Introduction to Data Science

What is Data Science?

Welcome to this course on Data Science. If you are active on Fb/Insta/X, **share your Data Science journey on these platforms and tag me**. I would love to see your progress.

I am so happy you decided to learn Data Science with me in my style. Lets step back and talk a bit about why do we need data science. What kind of tasks we will do as a data scientist in an organization

What is Data Science?

At its core, **Data Science** is the field of study that uses **mathematics, statistics, programming, and domain knowledge** to extract **meaningful insights** from data. Well that sounds like a bookish definition which you are free to write in your semester exams but Data Science pretty much blends various techniques from different disciplines to analyze large amounts of information and solve real-world problems.

Simple Definition:

Let me give you a very simple non fancy definition of Data Science

> Data Science is the process of collecting, cleaning, analyzing, and interpreting data to make informed decisions.

Why is Data Science Important?

In today’s world, **data is everywhere**—from your online shopping habits to the sensors in smart devices. Companies use this data to:

- **Make better decisions** (e.g., which product to launch next).
- **Predict outcomes** (e.g., weather forecasts or stock prices).
- **Automate processes** (e.g., self-driving cars).
- **Personalize experiences** (e.g., Netflix recommendations, YouTube recommendations).

Key Steps in Data Science:

1. **Data Collection** – Gathering raw data from various sources (websites, databases, IoT devices, etc.).
2. **Data Cleaning** – Fixing errors and handling missing values (this takes up **80% of a data scientist’s time**).
3. **Data Analysis** – Using statistical methods to find patterns and insights.
4. **Model Building** – Applying **machine learning** to make predictions.
5. **Interpretation & Communication** – Presenting findings in a clear way to help decision-making.

Where Do We See Data Science in Action?

- **Healthcare:** Predicting disease outbreaks and improving diagnoses.
- **E-commerce:** Personalized product recommendations.
- **Finance:** Detecting fraudulent transactions by recognizing patterns.
- **Entertainment:** Content recommendations on platforms like YouTube and Netflix.

Why Should You Care About Data Science?

- **High Demand:** There's a global shortage of skilled data scientists.
- **Great Pay:** The average salary of a data scientist in the US is **$120,000+ per year**; in India, ₹10-30 LPA for experienced professionals.
- **Future-Proof Career:** It powers everything from AI to business strategy—this field is only growing.

Note: The content you just read will be supplied to you as a downloadable PDF. Since I am writing a summarized version of the video content which will serve as revision notes, I recommend you try to read them along with watching the videos. Thanks and see you in the next lecture!

Data Science Lifecycle

The **Data Science Lifecycle** refers to the structured process used to extract insights from data. It involves several stages, from gathering raw data to delivering actionable insights. Here is a breakdown of each step:

1. Problem Definition

> Understanding the problem you want to solve.

- Identify business objectives and define the question to answer.
- Later, we will do a very interesting project called "Coders of Delhi" where we start by understanding the business objective.
- Example: "Can we predict customer churn?" or "What factors drive sales?". In "Coders of Delhi" project, the problem is how to find potential friends of a given person in a social network

Key Activities:

- Collaborate with stakeholders.
- Define success metrics.
- Set project goals and deliverables.

2. Data Collection

> Gathering relevant data from multiple sources.

- Sources may include databases, APIs, web scraping, or third-party datasets. Sometimes if this step is taken care of by another team or it's a data dump given by another team, we don't care where it came from.

Key Activities in data collection:

- Identify data sources (structured vs. unstructured).
- Collect data using SQL, Python, or automated pipelines.
- Ensure data relevance and completeness.

---

3. Data Cleaning (Data Preprocessing)

> Preparing raw data for analysis.

- This step addresses **missing values, duplicates, and inconsistencies**.

Key Activities:

- Handle missing or incorrect data.
- Standardize formats and remove duplicates.
- Manage outliers and inconsistencies.

Fun Fact: Data scientists spend **80% of their time** cleaning data!

---

4. Data Exploration (EDA – Exploratory Data Analysis)

> Analyzing data patterns and relationships.

- Understand data distributions and detect anomalies using visualizations.

Key Activities:

- Summarize data using statistics (mean, median, etc.).

- Visualize patterns (using **Matplotlib**, **Seaborn**, etc.).
- Identify correlations and outliers. (correlation is how two variables move in relation to each other and outlier is a data point that stands out as unusually different from the rest. Eg. A 190 Kg heavyweight person is an outlier)

---

5. Model Building

> Creating and training machine learning models.

- Use algorithms to predict outcomes or classify data.

Key Activities:

- Choose appropriate models (e.g., regression, decision trees, neural networks).
- Split data into training and testing sets.
- Train and fine-tune models.

Common Tools: Scikit-learn, TensorFlow, PyTorch.

---

6. Model Evaluation

> Measuring model performance and accuracy.

- Evaluate models using metrics to ensure reliability.

Key Activities:

- Use performance metrics (e.g., accuracy, RMSE, ROC curve) to answer questions like - *"How often is my model correct?"*
- Perform cross-validation for robustness. Train using some part of the data and test using some part and average out the accuracy. We will study this in detail later
- Compare multiple models for best outcomes.

Key Metrics:

- Classification: Accuracy, Precision, Recall, F1-Score.

- Regression: RMSE, R-squared.

---

7. Deployment

> Integrating the model into production systems.

- Deliver actionable results through APIs or dashboards.

Key Activities:

- Package the model for deployment (Usually done using web frameworks like Flask, and FastAPI).
- Automate pipelines for continuous learning (MLOps).
- Monitor performance post-deployment.

---

8. Communication & Reporting

> Sharing insights with stakeholders. At the end of the day the ML model solves a problem and proper reporting it to the concerned department is very important

Key Activities:

- Create dashboards
- Present findings clearly and concisely.
- Document the process and results.

---

9. Maintenance & Iteration

> Keeping the model accurate and up-to-date.

Key Activities:

- Monitor model performance.

- Update models with new data.
- Refine features and parameters.

Summary

The **Data Science Lifecycle** is a continuous, iterative process involving:

1. Problem Definition
2. Data Collection
3. Data Cleaning
4. Data Exploration
5. Model Building
6. Model Evaluation
7. Deployment
8. Communication & Reporting
9. Maintenance & Iteration

By following this lifecycle, data scientists transform raw data into meaningful insights that drive better decision-making.

Data Science Tools

I am enjoying teaching so far. When working in data science, the right tools make your work easier, faster, and more efficient. When I started my data science journey at IIT Kharagpur, I used to code using Pycharm and regular Python installation. I knew about Jupyter but wasn't familiar with its capabilities. From writing code to visualizing data, there are many options to choose from. Here is a breakdown of popular data science tools and why **Anaconda** with **Jupyter Notebook** is an excellent choice for beginners and advanced users.

How to run Python programs

The easiest way to run Python programs is by installing VS Code and using pip to install packages but we will use Anaconda and Jupyter notebooks

1. Jupyter Notebook (with Anaconda Distribution)

> An open-source web application that allows you to create and share documents with live code, equations, visualizations, and text.

Why Use Anaconda with Jupyter Notebook?

- **User-Friendly:** Interactive coding and easy-to-follow outputs, perfect for beginners.
- **All-in-One Package:** Anaconda includes essential libraries (NumPy, Pandas, Matplotlib + 1,500 other popular packages) pre-installed.
- **Ideal for Data Science:** Quick prototyping, data visualization, and exploratory analysis.
- **Environment Management:** Easily create isolated environments to manage package dependencies.

Common Use Cases:

- Data exploration and visualization
- Machine learning experiments
- Sharing research and reports

Installation: Althought we will do a detailed installation of Anaconda in the later sections, you can download and install the Anaconda distribution from anaconda.com. It includes Jupyter Notebook by default.

Command to Launch Jupyter Notebook:

```
jupyter notebook
```

Don't worry, we will do all these things step by step in the next section

---

2. Google Colab

> A free, cloud-based Jupyter Notebook environment provided by Google.

Why Use Google Colab?

- **Free GPU/TPU Access:** Great for deep learning without requiring expensive hardware.
- **Cloud-Based:** No local setup—just log in and start coding.
- **Collaboration:** Share notebooks via links for easy collaboration.

Common Use Cases:

- Machine learning and deep learning projects
- Quick experiments without local setup
- Collaborative projects

Access: Use Google Colab directly in your browser at colab.research.google.com.

---

3. VS Code (Visual Studio Code)

> A lightweight and powerful code editor by Microsoft with robust extensions.

Why Use VS Code?

- **Customizable:** Extensive extensions for Python and data science (e.g., Python extension by Microsoft).
- **Integrated Jupyter Support:** You can run Jupyter Notebooks directly in VS Code.
- **Debugging Tools:** Advanced debugging capabilities.

Common Use Cases:

- Large-scale data science projects
- Working with multiple languages (Python, R, etc.)
- Integrated development (data pipelines, APIs)

Installation: Can be downloaded from code.visualstudio.com but we will install and use Anaconda distribution throughout this Data Science course.

---

4. PyCharm

> A powerful, professional IDE for Python development by JetBrains.

Why Use PyCharm?

- **Professional Features:** Advanced code navigation, refactoring, and debugging.
- **Environment Management:** Virtual environment and package management built-in.
- **Scientific Mode:** Built-in support for Jupyter notebooks.

Common Use Cases:

- Large, production-level data science projects
- Building Python-based machine learning applications

Installation: Download from jetbrains.com/pycharm.

---

5. Cursor AI

> An AI-powered code editor designed for enhanced productivity with machine learning assistance.

Why Use Cursor AI?

- **AI Integration:** Code suggestions and completions for faster development.
- **Context-Aware:** Understands complex data science workflows.
- **Collaborative:** Works well with team-based projects.

Common Use Cases:

- Assisted coding for data science
- Accelerating research and prototyping
- Team collaboration

Access: You can visit cursor.so to download cursor AI but I don’t recommend using it just yet. Its important to understand the basics of data science and programming before you use such AI assistants

---

Which Tool Should You Choose?

