultaneously, and it's much faster than looping over the array.

Why is it Faster?

- **Low-level implementation**: NumPy's vectorized operations are implemented in **C** (compiled language), which is much faster than Python loops.
- **Batch processing**: NumPy processes multiple elements in parallel using **SIMD** (Single Instruction, Multiple Data), allowing multiple operations to be done simultaneously.

---

3. Broadcasting: Scaling Arrays Without Extra Memory

Broadcasting is a powerful feature of NumPy that allows you to perform operations on arrays of different shapes without creating copies. It "stretches" smaller arrays across larger arrays in a memory-efficient way, avoiding the overhead of creating multiple copies of data.

Example: Broadcasting with Scalar

Broadcasting is often used when you want to perform an operation on an array and a scalar value (e.g., add a number to all elements of an array).

```
arr = np.array([1, 2, 3, 4, 5])
result = arr + 10  # Broadcasting: 10 is added to all elements
print(result)  # Output: [11 12 13 14 15]
```

Here, the scalar `10` is "broadcast" across the entire array, and no extra memory is used.

---

4. Broadcasting with Arrays of Different Shapes

Broadcasting becomes more powerful when you apply operations on arrays of **different shapes**. NumPy automatically adjusts the shapes of arrays to make them compatible for element-wise operations, without actually copying the data.

Example: Broadcasting with Two Arrays

```
arr1 = np.array([1, 2, 3])
arr2 = np.array([10, 20, 30])

result = arr1 + arr2  # Element-wise addition
print(result)  # Output: [11 22 33]
```

NumPy automatically aligns the two arrays and performs element-wise addition, treating them as if they have the same shape.

Example: Broadcasting a 2D Array and a 1D Array

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

result = arr1 + arr2  # Broadcasting arr2 across arr1
print(result)
# Output:
# [[2 4 6]
#  [5 7 9]]
```

In this case, `arr2` is broadcast across the rows of `arr1`, adding `[1, 2, 3]` to each row.

How Broadcasting Works

1. **Dimensions must be compatible**: The size of the trailing dimensions of the arrays must be either the same or one of them must be 1.
2. **Stretching arrays**: If the shapes are compatible, NumPy stretches the smaller array to match the larger one, element-wise, without copying data.

---

5. Hands-on: Applying Broadcasting to Real-World Scenarios

Let's apply broadcasting to a real-world scenario: **scaling data** in machine learning.

Example: Normalizing Data Using Broadcasting

Imagine you have a dataset where each row represents a sample and each column represents a feature. You can **normalize** the data by subtracting the mean of each column and dividing by the standard deviation.

```
# Simulating a dataset (5 samples, 3 features)
data = np.array([[10, 20, 30],
                 [15, 25, 35],
                 [20, 30, 40],
                 [25, 35, 45],
                 [30, 40, 50]])

# Calculating mean and standard deviation for each feature (column)
mean = data.mean(axis=0)
std = data.std(axis=0)

# Normalizing the data using broadcasting
normalized_data = (data - mean) / std

print(normalized_data)
```

In this case, broadcasting allows you to subtract the mean and divide by the standard deviation for each feature without needing loops or creating copies of the data.

Summary:

- **Loops are slow** because Python's interpreter adds overhead, making iteration less efficient.
- **Vectorization** allows you to apply operations to entire arrays at once, greatly improving performance by utilizing NumPy's optimized C backend.
- **Broadcasting** enables operations between arrays of different shapes by automatically stretching the smaller array to match the shape of the larger array, without creating additional copies.
- **Real-world use**: Broadcasting can be used in data science tasks, such as **normalizing datasets**, without sacrificing memory or performance.

Built in Mathematical Functions in NumPy

Here are some common NumPy methods that are frequently used for statistical and mathematical operations:

1. `np.mean()` – Compute the **mean** (average) of an array.

```
np.mean(arr)
```

2. `np.std()` – Compute the **standard deviation** of an array.

```
np.std(arr)
```

3. `np.var()` – Compute the **variance** of an array.

```
np.var(arr)
```

4. `np.min()` – Compute the **minimum** value of an array.

```
np.min(arr)
```

5. `np.max()` – Compute the **maximum** value of an array.

```
np.max(arr)
```

6. `np.sum()` – Compute the **sum** of all elements in an array.

```
np.sum(arr)
```

7. `np.prod()` – Compute the **product** of all elements in an array.

```
np.prod(arr)
```

8. `np.median()` – Compute the **median** of an array.

```
np.median(arr)
```

9. `np.percentile()` – Compute the **percentile** of an array.

```
np.percentile(arr, 50)  # For the 50th percentile (median)
```

10. `np.argmin()` – Return the **index of the minimum** value in an array.

```
np.argmin(arr)
```

11. `np.argmax()` – Return the **index of the maximum** value in an array.

```
np.argmax(arr)
```

12. `np.corrcoef()` – Compute the **correlation coefficient** matrix of two arrays.

```
np.corrcoef(arr1, arr2)
```

13. `np.unique()` – Find the **unique elements** of an array.

```
np.unique(arr)
```

14. `np.diff()` – Compute the **n-th differences** of an array.

```
np.diff(arr)
```

15. `np.cumsum()` – Compute the **cumulative sum** of an array.

```
np.cumsum(arr)
```

16. `np.linspace()` – Create an array with **evenly spaced numbers** over a specified interval.

```
np.linspace(0, 10, 5)  # 5 numbers from 0 to 10
```

17. `np.log()` – Compute the **natural logarithm** of an array.

```
np.log(arr)
```

18. `np.exp()` – Compute the **exponential** of an array.

```
np.exp(arr)
```

These methods are used for performing mathematical and statistical operations with NumPy

Getting Started with Pandas

What is Pandas?

Pandas is a powerful, open-source Python library used for **data manipulation**, **cleaning**, and **analysis**. It provides two main data structures:

- **Series**: A one-dimensional labeled array
- **DataFrame**: A two-dimensional labeled table (like an Excel sheet or SQL table)

Pandas makes working with structured data fast, expressive, and flexible.

> - If you're working with tables, spreadsheets, or CSVs in Python—Pandas is your best friend.

Why Use Pandas?

Task	Without Pandas	With Pandas
Load a CSV	`open()` + loops	`pd.read_csv()`
Filter rows	Custom loop logic	`df[df["col"] > 5]`
Group & summarize	Manual aggregation	`df.groupby()`
Merge two datasets	Nested loops	`pd.merge()`

Pandas saves time, reduces code, and increases readability.

Installing Pandas

Install via pip:

```
pip install pandas
```

Or using conda (recommended if you're using Anaconda):

```
conda install pandas
```

Importing Pandas

```
import pandas as pd
```

> `pd` is the standard alias used by the data science community.

Pandas vs Excel vs SQL vs NumPy

Tool	Strengths	Weaknesses
Excel	Easy UI, great for small data	Slow, manual, not scalable
SQL	Efficient querying of big data	Not ideal for transformation logic
NumPy	Fast, low-level array operations	No labels, harder for tabular data
Pandas	Label-aware, fast, flexible	Slightly steep learning curve

Pandas bridges the gap between **NumPy performance** and **Excel-like usability**. Pandas is built on top of NumPy.

Summary

- Use Pandas when working with structured data.

- It’s the Swiss Army knife of data science.

Core Data Structures in Pandas

Pandas is built on **two main data structures**:

1. **Series** → One-dimensional (like a single column in Excel)
2. **DataFrame** → Two-dimensional (like a full spreadsheet or SQL table)

Series — 1D Labeled Array

A `Series` is like a list with **labels (index)**.

```
import pandas as pd

s = pd.Series([10, 20, 30, 40])
print(s)
```

Output:

```
0    10
1    20
2    30
3    40
dtype: int64
```

Notice the **automatic index**: 0, 1, 2, 3

You can also define a custom index:

```
s = pd.Series([10, 20, 30], index=["a", "b", "c"])
```

A `pandas.Series` may look similar to a Python dictionary because both store data with labels, but a Series offers much more. Unlike a dictionary, a Series supports fast vectorized operations, automatic index alignment during arithmetic, and handles missing data using `NaN`. It also allows both label-based and position-based access, and integrates seamlessly with the pandas ecosystem, especially DataFrames. While a dictionary is great for simple key–value storage, a Series is better suited for data analysis and manipulation tasks where performance, flexibility, and built-in functionality matter.

DataFrame — 2D Labeled Table

A `DataFrame` is like a **dictionary of Series** — multiple columns with labels.