Tool	Best For	Key Advantage
Jupyter Notebook	Interactive analysis, education	Easy to use and visualize data
Google Colab	Deep learning, cloud-based projects	Free GPU/TPU and no local setup
VS Code	Large projects, debugging	Lightweight with advanced features
PyCharm	Enterprise-level, complex applications	Professional IDE with deep features
Cursor AI	AI-assisted coding, productivity	AI-enhanced code suggestions
Spyder	Academic research, scientific computing	MATLAB-like interface

Recommendation: If you’re starting out or want a hassle-free experience, **Anaconda with Jupyter Notebook** is the best choice. For advanced AI and big data projects, **Google Colab** is an excellent free alternative. For robust, large-scale development, **VS Code** or **PyCharm** provides advanced capabilities. Since we are just starting our learning journey, I will be using Anaconda and Jupyter for the most part of this course

---

Summary

1. **Anaconda + Jupyter Notebook:** Best for beginners and interactive analysis.
2. **Google Colab:** Ideal for cloud-based work and deep learning.

3. **VS Code:** Perfect for integrated, large-scale projects.
4. **PyCharm:** A professional-grade IDE for Python development.
5. **Cursor AI:** AI-assisted productivity for fast development.

Choosing the right tool depends on your project size, complexity, and hardware needs. For most data science workflows, **Anaconda with Jupyter Notebook** offers the best balance of simplicity, flexibility, and power.

Python Refresher

Why choose Python

In this section, we will learn why Python is a popular and powerful choice for data science.

Variables, Data Types, and Typecasting

In this section, we will learn about variables, data types, and typecasting in Python to store and convert data effectively. ## Variables

- Containers for storing data values.
- No need to declare data type explicitly.

```
name = "Alice"
age = 25
is_student = True
```

Data Types

Type	Example	Description
`int`	`10` , `-5`	Integer numbers
`float`	`3.14` , `-0.5`	Decimal numbers
`str`	`"hello"`	Text (string)
`bool`	`True` , `False`	Boolean values
`list`	`[1, 2, 3]`	Ordered, mutable collection
`tuple`	`(1, 2, 3)`	Ordered, immutable collection

Type	Example	Description
`dict`	`{"a": 1}`	Key-value pairs

Typecasting (Type Conversion)

- Convert data from one type to another using built-in functions:

```
# str to int
x = int("10")      # 10

# int to str
y = str(25)        # "25"

# float to int
z = int(3.9)       # 3 (truncates, not rounds)

# list from string
lst = list("abc")  # ['a', 'b', 'c']
```

Quick Tips

- Use `type(variable)` to check a variable's data type.
- Typecasting errors can happen if the value isn't compatible:

  ```
  int("hello")  # ValueError
  ```

String and String Methods

In this section, we will learn about strings and string methods in Python to work with and manipulate text data.

What is a String?

- A string is a sequence of characters enclosed in single (`'`) or double (`"`) quotes.

```
name = "Alice"
greeting = 'Hello'
```

Multiline Strings

- Use triple quotes (`'''` or `"""`) for multiline text.

```
message = """This is
a multiline
string."""
```

String Indexing and Slicing

- Indexing starts at 0.

```
text = "Python"
text[0]    # 'P'
text[-1]   # 'n' (last character)
text[0:2]  # 'Py'
text[:3]   # 'Pyt'
text[3:]   # 'hon'
```

String Immutability

- Strings cannot be changed after creation.

```
text[0] = 'J'  # Error
```

Common String Methods

Method	Description
`str.lower()`	Converts to lowercase
`str.upper()`	Converts to uppercase
`str.strip()`	Removes leading/trailing spaces
`str.replace(old, new)`	Replaces substring
`str.split(sep)`	Splits string into a list
`str.join(list)`	Joins list into string
`str.find(sub)`	Returns index of first occurrence
`str.count(sub)`	Counts occurrences of substring
`str.startswith(prefix)`	Checks if string starts with value
`str.endswith(suffix)`	Checks if string ends with value
`str.isdigit()`	Checks if all chars are digits
`str.isalpha()`	Checks if all chars are letters
`str.isalnum()`	Checks if all chars are letters/digits

Examples

```python
"hello".upper()            # 'HELLO'
" Hello ".strip()          # 'Hello'
"hello world".split()      # ['hello', 'world']
"-".join(["2025", "04", "14"])  # '2025-04-14'
"python".find("th")        # 2
```

String Formatting (f-strings)

```
name = "Alice"
age = 30
f"Hello, {name}. You are {age} years old."
# 'Hello, Alice. You are 30 years old.'
```

Operators in Python

In this section, we will learn about different types of operators in Python and how they are used in expressions.
## 1. Arithmetic Operators Used for basic mathematical operations.

Operator	Description	Example (`a=10, b=5`)
`+`	Addition	`a + b # 15`
`-`	Subtraction	`a - b # 5`
`*`	Multiplication	`a * b # 50`
`/`	Division	`a / b # 2.0`
`//`	Floor Division	`a // b # 2`
`%`	Modulus	`a % b # 0`
`**`	Exponentiation	`a ** b # 100000`

2. Comparison Operators

Compare values and return `True` or `False`.

Operator	Description	Example (`a=10, b=5`)
`==`	Equal to	`a == b # False`
`!=`	Not equal to	`a != b # True`

Operator	Description	Example (`a=10, b=5`)
`>`	Greater than	`a > b # True`
`<`	Less than	`a < b # False`
`>=`	Greater or equal	`a >= b # True`
`<=`	Less or equal	`a <= b # False`

3. Logical Operators

Used to combine conditional statements.

Operator	Description	Example (`x=True, y=False`)
`and`	Both True	`x and y # False`
`or`	Either True	`x or y # True`
`not`	Negation	`not x # False`

4. Bitwise Operators

Perform bit-level operations.

Operator	Description	Example (`a=5, b=3`)
`&`	AND	`a & b # 1`
\\\|	OR	a \\\| b # 7

5. Assignment Operators

Used to assign values to variables.

Operator	Example (`a=10`)	Equivalent to
`=`	`a = 5`	`a = 5`

Operator	Example (`a=10`)	Equivalent to
`+=`	`a += 5`	`a = a + 5`
`-=`	`a -= 5`	`a = a - 5`
`*=`	`a *= 5`	`a = a * 5`
`/=`	`a /= 5`	`a = a / 5`
`//=`	`a //= 5`	`a = a // 5`
`%=`	`a %= 5`	`a = a % 5`
`**=`	`a **= 5`	`a = a ** 5`

6. Membership & Identity Operators

Check for presence and object identity.

Operator	Description	Example (`lst=[1,2,3]` , `x=2`)
`in`	Present in sequence	`x in lst # True`
`not in`	Not present	`x not in lst # False`
`is`	Same object	`a is b # False`
`is not`	Different object	`a is not b # True`

Taking input from the user

In this section, we will learn how to take input from the user in Python and use it in our programs. ## Basic Usage

```
name = input("Enter your name: ")
print("Hello", name)
```

Note: `input()` always returns a string.

Type Conversion

You can always convert the string output of input() function to other supported data types

```
age = int(input("Enter your age: "))
price = float(input("Enter the price: "))
```

Operator Precedence

Python follows **PEMDAS** (Parentheses, Exponents, Multiplication/Division, Addition/Subtraction). The order of operations in Python is:

1. **Parentheses** `()` – Highest precedence, operations inside parentheses are evaluated first.
2. **Exponents** `**` – Power calculations (e.g., `2 ** 3` → 8).
3. **Multiplication** `*`, **Division** `/`, **Floor Division** `//`, **Modulus** `%` – Evaluated from left to right.
4. **Addition** `+`, **Subtraction** `-` – Evaluated from left to right.

Example:

```
result = 10 + 2 * 3  # Multiplication happens first: 10 + (2 * 3)
print(result)

result = (10 + 2) * 3  # Parentheses first: (10 + 2) * 3 = 36
print(result)

result = 2 ** 3 ** 2  # Right-to-left exponentiation: 2 ** (3 **
print(result)
```

If else statements

In Python, conditional statements (`if`, `elif`, and `else`) are used to control the flow of a program based on conditions. These are essential in data science for handling different scenarios in data processing, decision-making, and logic execution.

Basic `if` Statement

The `if` statement allows you to execute a block of code only if a condition is `True`.

```python
x = 10
if x > 5:
    print("x is greater than 5")
```

Explanation:

- The condition `x > 5` is checked.
- If `True`, the indented block under `if` runs.
- If `False`, nothing happens.

`if-else` Statement

The `else` block executes when the `if` condition is `False`.

```python
x = 3
if x > 5:
    print("x is greater than 5")
else:
    print("x is not greater than 5")
```

Explanation:

- If `x > 5`, it prints the first message.
- Otherwise, the `else` block executes.

`if-elif-else` Statement

When multiple conditions need to be checked sequentially, use `elif` (short for "else if").

```python
x = 5
if x > 10:
    print("x is greater than 10")
elif x > 5:
    print("x is greater than 5 but not more than 10")
elif x == 5:
    print("x is exactly 5")
else:
    print("x is less than 5")
```

Explanation:

- The conditions are checked from top to bottom.
- The first `True` condition executes, and the rest are skipped.

Using `if-else` in Data Science

Conditional statements are widely used in data science for filtering, cleaning, and decision-making.

Example: Categorizing Data

```python
age = 25
if age < 18:
    category = "Minor"
elif age < 65:
    category = "Adult"
else:
    category = "Senior Citizen"

print("Category:", category)
```

Example: Applying Conditions on Pandas DataFrame

```
import pandas as pd

data = {'Name': ['Alice', 'Bob', 'Charlie'], 'Score': [85, 40, 75
df = pd.DataFrame(data)

df['Result'] = df['Score'].apply(lambda x: 'Pass' if x >= 50 else
print(df)
```

Summary

- `if` : Executes if the condition is `True` .
- `if-else` : Adds an alternative block if the condition is `False` .
- `if-elif-else` : Handles multiple conditions.
- Useful in data science for logic-based decision-making.

Match Case Statements

The `match-case` statement, introduced in Python 3.10, provides pattern matching similar to `switch` statements in other languages.

Syntax

```
def http_status(code):
    match code:
        case 200:
            return "OK"
        case 400:
            return "Bad Request"
        case 404:
            return "Not Found"
        case 500:
            return "Internal Server Error"
        case _:
```

```python
            return "Unknown Status"

print(http_status(200))  # Output: OK
print(http_status(404))  # Output: Not Found
```

Features:

- The `_` (underscore) acts as a default case.
- Patterns can include literals, variable bindings, and even structural patterns.

Example: Matching Data Structures

Lets try to match a tuple using Match-Case statements

```python
point = (3, 4)

match point:
    case (0, 0):
        print("Origin")
    case (x, 0):
        print(f"X-Axis at {x}")
    case (0, y):
        print(f"Y-Axis at {y}")
    case (x, y):
        print(f"Point at ({x}, {y})")
```

String Formatting and F-Strings

String is arguably the most used immutable data types in Python. Python provides multiple ways to format strings, including the `format()` method and f-strings (introduced in Python 3.6).

1. Using `format()`

```
name = "Alice"
age = 25
print("My name is {} and I am {} years old.".format(name, age))
# Output: My name is Alice and I am 25 years old.
```

Positional and Keyword Arguments

```
print("{0} is learning {1}".format("Alice", "Python"))  # Using p
print("{name} is learning {language}".format(name="Alice", langua
```

2. Using f-Strings (Recommended)

F-strings provide a cleaner and more readable way to format strings.

```
name = "Alice"
age = 25
print(f"My name is {name} and I am {age} years old.")
# Output: My name is Alice and I am 25 years old.
```

Expressions Inside f-Strings

```
a = 5
b = 10
print(f"Sum of {a} and {b} is {a + b}")
# Output: Sum of 5 and 10 is 15
```

Formatting Numbers

```
pi = 3.14159
print(f"Pi rounded to 2 decimal places: {pi:.2f}")
# Output: Pi rounded to 2 decimal places: 3.14
```

Padding and Alignment

```
print(f"{'Python':<10}")  # Left-align
print(f"{'Python':>10}")  # Right-align
print(f"{'Python':^10}")  # Center-align
```

:<10 → The < symbol means left-align the text within a total width of 10 characters.

F-strings are the most efficient and recommended way to format strings in modern Python!

Loops in Python

Python has two main loops: `for` and `while`.

1. For Loop

Used to iterate over sequences like lists, tuples, and strings.

```
fruits = ["apple", "banana", "cherry"]

for fruit in fruits:
    print(fruit)
```

Using `range()`

```
for i in range(3):
    print(i)  # Output: 0, 1, 2
```

2. While Loop

Runs as long as a condition is `True`.

```
count = 0
while count < 3:
    print(count)
    count += 1
```

Output:

```
0
1
2
```

3. Loop Control Statements

- `break` → Exits the loop.
- `continue` → Skips to the next iteration.
- `pass` → Does nothing (used as a placeholder).

```
for i in range(5):
    if i == 3:
        break  # Stops the loop at 3
    print(i)
```

List and List Methods

A **list** in Python is an ordered, mutable collection of elements. It can contain elements of different types.

Creating a List:

```
# Empty list
my_list = []

# List with elements
numbers = [1, 2, 3, 4, 5]
```

```python
# Mixed data types
mixed_list = [1, "Hello", 3.14, True]
```

Common List Methods

Method	Description	Example
`append(x)`	Adds an element `x` to the end of the list.	`my_list.append(10)`
`extend(iterable)`	Extends the list by appending all elements from an iterable.	`my_list.extend([6, 7, 8])`
`insert(index, x)`	Inserts `x` at the specified `index`.	`my_list.insert(2, "Python")`
`remove(x)`	Removes the first occurrence of `x` in the list.	`my_list.remove(3)`
`pop([index])`	Removes and returns the element at `index` (last element if index is not provided).	`my_list.pop(2)`
`index(x)`	Returns the index of the first occurrence of `x`.	`my_list.index(4)`
`count(x)`	Returns the number of times `x` appears in the list.	`my_list.count(2)`
`sort()`	Sorts the list in ascending order.	`my_list.sort()`
`reverse()`	Reverses the order of the list.	`my_list.reverse()`
`copy()`	Returns a shallow copy of the list.	`new_list = my_list.copy()`
`clear()`	Removes all elements from the list.	`my_list.clear()`

Example Usage:

```
fruits = ["apple", "banana", "cherry"]
fruits.append("orange")
print(fruits)  # ['apple', 'banana', 'cherry', 'orange']

fruits.sort()
print(fruits)  # ['apple', 'banana', 'cherry', 'orange']
```

Tuples and Tuple Methods

A **tuple** in Python is an ordered, immutable collection of elements. It is similar to a list, but once created, its elements cannot be modified.

Creating a Tuple:

```
# Empty tuple
empty_tuple = ()

# Tuple with elements
numbers = (1, 2, 3, 4, 5)

# Mixed data types
mixed_tuple = (1, "Hello", 3.14, True)

# Single element tuple (comma is necessary)
single_element = (42,)
```

Common Tuple Methods

Method	Description	Example
`count(x)`	Returns the number of times `x` appears in the tuple.	`my_tuple.count(2)`
`index(x)`		`my_tuple.index(3)`

Method	Description	Example
	Returns the index of the first occurrence of `x`.	

Tuple Characteristics

- **Immutable**: Once created, elements cannot be changed.
- **Faster than lists**: Accessing elements in a tuple is faster than in a list.
- **Can be used as dictionary keys**: Since tuples are immutable, they can be used as keys in dictionaries.

Accessing Tuple Elements

```python
my_tuple = (10, 20, 30, 40)

# Indexing
print(my_tuple[1])  # 20

# Slicing
print(my_tuple[1:3])  # (20, 30)
```

Tuple Packing and Unpacking

```python
# Packing
person = ("Alice", 25, "Engineer")

# Unpacking
name, age, profession = person
print(name)   # Alice
print(age)    # 25
```

When to Use Tuples?

- When you want an **unchangeable** collection of elements.
- When you need a **faster** alternative to lists.
- When storing **heterogeneous data** (e.g., database records, coordinates).

Set and Set Methods

A **set** in Python is an **unordered**, **mutable**, and **unique** collection of elements. It does not allow duplicate values.

Creating a Set:

```
# Empty set (must use set(), not {})
empty_set = set()

# Set with elements
numbers = {1, 2, 3, 4, 5}

# Mixed data types
mixed_set = {1, "Hello", 3.14, True}

# Creating a set from a list
unique_numbers = set([1, 2, 2, 3, 4, 4, 5])
print(unique_numbers)  # {1, 2, 3, 4, 5}
```

Common Set Methods

Method	Description	Example
`add(x)`	Adds an element `x` to the set.	`my_set.add(10)`
`update(iterable)`	Adds multiple elements from an iterable.	`my_set.update([6, 7, 8])`
`remove(x)`	Removes `x` from the set (raises an error if not found).	`my_set.remove(3)`

Method	Description	Example
`discard(x)`	Removes `x` from the set (does not raise an error if not found).	`my_set.discard(3)`
`pop()`	Removes and returns a random element.	`my_set.pop()`
`clear()`	Removes all elements from the set.	`my_set.clear()`
`copy()`	Returns a shallow copy of the set.	`new_set = my_set.copy()`

Set Operations

Operation	Description	Example
`union(set2)`	Returns a new set with all unique elements from both sets.	`set1.union(set2)`
`intersection(set2)`	Returns a set with elements common to both sets.	`set1.intersection(set2)`
`difference(set2)`	Returns a set with elements in `set1` but not in `set2` .	`set1.difference(set2)`
`symmetric_difference(set2)`		`set1.symmetric_difference(set2)`

Operation	Description	Example
	Returns a set with elements in either `set1` or `set2`, but not both.	
`issubset(set2)`	Returns `True` if `set1` is a subset of `set2`.	`set1.issubset(set2)`
`issuperset(set2)`	Returns `True` if `set1` is a superset of `set2`.	`set1.issuperset(set2)`

Example Usage:

In Python, sets support intuitive operators for common operations like union (`|`), intersection (`&`), difference (`-`), and symmetric difference (`^`). These have equivalent method forms too, like `.union()`, `.intersection()`, etc. Here's a quick example:

```
set1 = {1, 2, 3, 4}
set2 = {3, 4, 5, 6}

# Union - combines all unique elements
print(set1 | set2)            # {1, 2, 3, 4, 5, 6}
print(set1.union(set2))       # same result

# Intersection - common elements
print(set1 & set2)            # {3, 4}
```

```python
print(set1.intersection(set2))# same result

# Difference - in set1 but not in set2
print(set1 - set2)            # {1, 2}
print(set1.difference(set2))  # same result

# Symmetric Difference - in either set, but not both
print(set1 ^ set2)                     # {1, 2, 5, 6}
print(set1.symmetric_difference(set2))# same result
```

Key Properties of Sets:

- **Unordered**: No indexing or slicing.
- **Unique Elements**: Duplicates are automatically removed.
- **Mutable**: You can add or remove elements.