```
data = {
    "name": ["Alice", "Bob", "Charlie"],
    "age": [25, 30, 35],
    "city": ["Delhi", "Mumbai", "Bangalore"]
}

df = pd.DataFrame(data)
print(df)
```

Output:

```
      name  age       city
0    Alice   25      Delhi
1      Bob   30     Mumbai
2  Charlie   35  Bangalore
```

Each column in a `DataFrame` is a `Series`.

Index and Labels

Every Series and DataFrame has an **Index** — it helps with:

- Fast lookups
- Aligning data
- Merging & joining
- Time series operations

```
df.index         # Row labels
df.columns       # Column labels
```

You can change them using:

```
df.index = ["a", "b", "c"]
df.columns = ["Name", "Age", "City"]
```

Why Learn These Well?

Most Pandas operations are built on these foundations:

- Selection
- Filtering
- Merging
- Aggregation

Understanding Series & DataFrames will make everything else easier.

Summary

- `Series` = 1D array with labels
- `DataFrame` = 2D table with rows + columns

- Both come with index and are the heart of Pandas

Creating DataFrames

Let's look at different ways to create a Pandas `DataFrame` — the core data structure you'll be using 90% of the time in data science.

From Python Lists

```
import pandas as pd

data = [
    ["Alice", 25],
    ["Bob", 30],
    ["Charlie", 35]
]

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

From Dictionary of Lists

Most common and readable format:

```
data = {
    "Name": ["Alice", "Bob", "Charlie"],
    "Age": [25, 30, 35]
}

df = pd.DataFrame(data)
```

Each **key becomes a column**, and each list is the **column data**.

From NumPy Arrays

```
import numpy as np


arr = np.array([[1, 2], [3, 4]])
df = pd.DataFrame(arr, columns=["A", "B"])
```

Make sure to provide column names!

From CSV Files

```
df = pd.read_csv("data.csv")
```

Use options like: - `sep` , `header` , `names` , `index_col` , `usecols` , `nrows` , etc.

Example:

```
pd.read_csv("data.csv", usecols=["Name", "Age"])
```

From Excel Files

```
df = pd.read_excel("data.xlsx")
```

You may need to install `openpyxl` or `xlrd` :

```
pip install openpyxl
```

From JSON

```
df = pd.read_json("data.json")
```

Can also read from a URL or string.

From SQL Databases

```
import sqlite3

conn = sqlite3.connect("mydb.sqlite")
df = pd.read_sql("SELECT * FROM users", conn)
```

From the Web (Example: CSV from URL)

```
url = "https://raw.githubusercontent.com/mwaskom/seaborn-data/master/tips.csv"
df = pd.read_csv(url)
```

EDA (Exploratory Data Analysis)

Exploratory Data Analysis (EDA) is an essential first step in any data science project.

It involves taking a deep look at the dataset to understand its structure, spot patterns, identify anomalies, and uncover relationships between variables. This process includes generating summary statistics, checking for missing or duplicate data, and creating visualizations like histograms, box plots, and scatter plots. The goal of EDA is to get a clear picture of what the data is telling you before applying any analysis or machine learning models.

By exploring the data thoroughly, you can make better decisions about how to clean, transform, and model it effectively.

Once your DataFrame is ready, run these to understand your data:

```
df.head()         # First 5 rows
df.tail()         # Last 5 rows
df.info()         # Column info: types, non-nulls
df.describe()     # Stats for numeric columns
df.columns        # List of column names
df.shape          # (rows, columns)
```

Summary

- You can create DataFrames from lists, dicts, arrays, files, web, and SQL
- Use `.head()`, `.info()`, `.describe()` to quickly explore any dataset

Data Selection & Filtering

Selecting the right rows and columns is *the first step* in analyzing any dataset. Pandas gives you several powerful ways to do this.