Dictionary and Dictionary Methods

A **dictionary** in Python is an **unordered**, **mutable**, and **key-value** pair collection. It allows efficient data retrieval and modification. Dictionaries in Python are ordered as of Python 3.7

Creating a Dictionary:

```python
# Empty dictionary
empty_dict = {}

# Dictionary with key-value pairs
student = {
    "name": "Alice",
    "age": 25,
    "grade": "A"
}

# Using dict() constructor
person = dict(name="John", age=30, city="New York")
```

Accessing Dictionary Elements

```python
# Using keys
print(student["name"])  # Alice

# Using get() (avoids KeyError if key doesn't exist)
print(student.get("age"))  # 25
print(student.get("height", "Not Found"))  # Default value
```

Common Dictionary Methods

Method	Description	Example
`keys()`	Returns all keys in the dictionary.	`student.keys()`
`values()`	Returns all values in the dictionary.	`student.values()`
`items()`	Returns key-value pairs as tuples.	`student.items()`
`get(key, default)`	Returns value for `key`, or `default` if key not found.	`student.get("age", 0)`
`update(dict2)`	Merges `dict2` into the dictionary.	`student.update({"age": 26})`
`pop(key, default)`	Removes key and returns its value (or `default` if key not found).	`student.pop("grade")`
`popitem()`	Removes and returns the last inserted key-value pair.	`student.popitem()`

Method	Description	Example
`setdefault(key, default)`	Returns value for `key`, else sets it to `default`.	`student.setdefault("city", "Unknown")`
`clear()`	Removes all items from the dictionary.	`student.clear()`
`copy()`	Returns a shallow copy of the dictionary.	`new_dict = student.copy()`

Example Usage:

```
student = {"name": "Alice", "age": 25, "grade": "A"}

# Adding a new key-value pair
student["city"] = "New York"

# Updating an existing value
student["age"] = 26

# Removing an item
student.pop("grade")

# Iterating over a dictionary
for key, value in student.items():
    print(key, ":", value)

# Output:
# name : Alice
# age : 26
# city : New York
```

Dictionary Comprehension:

```
# Creating a dictionary using comprehension
squares = {x: x**2 for x in range(1, 6)}
print(squares)  # {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

Key Properties of Dictionaries:

- **Unordered** (Python 3.6+ maintains insertion order).
- **Keys must be unique and immutable** (e.g., strings, numbers, tuples).
- **Values can be mutable** and of any type.

File Handling in Python

File handling allows Python programs to **read, write, and manipulate files** stored on disk. Python provides built-in functions for working with files.

Opening a File

Python uses the `open()` function to open a file.

Syntax

```
file = open("filename", mode)
```

- **`filename`** → The name of the file to open.
- **`mode`** → Specifies how the file should be opened.

File Modes

Mode	Description
`'r'`	Read (default) – Opens file for reading, **raises an error if file does not exist**.

Mode	Description
`'w'`	Write – Opens file for writing, **creates a new file if not found**, and **overwrites existing content**.
`'a'`	Append – Opens file for writing, **creates a new file if not found**, and appends content instead of overwriting.
`'x'`	Create – Creates a new file, but **fails if the file already exists**.
`'b'`	Binary mode – Used with `rb`, `wb`, `ab`, etc., for working with non-text files (e.g., images, PDFs).
`'t'`	Text mode (default) – Used for text files (e.g., `rt`, `wt`).

Reading Files

Using `read()` – Read Entire File

```
file = open("example.txt", "r")
content = file.read()
print(content)
file.close()  # Always close the file after use
```

Using `readline()` – Read Line by Line

```
file = open("example.txt", "r")
line1 = file.readline()  # Reads first line
print(line1)
file.close()
```

Using `readlines()` – Read All Lines as List

```
file = open("example.txt", "r")
lines = file.readlines()  # Reads all lines into a list
print(lines)
file.close()
```

Writing to Files

Using `write()` – Overwrites Existing Content

```
file = open("example.txt", "w")  # Opens file in write mode
file.write("Hello, World!")  # Writes content
file.close()
```

Using `writelines()` – Write Multiple Lines

```
lines = ["Hello\n", "Welcome to Python\n", "File Handling\n"]

file = open("example.txt", "w")
file.writelines(lines)  # Writes multiple lines
file.close()
```

Appending to a File

The `a` **(append) mode** is used to add content to an existing file without erasing previous data.

```
file = open("example.txt", "a")
file.write("\nThis is an additional line.")
file.close()
```

Using `with` Statement (Best Practice)

Using `with open()` ensures the file is **automatically closed** after execution.

```
with open("example.txt", "r") as file:
    content = file.read()
    print(content)  # No need to manually close the file
```

Checking if a File Exists

Use the `os` module to check if a file exists before opening it.

```
import os

if os.path.exists("example.txt"):
    print("File exists!")
else:
    print("File not found!")
```

Deleting a File

Use the `os` module to delete a file.

```
import os

if os.path.exists("example.txt"):
    os.remove("example.txt")
    print("File deleted.")
else:
    print("File does not exist.")
```

Working with Binary Files

Binary files (`.jpg` , `.png` , `.pdf` , etc.) should be opened in **binary mode** (`'b'`).

Reading a Binary File

```
with open("image.jpg", "rb") as file:
    data = file.read()
    print(data)  # Outputs binary content
```

Writing to a Binary File

```
with open("new_image.jpg", "wb") as file:
    file.write(data)  # Writes binary content to a new file
```

Summary of File Operations

Operation	Description	Example
Open file	Open a file	`file = open("example.txt", "r")`
Read file	Read all content	`file.read()`
Read line	Read one line	`file.readline()`
Read lines	Read all lines into list	`file.readlines()`
Write file	Write content (overwrite)	`file.write("Hello")`
Append file	Add content to the end	`file.write("\nMore text")`
		`os.path.exists("file.txt")`

Operation	Description	Example
Check file existence	Check before opening/deleting	
Delete file	Remove a file	`os.remove("file.txt")`

JSON module in Python

JSON (**JavaScript Object Notation**) is a lightweight data format used for data exchange between servers and applications. It is widely used in **APIs, web applications, and configurations**.

Python provides the `json` module to work with JSON data. You can import the json module like this:

```
import json
```

Converting Python Objects to JSON (Serialization)

Serialization (also called **encoding or dumping**) is converting a Python object into a JSON-formatted string.

`json.dumps()` – Convert Python object to JSON string

```
import json

data = {"name": "Alice", "age": 25, "city": "New York"}

json_string = json.dumps(data)
print(json_string)  # Output: {"name": "Alice", "age": 25, "city"
print(type(json_string))  # <class 'str'>
```

`json.dump()` – Write JSON data to a file

```
with open("data.json", "w") as file:
    json.dump(data, file)
```

Converting JSON to Python Objects (Deserialization)

Deserialization (also called **decoding or loading**) is converting JSON-formatted data into Python objects.

`json.loads()` – Convert JSON string to Python object

```
json_data = '{"name": "Alice", "age": 25, "city": "New York"}'

python_obj = json.loads(json_data)
print(python_obj)  # Output: {'name': 'Alice', 'age': 25, 'city':
print(type(python_obj))  # <class 'dict'>
```

`json.load()` – Read JSON data from a file

```
with open("data.json", "r") as file:
    python_data = json.load(file)

print(python_data)  # Output: {'name': 'Alice', 'age': 25, 'city'
```

Formatting JSON Output

You can format JSON for better readability using **indentation**.

```
formatted_json = json.dumps(data, indent=4)
print(formatted_json)
```

Output:

```
{
    "name": "Alice",
    "age": 25,
    "city": "New York"
}
```

Summary of Common JSON Methods

Method	Description	Example
`json.dumps(obj)`	Converts Python object to JSON string	`json.dumps(data)`
`json.dump(obj, file)`	Writes JSON to a file	`json.dump(data, file)`
`json.loads(json_string)`	Converts JSON string to Python object	`json.loads(json_data)`
`json.load(file)`	Reads JSON from a file	`json.load(file)`

Object Oriented Programming in Python

Object-Oriented Programming (OOP) is a **programming paradigm** that organizes code into objects that contain both **data (attributes)** and **behavior (methods)**.

Key Concepts of OOP

Concept	Description
Class	A blueprint for creating objects.

Concept	Description
Object	An instance of a class with specific data and behavior.
Attributes	Variables that store data for an object.
Methods	Functions inside a class that define object behavior.
Encapsulation	Restricting direct access to an object's data.
Inheritance	Creating a new class from an existing class.
Polymorphism	Using the same method name for different classes.

---

1. Defining a Class and Creating an Object

Creating a Class

```python
class Car:
    def __init__(self, brand, model):
        self.brand = brand  # Attribute
        self.model = model  # Attribute

    def display_info(self):  # Method
        return f"{self.brand} {self.model}"

# Creating an Object (Instance)
car1 = Car("Toyota", "Camry")
print(car1.display_info())  # Output: Toyota Camry
```

---

2. Encapsulation (Data Hiding)

Encapsulation prevents direct modification of attributes and allows controlled access using **getter and setter methods**.

```python
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance  # Private Attribute

    def get_balance(self):  # Getter
        return self.__balance

    def deposit(self, amount):  # Setter
        if amount > 0:
            self.__balance += amount

# Using Encapsulation
account = BankAccount(1000)
account.deposit(500)
print(account.get_balance())  # Output: 1500
```

◇ **Why use encapsulation?**

It protects data by restricting direct modification.

---

3. Inheritance (Reusing Code)

Inheritance allows a class (**child**) to inherit attributes and methods from another class (**parent**).

Example of Single Inheritance

```python
class Animal:
    def speak(self):
        return "Animal makes a sound"

class Dog(Animal):  # Inheriting from Animal
    def speak(self):
        return "Bark"

dog = Dog()
print(dog.speak())  # Output: Bark
```

◇ **Why use inheritance?**
It promotes **code reusability** and maintains a cleaner code structure.

---

4. Multiple Inheritance

A class can inherit from multiple parent classes.