Selecting Rows & Columns

Selecting Columns

```
df["column_name"]        # Single column (as Series)
df[["col1", "col2"]]     # Multiple columns (as DataFrame)
```

Selecting Rows by Index

Use `.loc[]` (label-based) and `.iloc[]` (position-based):

```
df.loc[0]               # First row (by label)
df.iloc[0]              # First row (by position)
```

Select Specific Rows and Columns

```
df.loc[0, "Name"]       # Value at row 0, column 'Name'
df.iloc[0, 1]           # Value at row 0, column at index 1
```

You can also slice:

```
df.loc[0:2, ["Name", "Age"]]   # Rows 0 to 2, selected columns
df.iloc[0:2, 0:2]              # Rows and cols by index position
```

Fast Access: `.at` and `.iat`

These are optimized for **single element access**:

```
df.at[0, "Name"]       # Fast label-based access
df.iat[0, 1]           # Fast position-based access
```

Filtering with Conditions

Simple Condition

```
df[df["Age"] > 30]
```

Multiple Conditions (AND / OR)

```
df[(df["Age"] > 25) & (df["City"] == "Delhi")]
df[(df["Name"] == "Bob") | (df["Age"] < 30)]
```

> Use parentheses around each condition!

Querying with `.query()`

The `.query()` method in pandas lets you filter DataFrame rows using a string expression — it's a more readable and often more concise alternative to using boolean indexing.

This is a cleaner, SQL-like way to filter:

```
df.query("Age > 25 and City == 'Delhi'")
```

Dynamic column names:

```
col = "Age"
df.query(f"{col} > 25")
```

Here are the main **rules and tips** for using `.query()` in pandas:

---

1. Column names become variables

You can reference column names directly in the query string:

```
df.query("age > 25 and city == 'Delhi'")
```

---

2. String values must be in quotes

Use **single** or **double** quotes around strings in the expression:

```
df.query("name == 'Harry'")
```

If you have quotes inside quotes, mix them:

```
df.query('city == "Mumbai"')
```

---

3. Use backticks for column names with spaces or special characters

If a column name has spaces, use backticks (`` ` ``):

```
df.query("`first name` == 'Alice'")
```

---

4. You can use `@` to reference Python variables

To pass external variables into `.query()` :

```
age_limit = 30
df.query("age > @age_limit")
```

---

5. Logical operators

Use these: - `and` , `or` , `not` — instead of `&` , `|` , `~` - `==` , `!=` , `<` , `>` , `<=` , `>=`

Bad:

```
df.query("age > 30 & city == 'Delhi'")  # ❌
```

Good:

```
df.query("age > 30 and city == 'Delhi'")  # ☑
```

---

6. Chained comparisons

Just like Python:

```
df.query("25 < age <= 40")
```

---

7. Avoid using reserved keywords as column names

If you have a column named `class`, `lambda`, etc., you'll need to use backticks:

```
df.query("`class` == 'Physics'")
```

---

8. Case-sensitive

Column names and string values are case-sensitive:

```
df.query("City == 'delhi'")  # ❌ if actual value is 'Delhi'
```

---

9. `.query()` returns a copy, not a view

The result is a new DataFrame. Changes won't affect the original unless reassigned:

```
filtered = df.query("age < 50")
```

Summary

- Use `df[col]`, `.loc[]`, `.iloc[]`, `.at[]`, `.iat[]` to access data
- Filter with logical conditions or `.query()` for readable code
- Mastering selection makes the rest of pandas feel easy

Data Cleaning & Preprocessing

Real-world data is messy. Pandas gives us powerful tools to clean and transform data before analysis.

Handling Missing Values

Check for Missing Data

```
df.isnull()             # True for NaNs
df.isnull().sum()       # Count missing per column
```

Drop Missing Data

```
df.dropna()             # Drop rows with *any* missing values
df.dropna(axis=1)       # Drop columns with missing values
```

Fill Missing Data

In pandas, fillna is used to fill unknown values. ffill and bfill are methods used to fill missing values (like NaN, None, or pd.NA) by propagating values forward or backward.

```
df.fillna(0)                    # Replace NaN with 0
df["Age"].fillna(df["Age"].mean())  # Replace with mean
df.ffill()      # Forward fill
df.bfill()      # Backward fill
```

Detecting & Removing Duplicates

df.duplicated() returns a boolean Series where: True means that row is a duplicate of a previous row. False means it's the first occurrence (not a duplicate yet).

```
df.duplicated()          # True for duplicates
df.drop_duplicates()     # Remove duplicate rows
```

Check based on specific columns:

```
df.duplicated(subset=["Name", "Age"])
```

String Operations with `.str`

Works like vectorized string methods and returns a pandas Series:

```
df["Name"].str.lower() # Converts all names to lowercase.
df["City"].str.contains("delhi", case=False) # Checks if 'delhi' is in the city
name, case-insensitive.
df["Email"].str.split("@") # Outputs a pandas Series where each element is a list
```

```
of strings (the split parts). This is where a Python list comes into play, but the
outer object is still a pandas Series.
```

We can always chain methods like `.str.strip().str.upper()` for clean-up.

Type Conversions with `.astype()`

Convert column data types:

```
df["Age"] = df["Age"].astype(int)
df["Date"] = pd.to_datetime(df["Date"])
df["Category"] = df["Category"].astype("category")
```

Why is pd.to_datetime() special?

Unlike astype(), which works on simple data types (like integers, strings, etc.), pd.to_datetime() is designed to:

- Handle different date formats (e.g., “YYYY-MM-DD”, “MM/DD/YYYY”, etc.).
- Handle mixed types (e.g., some date strings, some NaT, or missing values).
- Convert integer timestamps (e.g., UNIX time) into datetime objects.
- Recognize timezones if provided.

Check data types:

```
df.dtypes
```

Applying Functions

`.apply()` → Apply any function to rows or columns

```
df["Age Group"] = df["Age"].apply(lambda x: "Adult" if x >= 18 else "Minor")
```

`.map()` → Element-wise mapping for Series

```
gender_map = {"M": "Male", "F": "Female"}
df["Gender"] = df["Gender"].map(gender_map)
```

`.replace()` → Replace specific values

```
df["City"].replace({"Del": "Delhi", "Mum": "Mumbai"})
```

Summary

- Use `isnull()`, `fillna()`, `dropna()` for missing data
- Clean text with `.str`, convert types with `.astype()`
- Use `apply()`, `map()`, `replace()` to transform your columns
- Data cleaning is where 80% of your time goes in real projects

Data Transformation

Once your data is clean, the next step is to **reshape, reformat, and reorder** it as needed for analysis. Pandas gives you plenty of flexible tools to do this.

Sorting & Ranking

Sort by Values

```
df.sort_values("Age")                    # Ascending sort
df.sort_values("Age", ascending=False)   # Descending
df.sort_values(["Age", "Salary"])        # Sort by multiple columns
```

df.sort_values(["Age", "Salary"]) sorts the DataFrame first by the "Age" column, and if there are ties (i.e., two or more rows with the same "Age"), it will sort by the "Salary" column.

Reset Index

If you want the index to start from 0 and be sequential, you can reset it using reset_index()

```
df.reset_index(drop=True, inplace=True)  # Reset the index and drop the old index
```

Sort by Index

```
df.sort_index()
```

The df.sort_index() function is used to sort the DataFrame based on its index values. If the index is not in a sequential order (e.g., you have dropped rows or performed other operations that change the index), you can use sort_index() to restore it to a sorted order. ### Ranking The .rank() function in pandas is used to assign ranks to numeric values in a column, like scores or points. By default, it gives the average rank to tied values, which can result in decimal numbers. For example, if two people share the top score, they both get a rank of 1.5. You can customize the ranking behavior using the method parameter. One useful option is method='dense', which assigns the same rank to ties but doesn't leave gaps in the ranking sequence. This is helpful when you want a clean, consecutive ranking system without skips.

```
df["Rank"] = df["Score"].rank()                 # Default: average method
df["Rank"] = df["Score"].rank(method="dense")   # 1, 2, 2, 3
```