```
class A:
    def method_a(self):
        return "Method A"

class B:
    def method_b(self):
        return "Method B"

class C(A, B):  # Multiple Inheritance
    pass

obj = C()
print(obj.method_a())  # Output: Method A
print(obj.method_b())  # Output: Method B
```

◇ **Why use multiple inheritance?**
It allows a class to inherit **features from multiple parent classes**.

---

5. Polymorphism (Same Method, Different Behavior)

Polymorphism allows different classes to use the **same method name**.

Method Overriding Example

```
class Bird:
    def fly(self):
        return "Birds can fly"
```

```python
class Penguin(Bird):
    def fly(self):
        return "Penguins cannot fly"

bird = Bird()
penguin = Penguin()

print(bird.fly())       # Output: Birds can fly
print(penguin.fly())    # Output: Penguins cannot fly
```

◇ **Why use polymorphism?**
It provides **flexibility** by allowing different classes to define the same method differently.

---

6. Abstraction (Hiding Implementation Details)

Abstraction is used to define a method **without implementing it** in the base class. It is achieved using **abstract base classes** (`ABC` module).

```python
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass  # No implementation

class Square(Shape):
    def __init__(self, side):
        self.side = side

    def area(self):
        return self.side * self.side  # Implemented in child clas

square = Square(4)
print(square.area())  # Output: 16
```

◇ **Why use abstraction?**

It enforces **consistent implementation** across child classes.

---

7. Magic Methods (Dunder Methods)

Magic methods allow objects to behave like **built-in types**.

Example: `__str__()` and `__len__()`

Have a look at the code below:

```python
class Book:
    def __init__(self, title, pages):
        self.title = title
        self.pages = pages

    def __str__(self):  # String representation
        return f"Book: {self.title}"

    def __len__(self):  # Define behavior for len()
        return self.pages

book = Book("Python Basics", 300)
print(str(book))  # Output: Book: Python Basics
print(len(book))  # Output: 300
```

---

8. Class vs. Static Methods

Method Type	Description	Uses `self` ?	Uses `cls` ?
Instance Method	Works with instance attributes	☑	✕

Method Type	Description	Uses `self`?	Uses `cls`?
Class Method	Works with class attributes	✗	☑
Static Method	Does not use class or instance variables	✗	✗

Example

```python
class Example:
    class_var = "I am a class variable"

    def instance_method(self):
        return "Instance Method"

    @classmethod
    def class_method(cls):
        return cls.class_var

    @staticmethod
    def static_method():
        return "Static Method"

obj = Example()
print(obj.instance_method())  # Output: Instance Method
print(Example.class_method()) # Output: I am a class variable
print(Example.static_method()) # Output: Static Method
```

Summary of OOP Concepts

Concept	Description	Example
Class	A blueprint for creating objects	`class Car:`

Concept	Description	Example
Object	An instance of a class	`car1 = Car()`
Encapsulation	Restrict direct access to data	`self.__balance`
Inheritance	A class inherits from another class	`class Dog(Animal)`
Polymorphism	Using the same method in different ways	`def fly(self)`
Abstraction	Hiding implementation details	`@abstractmethod`
Magic Methods	Special methods like `__str__()`	`def __len__(self)`
Class Methods	Works with class variables	`@classmethod`
Static Methods	Independent of class and instance	`@staticmethod`

List Comprehension

List comprehension is a **concise** and **efficient** way to create lists in Python. It allows you to generate lists in a **single line of code**, making your code more readable and Pythonic.

---

1. Basic Syntax

```
[expression for item in iterable]
```

- `expression` → The operation to perform on each item
- `item` → The variable representing each element in the iterable
- `iterable` → The data structure being iterated over (list, range, etc.)

Example: Creating a list of squares

```
squares = [x**2 for x in range(5)]
print(squares)  # Output: [0, 1, 4, 9, 16]
```

---

2. Using if Condition in List Comprehension

Example: Filtering even numbers

```
evens = [x for x in range(10) if x % 2 == 0]
print(evens)  # Output: [0, 2, 4, 6, 8]
```

---

3. Using if-else Condition in List Comprehension

Example: Replacing even numbers with “Even” and odd numbers with “Odd”

```
numbers = ["Even" if x % 2 == 0 else "Odd" for x in range(5)]
print(numbers)  # Output: ['Even', 'Odd', 'Even', 'Odd', 'Even']
```

---

4. Nested Loops in List Comprehension

Example: Creating pairs from two lists

```
pairs = [(x, y) for x in range(2) for y in range(3)]
print(pairs)  # Output: [(0, 0), (0, 1), (0, 2), (1, 0), (1, 1),
```

---

5. List Comprehension with Functions

Example: Converting a list of strings to uppercase

```
words = ["hello", "world", "python"]
upper_words = [word.upper() for word in words]
print(upper_words)  # Output: ['HELLO', 'WORLD', 'PYTHON']
```

---

6. List Comprehension with Nested List Comprehension

Example: Flattening a 2D list

```
matrix = [[1, 2], [3, 4], [5, 6]]
flattened = [num for row in matrix for num in row]
print(flattened)  # Output: [1, 2, 3, 4, 5, 6]
```

---

7. List Comprehension with Set and Dictionary Comprehensions

Set Comprehension

```
unique_numbers = {x for x in [1, 2, 2, 3, 4, 4]}
print(unique_numbers)  # Output: {1, 2, 3, 4}
```

Dictionary Comprehension

```
squared_dict = {x: x**2 for x in range(5)}
print(squared_dict)  # Output: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

8. When to Use List Comprehensions?

- You need to create a list in a **single line**
- The logic is **simple and readable**
- You want to improve **performance** (faster than loops)

Avoid when: - The logic is **too complex** (use a standard loop instead for clarity)

9. Performance Comparison: List Comprehension vs. Loop

```python
import time

# Using a for loop
start = time.time()
squares_loop = []
for x in range(10**6):
    squares_loop.append(x**2)
print("Loop time:", time.time() - start)

# Using list comprehension
start = time.time()
squares_comp = [x**2 for x in range(10**6)]
print("List Comprehension time:", time.time() - start)
```

List comprehensions are generally faster than loops because they are optimized internally by Python.

Summary

Concept	Example
Basic List Comprehension	`[x**2 for x in range(5)]`
With Condition (`if`)	`[x for x in range(10) if x % 2 == 0]`
With `if-else`	`["Even" if x % 2 == 0 else "Odd" for x in range(5)]`
Nested Loop	`[(x, y) for x in range(2) for y in range(3)]`
Flatten 2D List	`[num for row in matrix for num in row]`
Set Comprehension	`{x for x in [1, 2, 2, 3]}`
Dictionary Comprehension	`{x: x**2 for x in range(5)}`

Lambda Functions

A **lambda function** in Python is an **anonymous, single-expression function** defined using the `lambda` keyword. It is commonly used for **short, throwaway functions** where a full function definition is unnecessary.

---

1. Syntax of Lambda Functions

```
lambda arguments: expression
```

- `lambda` → Keyword to define a lambda function
- `arguments` → Input parameters (comma-separated)
- `expression` → The operation performed (must be a **single** expression, not multiple statements)

Example: Simple Lambda Function

```python
square = lambda x: x ** 2
print(square(5))  # Output: 25
```

---

2. Using Lambda Functions with `map()`, `filter()`, and `reduce()`

2.1 Using `map()` with Lambda

Applies a function to each element of an iterable.

```python
numbers = [1, 2, 3, 4]
squared = list(map(lambda x: x**2, numbers))
print(squared)  # Output: [1, 4, 9, 16]
```

2.2 Using `filter()` with Lambda

Filters elements based on a condition.

```python
numbers = [1, 2, 3, 4, 5, 6]
evens = list(filter(lambda x: x % 2 == 0, numbers))
print(evens)  # Output: [2, 4, 6]
```

2.3 Using `reduce()` with Lambda

Reduces an iterable to a single value (requires `functools.reduce`).

```python
from functools import reduce
```

```
numbers = [1, 2, 3, 4]
product = reduce(lambda x, y: x * y, numbers)
print(product)  # Output: 24
```

3. Lambda with Multiple Arguments

Example: Adding Two Numbers

```
add = lambda x, y: x + y
print(add(3, 7))  # Output: 10
```

Example: Finding the Maximum of Two Numbers

```
maximum = lambda x, y: x if x > y else y
print(maximum(10, 5))  # Output: 10
```

4. Lambda in Sorting Functions

Sorting a List of Tuples

```
students = [("Alice", 85), ("Bob", 78), ("Charlie", 92)]
students.sort(key=lambda student: student[1])  # Sort by score
print(students)  # Output: [('Bob', 78), ('Alice', 85), ('Charlie
```

5. When to Use Lambda Functions?

Use Lambda Functions When:

- The function is **short and simple**.

- Used **temporarily** inside another function (e.g., `map` , `filter`).
- Avoiding defining a full function with `def` .

Avoid Lambda Functions When:

- The function is **complex** (use `def` for readability).
- Multiple operations/statements are needed.

Summary

Feature	Example
Basic Lambda Function	`lambda x: x**2`
With `map()`	`map(lambda x: x**2, numbers)`
With `filter()`	`filter(lambda x: x % 2 == 0, numbers)`
With `reduce()`	`reduce(lambda x, y: x * y, numbers)`
Multiple Arguments	`lambda x, y: x + y`
Sorting with Lambda	`sort(key=lambda x: x[1])`

Coders of Delhi

Welcome - Your Data Science Internship begins

Congratulations! You have just been hired as a **Data Scientist Intern** at **CodeBook – The Social Media for Coders**. This Delhi-based company is offering you a **₹10 LPA job** if you successfully complete this **1-month internship**. But before you get there, you must prove your skills using **only Python**—no pandas, NumPy, or fancy libraries!

Your manager Puneet Kumar has assigned you your **first task**: analyzing a data dump of CodeBook users using pure Python. Your job is to **load and explore the data** to understand its structure.

Task 1: Load the User Data

Your manager has given you a dataset containing information about CodeBook users, their connections (friends), and the pages they have liked.