Renaming Columns & Index

```
df.rename(columns={"oldName": "newName"}, inplace=True)
df.rename(index={0: "row1", 1: "row2"}, inplace=True)
```

To rename all columns:

```
df.columns = ["Name", "Age", "City"]
```

Changing Column Order

Just pass a new list of column names:

```
df = df[["City", "Name", "Age"]]   # Reorder as desired
```

You can also move one column to the front:

```
cols = ["Name"] + [col for col in df.columns if col != "Name"]
df = df[cols]
```

Summary

- Sort, rank, and rename to prepare your data
- Reordering and reshaping are key for EDA and visualization

Reshaping Data using Melt and Pivot

`melt()` — Wide to Long

The `melt()` method in Pandas is used to **unpivot** a DataFrame from wide format to long format. In other words, it takes columns that represent different variables and combines them into key-value pairs (i.e., long-form data).

When to Use `melt()`:

- When you have a DataFrame where each row is an observation, and each column represents a different variable or measurement, and you want to reshape the data into a longer format for easier analysis or visualization.

Syntax:

```
df.melt(id_vars=None, value_vars=None, var_name=None, value_name="value",
col_level=None)
```

Parameters:

- **`id_vars`**: The columns that you want to keep fixed (these columns will remain as identifiers).
- **`value_vars`**: The columns you want to unpivot (the ones you want to "melt" into a single column).
- **`var_name`**: The name to use for the new column that will contain the names of the melted columns (default is `'variable'`).
- **`value_name`**: The name to use for the new column that will contain the values from the melted columns (default is `'value'`).
- **`col_level`**: Used for multi-level column DataFrames.

Example:

Use this code to generate the Dataframe

```
import pandas as pd

# Sample DataFrame
data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Math': [85, 78, 92],
    'Science': [90, 82, 89],
    'English': [88, 85, 94]
}

df = pd.DataFrame(data)

# Display the DataFrame
print(df)
```

Let's say we have the following DataFrame in a wide format:

Name	Math	Science	English
Alice	85	90	88
Bob	78	82	85
Charlie	92	89	94

Using `melt()`:

If we want to "melt" the DataFrame so that each row represents a student-subject pair, we can do:

```
df.melt(id_vars=["Name"], value_vars=["Math", "Science", "English"], var_name="Subj
ect", value_name="Score")
```

This will result in the following long-format DataFrame:

Name	Subject	Score
Alice	Math	85

Name	Subject	Score
Alice	Science	90
Alice	English	88
Bob	Math	78
Bob	Science	82
Bob	English	85
Charlie	Math	92
Charlie	Science	89
Charlie	English	94

Explanation:

- `id_vars=["Name"]` : We keep the “Name” column as it is because it’s the identifier.
- `value_vars=["Math", "Science", "English"]` : These are the columns we want to melt.
- `var_name="Subject"` : The new column containing the names of the subjects.
- `value_name="Score"` : The new column containing the scores.

Why Use `melt()` ?

- **Data normalization**: Helps in transforming data for statistical modeling and data visualization.
- **Pivot tables**: Many times, plotting functions or statistical models work better with long-format data.

This is useful for converting columns into rows — perfect for plotting or tidy data formats.

`pivot()` — Long to Wide

The `pivot()` function in Pandas is used to **reshape** data, specifically to **turn long-format data into wide-format data**. This is the reverse operation of `melt()`.

How it works:

- `pivot()` takes a **long-format DataFrame** and **turns it into a wide-format DataFrame** by specifying which columns will become the new columns, the rows, and the values.

Syntax:

```
df.pivot(index=None, columns=None, values=None)
```

Parameters:

- `index` : The column whose unique values will become the rows of the new DataFrame.
- `columns` : The column whose unique values will become the columns of the new DataFrame.
- `values` : The column whose values will fill the new DataFrame. These will become the actual data (values in the table).