This is how the data will look (in JSON format):

```json
{
    "users": [
        {"id": 1, "name": "Amit", "friends": [2, 3], "liked_pages": [101]},
        {"id": 2, "name": "Priya", "friends": [1, 4], "liked_pages": [102]},
        {"id": 3, "name": "Rahul", "friends": [1], "liked_pages": [101, 103]},
        {"id": 4, "name": "Sara", "friends": [2], "liked_pages": [104]}
    ],
    "pages": [
        {"id": 101, "name": "Python Developers"},
        {"id": 102, "name": "Data Science Enthusiasts"},
        {"id": 103, "name": "AI & ML Community"},
        {"id": 104, "name": "Web Dev Hub"}
    ]
}
```

We have to read this data and understand its structure. The data contains three main components: 1. **Users**: Each user has an ID, name, a list of friends (by their IDs), and a list of liked pages (by their IDs). 2. **Pages**: Each page has an ID and a name. 3. **Connections**: Users can have multiple friends and can like multiple pages.

Task 2: Read and Display the Data using Python

Your goal is to **load** this data and **print** it in a structured way. Use Python's built-in modules to accomplish this.

Steps:

1. Save the JSON data in a file (`codebook_data.json`).
2. Read the JSON file using Python.
3. Print user details and their connections.
4. Print available pages.

Code Implementation

Here's a simple way to load and display the data:

```python
import json

# Load the JSON file
def load_data(filename):
    with open(filename, "r") as file:
        data = json.load(file)
    return data

# Display users and their connections
def display_users(data):
    print("Users and Their Connections:\n")
    for user in data["users"]:
        print(f"{user['name']} (ID: {user['id']}) - Friends: {user['friends']} - Liked Pages: {user['liked_pages']}")
    print("\nPages:\n")
    for page in data["pages"]:
        print(f"{page['id']}: {page['name']}")

# Load and display the data
data = load_data("codebook_data.json")
display_users(data)
```

Expected Output:

```
Users and Their Connections:
Amit (ID: 1) - Friends: [2, 3] - Liked Pages: [101]
Priya (ID: 2) - Friends: [1, 4] - Liked Pages: [102]
Rahul (ID: 3) - Friends: [1] - Liked Pages: [101, 103]
Sara (ID: 4) - Friends: [2] - Liked Pages: [104]

Pages:
101: Python Developers
102: Data Science Enthusiasts
103: AI & ML Community
104: Web Dev Hub
```

Next Steps

Your manager is happy with your progress but says: “**The data looks messy. Can you clean and structure it better?**”

Cleaning and Structuring the Data

Introduction

Your manager is impressed with your progress but points out that the data is messy. Before we can analyze it effectively, we need to **clean and structure the data** properly.

Your task is to:

- Handle missing values
- Remove duplicate or inconsistent data
- Standardize the data format

Let's get started!

Task 1: Identify Issues in the Data

Your manager provides you with an example dataset where some records are incomplete or incorrect. Here's an example:

```
{
    "users": [
        {"id": 1, "name": "Amit", "friends": [2, 3], "liked_pages": [101]},
        {"id": 2, "name": "Priya", "friends": [1, 4], "liked_pages": [102]},
        {"id": 3, "name": "", "friends": [1], "liked_pages": [101, 103]},
        {"id": 4, "name": "Sara", "friends": [2, 2], "liked_pages": [104]},
        {"id": 5, "name": "Amit", "friends": [], "liked_pages": []}
    ],
    "pages": [
        {"id": 101, "name": "Python Developers"},
        {"id": 102, "name": "Data Science Enthusiasts"},
        {"id": 103, "name": "AI & ML Community"},
        {"id": 104, "name": "Web Dev Hub"},
        {"id": 104, "name": "Web Development"}
    ]
}
```

Problems:

1. User **ID 3** has an empty name.
2. User **ID 4** has a duplicate friend entry.
3. User **ID 5** has no connections or liked pages (inactive user).
4. The **pages list** contains duplicate page IDs.

Task 2: Clean the Data

We will:

1. Remove users with missing names.
2. Remove duplicate friend entries.
3. Remove inactive users (users with no friends and no liked pages).
4. Deduplicate pages based on IDs.

Code Implementation

```
import json

def clean_data(data):
    # Remove users with missing names
    data["users"] = [user for user in data["users"] if user["name"].strip()]

    # Remove duplicate friends
    for user in data["users"]:
        user["friends"] = list(set(user["friends"]))

    # Remove inactive users
    data["users"] = [user for user in data["users"] if user["friends"] or user["lik
ed_pages"]]

    # Remove duplicate pages
    unique_pages = {}
    for page in data["pages"]:
        unique_pages[page["id"]] = page
    data["pages"] = list(unique_pages.values())

    return data
```

```
# Load, clean, and display the cleaned data
data = json.load(open("codebook_data.json"))
data = clean_data(data)
json.dump(data, open("cleaned_codebook_data.json", "w"), indent=4)
print("Data cleaned successfully!")
```

Expected Output:

```
{
    "users": [
        {"id": 1, "name": "Amit", "friends": [2, 3], "liked_pages": [101]},
        {"id": 2, "name": "Priya", "friends": [1, 4], "liked_pages": [102]},
        {"id": 4, "name": "Sara", "friends": [2], "liked_pages": [104]}
    ],
    "pages": [
        {"id": 101, "name": "Python Developers"},
        {"id": 102, "name": "Data Science Enthusiasts"},
        {"id": 103, "name": "AI & ML Community"},
        {"id": 104, "name": "Web Development"}
    ]
}
```

The cleaned dataset will:

- Remove users with missing names
- Ensure friend lists contain unique entries
- Remove inactive users
- Deduplicate pages

Next Steps

Your manager is happy with the cleaned data and says: "**Great! Now that our data is structured, let's start analyzing it**. You are an intern, but he is so confident in your skills that he asks you - Can you build a 'People You May Know' feature?" Let's do that next!

Finding “People You May Know”

Now that our data is cleaned and structured, your manager assigns you a new task: **Build a ‘People You May Know’ feature!**

In social networks, this feature helps users connect with others by suggesting friends based on mutual connections. Your job is to **analyze mutual friends and recommend potential connections**.

Task 1: Understand the Logic

How ‘People You May Know’ Works:

- If **User A** and **User B** are not friends but have **mutual friends**, we suggest User B to User A and vice versa.
- More mutual friends = higher priority recommendation.

Example:

- **Amit (ID: 1)** is friends with **Priya (ID: 2)** and **Rahul (ID: 3)**.
- **Priya (ID: 2)** is friends with **Sara (ID: 4)**.
- Amit is not directly friends with Sara, but they share **Priya as a mutual friend**.
- Suggest **Sara to Amit** as “People You May Know”.

But there are cases where we will have more than one “People You May Know”. In those cases, greater the number of mutual friends, higher the probability that the user might know the person we are recommending.

Task 2: Implement the Algorithm

We’ll create a function that:

1. Finds all friends of a given user.
2. Identifies mutual friends between non-friends.
3. Ranks recommendations by the number of mutual friends.

Code Implementation

```
import json

def load_data(filename):
    with open(filename, "r") as file:
        return json.load(file)

def find_people_you_may_know(user_id, data):
    user_friends = {}
    for user in data["users"]:
        user_friends[user["id"]] = set(user["friends"])

    if user_id not in user_friends:
        return []

    direct_friends = user_friends[user_id]
    suggestions = {}

    for friend in direct_friends:
        # For all friends of friend
        for mutual in user_friends[friend]:
            # If mutual id is not the same user and not already a direct friend of
user
            if mutual != user_id and mutual not in direct_friends:
                # Count mutual friends
                suggestions[mutual] = suggestions.get(mutual, 0) + 1

    sorted_suggestions = sorted(suggestions.items(), key=lambda x: x[1], reverse=Tr
ue)
    return [user_id for user_id, _ in sorted_suggestions]

# Load data
data = load_data("cleaned_codebook_data.json")
user_id = 1  # Example: Finding suggestions for Amit
recommendations = find_people_you_may_know(user_id, data)
print(f"People You May Know for User {user_id}: {recommendations}")
```

Expected Output:

If Amit (ID: 1) and Sara (ID: 4) share Priya (ID: 2) as a mutual friend, the output might be:

```
People You May Know for User 1: [4]
```

This suggests that **Amit should connect with Sara!**

The Fontend developer of CodeBook can use this data via API and show it in the frontend when Amit logs in

---

Next Steps

Your manager is excited about your progress and now says: “Great job! Next, find ‘Pages You Might Like’ based on your connections and preferences.”

Let’s make sure we live up to his expectations.

Finding “Pages You Might Like”

We’ve officially reached the final milestone of our first data science project at **CodeBook – The Social Media for Coders**. After cleaning messy data and building features like **People You May Know**, it’s time to launch our last feature: **Pages You Might Like**.

Why This Matters

In real-world social networks, content discovery keeps users engaged. This feature simulates that experience using nothing but **pure Python**, showing how even simple logic can power impactful insights. So now the situation is that your manager is impressed with your ‘People You May Know’ feature and now assigns you a new challenge: **Recommend pages that users might like!**

On social media platforms, users interact with pages by liking, following, or engaging with posts. The goal is to analyze these interactions and suggest relevant pages based on user behavior.

Task 1: Understanding the Recommendation Logic

How ‘Pages You Might Like’ Works:

- Users engage with pages (like, comment, share, etc.).
- If two users have interacted with similar pages, they are likely to have common interests.
- For the sake of this implementation, we consider liking a page as an interaction
- Pages followed by similar users should be recommended.

Example:

- **Amit (ID: 1)** likes **Python Hub (Page ID: 101)** and **AI World (Page ID: 102)**.
- **Priya (ID: 2)** likes **AI World (Page ID: 102)** and **Data Science Daily (Page ID: 103)**.
- Since Amit and Priya both like **AI World (102)**, we suggest **Data Science Daily (103)** to Amit and **Python Hub (101)** to Priya.

What we are using here is called **collaborative filtering**:

> *“If two people like the same thing, maybe they’ll like other things each one likes too.”*

This is a basic form of a real-world recommendation engine, and our task is to implement it in pure Python. Let’s Go!

Task 2: Implement the Algorithm

We’ll create a function that:

1. Maps users to pages they have interacted with.
2. Identifies pages liked by users with similar interests.
3. Ranks recommendations based on common interactions.

Code Implementation

```python
import json

# Function to load JSON data from a file
def load_data(filename):
    with open(filename, "r") as file:
        return json.load(file)

# Function to find pages a user might like based on common interests
def find_pages_you_might_like(user_id, data):
    # Dictionary to store user interactions with pages
    user_pages = {}
    for user in data["users"]:
        user_pages[user["id"]] = set(user["liked_pages"])

    # If the user is not found, return an empty list
    if user_id not in user_pages:
        return []

    user_liked_pages = user_pages[user_id]
    page_suggestions = {}

    for other_user, pages in user_pages.items():
        if other_user != user_id:
            shared_pages = user_liked_pages.intersection(pages)
            for page in pages:
                if page not in user_liked_pages:
                    page_suggestions[page] = page_suggestions.get(page, 0) + len(sh
ared_pages)

    # Sort recommended pages based on the number of shared interactions
    sorted_pages = sorted(page_suggestions.items(), key=lambda x: x[1], reverse=Tru
e)
    return [page_id for page_id, _ in sorted_pages]

# Load data
data = load_data("cleaned_codebook_data.json")
user_id = 1  # Example: Finding recommendations for Amit
```

```
page_recommendations = find_pages_you_might_like(user_id, data)
print(f"Pages You Might Like for User {user_id}: {page_recommendations}")
```

Expected Output:

If Amit (ID: 1) and Priya (ID: 2) both like AI World (102), and Priya also likes Data Science Daily (103), the output might be:

```
Pages You Might Like for User 1: [103]
```

This suggests that **Amit might be interested in 'Data Science Daily'!**

Now this was a very simple case, there is a chance that two users like 10 pages each, in that case all the pages liked by user 1 will be recommended to user 2 and vice versa. The score of this recommendation is the number of common pages they like

---

Conclusion

In simple terms, if two users like some of the same pages, it's likely they share similar interests. So, pages liked by one can be recommended to the other.

To make these recommendations smarter, we assign a similarity score — the number of pages they both like. The higher the score, the stronger the connection.

In larger networks where users follow many pages, this score helps us prioritize and personalize suggestions, just like how platforms like Facebook and LinkedIn recommend content or connections.

And that's a wrap!

From loading and cleaning data to recommending people and pages, you've successfully completed your first end-to-end data project using raw Python.

Your manager is impressed. Your ₹10 LPA dream offer letter is waiting for you!

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.

```python
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

```python
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

```python
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

```python
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

```python
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

```python
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

```python
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:

```python
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

```python
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:

```python
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

Introduction to Data Visualization

Why Data Visualization is Important

Data visualization is the bridge between raw data and human understanding. When done right, it helps:

- Reveal patterns, trends, and correlations in the data.
- Communicate insights clearly to stakeholders.
- Speed up decision-making by simplifying complex datasets.
- Make data storytelling engaging and accessible to all.

> A picture is worth a thousand words

> “The greatest value of a picture is when it forces us to notice what we never expected to see.” – John Tukey

---

Exploratory vs Explanatory Visuals

Exploratory Visualizations

- Purpose: Explore the data, uncover insights, find patterns.
- Audience: You (the data analyst/scientist).
- Example: Pair plots, correlation heatmaps, scatter matrix.

Explanatory Visualizations

- Purpose: Communicate a specific insight or story.
- Audience: Stakeholders, clients, public.
- Example: A bar chart in a presentation showing sales trends.

Aspect	Exploratory	Explanatory
Goal	Find insights	Communicate insights
Audience	Analyst / Data Scientist	Stakeholders / Public
Style	Raw, fast, flexible	Polished, focused, clean

Basic Principles of Good Visualizations

1. **Clarity**
   Avoid clutter. Use labels, legends, and proper axis scales.
2. **Context**
   Always provide context: What is being measured? Over what time frame? In what units?
3. **Focus**
   Highlight the key insight. Use colors and annotations to draw attention.
4. **Storytelling**
   Don’t just show data — tell a story. Guide the viewer through a narrative.
5. **Accessibility**
   Use carefully chosen color palettes that enhance readability for all viewers.

Pro Tip:

> Always ask yourself: “What is the one thing I want the viewer to understand from this visual?”

Introduction to Matplotlib

Today, we'll cover the **basics of Matplotlib** — the most fundamental plotting library in Python. By the end, you'll understand how to make clean and powerful plots, step by step.

What is `matplotlib.pyplot`?

- `matplotlib.pyplot` is a module in Matplotlib — it's like a paintbrush for your data.
- We usually import it as `plt`:

```
import matplotlib.pyplot as plt
```

> `plt` is just a short alias to save typing!

What is `plt.show()`?

- `plt.show()` is used to **display the plot**.
- Without it, in scripts, you might not see the plot window.

```
plt.plot([1, 2, 3], [4, 5, 6])
plt.show()
```

Interacting with the Plot

When a plot appears, you can:

- Zoom In/Out
- Pan around
- Use arrows to navigate history
- Reset to home
- Save as PNG using the disk icon

> These features are **automatically included** in the plot window!

Real Data Example: Sachin Tendulkar's Runs Over Time

```
years = [1990, 1992, 1994, 1996, 1998, 2000, 2003, 2005, 2007, 2010]
runs =  [500, 700, 1100, 1500, 1800, 1200, 1700, 1300, 900, 1500]
```

```
plt.plot(years, runs)
plt.show()
```

Adding X and Y Labels

```
plt.plot(years, runs)
plt.xlabel("Year")
plt.ylabel("Runs Scored")
plt.title("Sachin Tendulkar's Yearly Runs")
plt.show()
```

Multiple Lines in One Plot

```
kohli = [0, 0, 500, 800, 1100, 1300, 1500, 1800, 1900, 2100]
sehwag = [0, 300, 800, 1200, 1500, 1700, 1600, 1400, 1000, 0]

plt.plot(years, kohli, label="Virat Kohli")
plt.plot(years, sehwag, label="Virender Sehwag")

plt.xlabel("Year")
plt.ylabel("Runs Scored")
plt.title("Performance Comparison")
plt.legend()
plt.show()
```

Why `label` is Better than List in `legend`

Bad practice:

```
plt.plot(years, kohli)
plt.plot(years, sehwag)
plt.legend(["Kohli", "Sehwag"])  # prone to mismatch
```

Better:

```
plt.plot(years, kohli, label="Kohli")
plt.plot(years, sehwag, label="Sehwag")
plt.legend()
```

Using Format Strings

```
plt.plot(years, kohli, 'ro--', label="Kohli")  # red circles with dashed lines
plt.plot(years, sehwag, 'g^:', label="Sehwag")  # green triangles dotted
plt.legend()
```

Color and Line Style Arguments

```
plt.plot(years, kohli, color='orange', linestyle='--', label="Kohli")
plt.plot(years, sehwag, color='green', linestyle='-.', label="Sehwag")
plt.plot(years, runs, color='blue', label="Tendulkar")
plt.legend()
```

Line Width and Layout Tweaks

```
plt.plot(years, kohli, linewidth=3, label="Kohli")
plt.plot(years, sehwag, linewidth=2, label="Sehwag")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

Using Styles in Matplotlib

```
print(plt.style.available)
```

Try a few:

```
plt.style.use("ggplot")
# or
plt.style.use("seaborn-v0_8-bright")
```

XKCD Comic Style

```
with plt.xkcd():
    plt.plot(years, kohli, label="Kohli")
    plt.plot(years, sehwag, label="Sehwag")
    plt.title("Epic Battle of the Batsmen")
    plt.legend()
    plt.show()
```

Visualizing Tons of Data – What Crowded Looks Like

```
import numpy as np

for i in range(50):
    plt.plot(np.random.rand(100), linewidth=1)

plt.title("Too Much Data Can Be Confusing!")
plt.grid(True)
plt.tight_layout()
plt.show()
```

Final Tips for Beginners

- Always start with simple plots
- Add labels and legends early
- Use `plt.grid()` and `plt.tight_layout()` to improve readability
- Try different styles to find what works for your use case

Assignment:
Create a plot comparing **Kohli**, **Rohit Sharma**, and **Sehwag** across 10 years of hypothetical runs.
Use:

- Labels
- Legends
- Colors
- Line styles
- One custom style

Matplotlib Bar Charts

Bar charts are used to **compare quantities** across categories. They are easy to read and powerful for visual analysis.

We'll use **Sachin Tendulkar's yearly run data**, then learn how to create grouped bar charts and horizontal bar charts with examples.

Sachin's Yearly Runs – Vertical Bar Chart

The Data

```
import matplotlib.pyplot as plt

years = [1990, 1992, 1994, 1996, 1998, 2000, 2003, 2005, 2007, 2010]
runs =  [500, 700, 1100, 1500, 1800, 1200, 1700, 1300, 900, 1500]
```

Basic Bar Plot

```
plt.bar(years, runs)
plt.xlabel("Year")
plt.ylabel("Runs Scored")
plt.title("Sachin Tendulkar's Yearly Runs")
plt.show()
```

Setting Bar Width & Side-by-Side Bar Charts

Let's compare **Sachin**, **Sehwag**, and **Kohli** side-by-side for the same years.

```
import numpy as np

sachin = [500, 700, 1100, 1500, 1800, 1200, 1700, 1300, 900, 1500]
sehwag = [0, 200, 900, 1400, 1600, 1800, 1500, 1100, 800, 0]
kohli  = [0, 0, 500, 800, 1100, 1300, 1500, 1800, 1900, 2100]

x = np.arange(len(years))  # index positions
width = 0.25
```

The value of width ranges from 0 to 1 (default 0.8); it can go above 1, but bars will start to overlap.

Plotting Side-by-Side Bars

```
plt.bar(x - width, sachin, width=width, label="Sachin")
plt.bar(x, sehwag, width=width, label="Sehwag")
plt.bar(x + width, kohli, width=width, label="Kohli")

plt.xlabel("Year")
plt.ylabel("Runs")
plt.title("Run Comparison")
plt.xticks(x, years)  # Show actual year instead of 0,1,2,...
plt.legend()
plt.tight_layout()
plt.show()
```

Why Use `xticks()`?

By default, `plt.bar()` uses numeric x-values (0, 1, 2, ...).
We use `plt.xticks()` to set the correct category labels like years or names.

Horizontal Bar Charts with `barh()`

Let's compare **total runs scored in the first 5 years** by different players.