Example:

Suppose we have the following long-format DataFrame:

Name	Subject	Score
Alice	Math	85
Alice	Science	90
Alice	English	88
Bob	Math	78
Bob	Science	82

Name	Subject	Score
Bob	English	85
Charlie	Math	92
Charlie	Science	89
Charlie	English	94

Using `pivot()` to reshape it into wide format:

```
df.pivot(index="Name", columns="Subject", values="Score")
```

Resulting DataFrame:

Name	English	Math	Science
Alice	88	85	90
Bob	85	78	82
Charlie	94	92	89

Explanation:

- `index="Name"` : The unique values in the “Name” column will become the rows in the new DataFrame.
- `columns="Subject"` : The unique values in the “Subject” column will become the columns in the new DataFrame.
- `values="Score"` : The values from the “Score” column will populate the table.

Why use `pivot()` ?

1. **Better data structure**: It makes data easier to analyze when you have categories that you want to split into multiple columns.
2. **Easier visualization**: Often, you want to represent data in a format where categories are split across columns (for example, when creating pivot tables for reporting).

3. **Aggregating data**: You can perform aggregations (like `sum`, `mean`, etc.) to group values before pivoting.

Important Notes:

1. **Duplicate Entries**: If you have multiple rows with the same combination of `index` and `columns`, **pivot()** will raise an error. In such cases, you should use `pivot_table()` (which can handle duplicate entries by aggregating them).

Example of `pivot_table()` to handle duplicates:

Suppose the DataFrame is like this (with duplicate entries):

Name	Subject	Score
Alice	Math	85
Alice	Math	80
Alice	Science	90
Bob	Math	78
Bob	Math	82

We can use `pivot_table()` to aggregate values (e.g., taking the **mean** for duplicate entries):

```
df.pivot_table(index="Name", columns="Subject", values="Score", aggfunc="mean")
```

Resulting DataFrame:

Name	Math	Science
Alice	82.5	90
Bob	80	NaN

In this case, the **Math** score for Alice is averaged (85 + 80) / 2 = 82.5. If a cell is empty, it means there was no value for that combination.

Summary:

- Use `melt()` to go long, `pivot()` to go wide
- **`pivot()`** is used to turn long-format data into wide-format by spreading unique column values into separate columns.
- If there are **duplicate values** for a given combination of `index` and `columns`, you should use **`pivot_table()`** with an aggregation function to handle the duplicates.

Aggregation & Grouping

Grouping and aggregating helps you **summarize your data** — like answering:

> "What's the average salary *per department*?"
> "How many users joined the Gym *per month*?"

`.groupby()` Function

df.groupby() is used to group rows of a DataFrame based on the values in one or more columns, which allows you to then perform aggregate functions (like sum(), mean(), count(), etc.) on each group. Consider this DataFrame:

```
df = pd.DataFrame({
    "Department": ["HR", "HR", "IT", "IT", "Marketing", "Marketing", "Sales", "Sale
s"],
    "Team": ["A", "A", "B", "B", "C", "C", "D", "D"],
    "Gender": ["M", "F", "M", "F", "M", "F", "M", "F"],
    "Salary": [85, 90, 78, 85, 92, 88, 75, 80],
    "Age": [23, 25, 30, 22, 28, 26, 21, 27],
    "JoinDate": pd.to_datetime([
        "2020-01-10", "2020-02-15", "2021-03-20", "2021-04-10",
        "2020-05-30", "2020-06-25", "2021-07-15", "2021-08-01"
    ])
})
```

```
df.groupby("Department")["Salary"].mean()
```

This says:
> “Group by Department, then calculate average Salary for each group.”

Common Aggregation Functions

```
df.groupby("Team")["Salary"].mean()     # Average per team
df.groupby("Team")["Salary"].sum()      # Total score
df.groupby("Team")["Salary"].count()    # How many entries
df.groupby("Team")["Salary"].min()
df.groupby("Team")["Salary"].max()
```

To group by multiple columns:

```
df.groupby(["Team", "Gender"])["Salary"].mean()
```

Custom Aggregations with `.agg()`

Apply **multiple functions** at once like this:

```
df.groupby("Team")["Salary"].agg(["mean", "max", "min"])
```

In pandas, .agg and .aggregate are exactly the same — they’re aliases for the same method

Name your own functions:

```
df.groupby("Team")["Salary"].agg(
    avg_score="mean",
```

```
    high_score="max"
)
```

Apply different functions to different columns:

```
df.groupby("Team").agg({
    "Salary": "mean",
    "Age": "max"
})
```

Transform vs Aggregate vs Filter

Operation	Returns	When to Use
`.aggregate()`	Single value per group	Summary (like mean)
`.transform()`	Same shape as original	Add new column based on group
`.filter()`	Subset of rows	Keep/discard whole groups

`.transform()` Example:

```
df["Team Avg"] = df.groupby("Team")["Salary"].transform("mean")
```

Now each row gets its **team average** — great for comparisons!

`.filter()` Example:

```
df.groupby("Team").filter(lambda x: x["Salary"].mean() > 80)
```

Only keeps teams with average score > 80.

Summary

- `.groupby()` helps you summarize large datasets by category
- Use `mean()`, `sum()`, `count()`, `.agg()` for custom metrics
- `.transform()` adds values back to original rows
- `.filter()` keeps only groups that meet conditions

Merging & Joining Data

Often, data is split across multiple tables or files. Pandas lets you **combine** them just like SQL — or even more flexibly!

Sample DataFrames

```
employees = pd.DataFrame({
    "EmpID": [1, 2, 3],
    "Name": ["Alice", "Bob", "Charlie"],
    "DeptID": [10, 20, 30]
})

departments = pd.DataFrame({
    "DeptID": [10, 20, 40],
    "DeptName": ["HR", "Engineering", "Marketing"]
})
```

Merge Like SQL: `pd.merge()`

Inner Join (default)

```
pd.merge(employees, departments, on="DeptID")
```

Returns only matching DeptIDs:

EmpID	Name	DeptID	DeptName
1	Alice	10	HR
2	Bob	20	Engineering

Left Join

```
pd.merge(employees, departments, on="DeptID", how="left")
```

Keeps all employees, fills `NaN` where no match.

Right Join

```
pd.merge(employees, departments, on="DeptID", how="right")
```

Keeps all departments, even if no employee.

Outer Join

```
pd.merge(employees, departments, on="DeptID", how="outer")
```

Includes *all* data, fills missing with `NaN`.

Concatenating DataFrames

Use `pd.concat()` to **stack** datasets either vertically or horizontally.

Vertical (rows)

```
df1 = pd.DataFrame({"Name": ["Alice", "Bob"]})
df2 = pd.DataFrame({"Name": ["Charlie", "David"]})

pd.concat([df1, df2])
```

Horizontal (columns)

```
df1 = pd.DataFrame({"ID": [1, 2]})
df2 = pd.DataFrame({"Score": [90, 80]})

pd.concat([df1, df2], axis=1)
```

> Make sure indexes align when using `axis=1`

When to Use What?

Use Case	Method
SQL-style joins (merge keys)	`pd.merge()` or `.join()`
Stack datasets vertically	`pd.concat([df1, df2])`
Combine different features side-by-side	`pd.concat([df1, df2], axis=1)`
Align on index	`.join()` or merge with `right_index=True`

Summary

- Use `merge()` like SQL joins (`inner` , `left` , `right` , `outer`)
- Use `concat()` to stack DataFrames (rows or columns)
- Handle mismatched keys and indexes with care
- Merging and joining are essential for real-world projects

Reading & Writing Files in Pandas

CSV Files

Read CSV

```
df = pd.read_csv("data.csv")
```

Options:

```
pd.read_csv("data.csv", usecols=["Name", "Age"], nrows=10)
```

Write CSV

```
df.to_csv("output.csv", index=False)
```

Excel Files

Read Excel

```
df = pd.read_excel("data.xlsx")
```

Options:

```
pd.read_excel("data.xlsx", sheet_name="Sales")
```

Write Excel

```
df.to_excel("output.xlsx", index=False)
```

Multiple sheets:

```
with pd.ExcelWriter("report.xlsx") as writer:
    df1.to_excel(writer, sheet_name="Summary", index=False)
    df2.to_excel(writer, sheet_name="Details", index=False)
```

JSON Files

Read JSON

```
df = pd.read_json("data.json")
```

Summary

- `read_*` and `to_*` methods for CSV, Excel, JSON
- Use `sheet_name` for Excel