```
players = ["Sachin", "Sehwag", "Kohli", "Yuvraj"]
runs_5yrs = [500+700+1100+1500+1800, 0+200+900+1400+1600, 0+0+500+800+1100,
300+600+800+1100+900]
```

Plotting with `barh()`

```
plt.barh(players, runs_5yrs, color="skyblue")
plt.xlabel("Total Runs in First 5 Years")
plt.title("First 5-Year Performance of Indian Batsmen")
plt.tight_layout()
plt.show()
```

Why Switch `x` and `y`?

- In `bar()` → `x = categories`, `y = values`
- In `barh()` → `y = categories`, `x = values`

> Horizontal bars help when category names are long or when you want to emphasize comparisons from left to right.

Adding Value Labels with `plt.text()`

You can use `plt.text()` to display values **on top of bars**, making your chart easier to read. It's especially useful in presentations and reports.

```
import matplotlib.pyplot as plt

players = ["Sachin", "Sehwag", "Kohli"]
runs = [1500, 1200, 1800]

plt.bar(players, runs, color="skyblue")

# Add labels on top of bars
for i in range(len(players)):
    plt.text(i, runs[i] + 50, str(runs[i]), ha='center')
plt.ylabel("Runs")
plt.title("Runs Scored by Players")
plt.tight_layout()
plt.show()
```

Tip: Use `ha='center'` to center the text and add a small offset (`+50` here) to avoid overlap with the bar.

Summary

Feature	Use
`plt.bar()`	Vertical bars for categorical comparison
`plt.barh()`	Horizontal bars (great for long labels)
`width=`	Control thickness/spacing of bars
`np.arange()`	Helps to align multiple bars side by side
`plt.xticks()`	Replaces index numbers with real labels
`plt.tight_layout()`	Prevents labels from overlapping

Assignment:
Create a side-by-side bar chart comparing runs of Sachin, Kohli, and Sehwag over 5 selected years.
Then create a horizontal bar chart showing total runs scored in their debut 5 years.

Matplotlib Pie Charts

Pie charts are used to show **part-to-whole relationships**. They're visually appealing but best used with **fewer categories**, as too many slices can get cluttered and hard to interpret.

Setting a Style

```
import matplotlib.pyplot as plt

plt.style.use("ggplot")  # Choose any style you like
```

Basic Pie Chart Example

```
# Data
labels = ["Sachin", "Sehwag", "Kohli", "Yuvraj"]
runs = [18000, 8000, 12000, 9500]
```

```
plt.title("Career Runs of Indian Batsmen")
plt.pie(runs, labels=labels)
plt.show()
```

Custom Colors

You can use color names or hex codes.

```
colors = ['#ff9999','#66b3ff','#99ff99','#ffcc99']
plt.pie(runs, labels=labels, colors=colors)
plt.show()
```

Add Edges and Style Slices with `wedgeprops`

You can customize the look of the slices using `wedgeprops`.

```
plt.pie(
    runs,
    labels=labels,
    colors=colors,
    wedgeprops={'edgecolor': 'black', 'linewidth': 2, 'linestyle': '--'}
)
plt.show()
```

Tip: You can Google “matplotlib wedgeprops” for more customization options.

Visit this documentation page for more

Highlight a Slice Using `explode`

Use `explode` to **pull out** one or more slices.

```
explode = [0.1, 0, 0, 0]  # Only highlight Sachin's slice

plt.pie(runs, labels=labels, explode=explode, colors=colors)
plt.title("Exploded Pie Example")
plt.show()
```

Add Shadows for a 3D Feel

```
plt.pie(
    runs,
    labels=labels,
    explode=explode,
    colors=colors,
    shadow=True  # adds a 3D-like shadow
)
plt.show()
```

Start Angle

Use `startangle` to rotate the pie for better alignment.

```
plt.pie(
    runs,
    labels=labels,
    explode=explode,
    colors=colors,
    shadow=True,
    startangle=140
)
plt.show()
```

Show Percentages with `autopct`

```
plt.pie(
    runs,
    labels=labels,
    autopct='%1.1f%%',
    startangle=140
)
plt.title("Career Run Share")
plt.show()
```

Crowded Pie Chart – Why to Avoid

```
# Too many categories
languages = ['Python', 'Java', 'C++', 'JavaScript', 'C#', 'Ruby', 'Go', 'Rust',
'Swift', 'PHP']
usage = [30, 20, 10, 10, 7, 5, 4, 3, 2, 1]

plt.pie(usage, labels=languages, autopct='%1.1f%%', startangle=90)
plt.title("Programming Language Usage (Crowded Example)")
plt.show()
```

Why avoid it?

- Hard to compare slice sizes visually
- Cluttered and confusing
- No clear insight
  Better alternatives: bar charts or horizontal bar charts

Summary of Useful Pie Chart Parameters

Parameter	Use
`labels`	Label each slice
`colors`	Customize slice colors (names or hex)
`explode`	Pull out slices for emphasis
`shadow`	Adds depth-like shadow
`startangle`	Rotates pie to start from a different angle
`autopct`	Shows percentage text on slices
`wedgeprops`	Customize slice edge, fill, width, etc.

Assignment:
Create a pie chart showing the market share of mobile OS (Android, iOS, others). Then recreate the same using a horizontal bar chart and observe which is easier to understand.

Stack Plots in Matplotlib

Let's Start with a Pie Chart

Before we understand what a stack plot is, let's visualize some simple data

Imagine you surveyed how a group of students spends their after-school time:

```
import matplotlib.pyplot as plt

activities = ['Studying', 'Playing', 'Watching TV', 'Sleeping']
time_spent = [3, 2, 2, 5]  # hours in a day

colors = ['skyblue', 'lightgreen', 'gold', 'lightcoral']

plt.figure(figsize=(6,6))
plt.pie(time_spent, labels=activities, colors=colors, autopct='%1.1f%%',
startangle=90)
plt.title("After School Activities")
plt.axis('equal')  # Equal aspect ratio ensures that pie is drawn as a circle.
plt.show()
```

Interpretation:

- **Studying**: 3 hours
- **Playing**: 2 hours
- **Watching TV**: 2 hours
- **Sleeping**: 5 hours

This pie chart shows how a single student spends time **in one day**.

What is a Stack Plot?

Now, let’s imagine you surveyed **multiple days**, and you want to track how the time spent on each activity changes over a week.

A **Stack Plot** is a type of area chart that helps visualize **multiple quantities over time**, stacked on top of each other. It's especially useful to see **how individual parts contribute to a whole over time**.

Use Cases:

- Time spent on different activities over days
- Distribution of tasks by team members over a project timeline
- Website traffic sources over a week

Stack Plot Example

```python
import matplotlib.pyplot as plt

days = [1, 2, 3, 4, 5, 6, 7]  # Days of the week
studying = [3, 4, 3, 5, 4, 3, 4]
playing = [2, 2, 1, 1, 2, 3, 2]
watching_tv = [2, 1, 2, 2, 1, 1, 1]
sleeping = [5, 5, 6, 5, 6, 5, 5]

labels = ['Studying', 'Playing', 'Watching TV', 'Sleeping']
colors = ['skyblue', 'lightgreen', 'gold', 'lightcoral']

plt.figure(figsize=(10,6))
plt.stackplot(days, studying, playing, watching_tv, sleeping,
              labels=labels, colors=colors, alpha=0.8)

plt.legend(loc='upper left')  # Location of the legend
plt.title('Weekly Activity Tracker')
plt.xlabel('Day')
plt.ylabel('Hours')
plt.grid(True)
plt.show()
```

Key Parameters in `stackplot()`

Parameter	Description
`x`	The x-axis data (like days)
`*args`	Multiple y-values (like studying, playing, etc.)
`labels`	Labels for the legend
`colors`	List of colors for each stack
`alpha`	Transparency level (0 to 1)
`loc` (in `legend`)	Position of the legend (`'upper left'`, `'best'`, etc.)

Summary

- Use **pie charts** for a snapshot in time.
- Use **stack plots** to see how data changes over time while still showing parts of a whole.
- Customize your plot using parameters like `colors`, `labels`, `alpha`, and `legend`.

Stack plots are a great way to **tell a story over time**.

Quick quiz

Try making a stackplot with your own weekly schedule!

Histograms in Matplotlib

A **histogram** is a type of plot that shows the distribution of a dataset. It's especially useful for visualizing the **frequency of numerical data** within specified ranges (called *bins*).

Why and When to Use Histograms?

Use histograms when:

- You want to **understand the distribution** of a numerical dataset (e.g. age, salary, views).
- You want to **detect skewness**, outliers, or **understand spread**.
- You're **binning continuous data** into intervals.

Examples include:

- Analyzing the age of your YouTube viewers

- Understanding test scores distribution
- Checking if data is normally distributed

Understanding the `bins` Argument

The `bins` argument controls how the data is grouped:

- If an **integer**, it defines the number of equal-width bins.
- If a **list**, it defines custom bin **edges**, allowing you to control the range and width of each bin.

```
plt.hist(data, bins=10)  # 10 equal-width bins
plt.hist(data, bins=[10, 20, 30, 40, 60, 100])  # Custom age bins
```

`edgecolor` for Better Visibility

The `edgecolor` parameter adds borders to the bars, improving clarity:

```
plt.hist(data, bins=10, edgecolor='black')
```

Example: Age Distribution of YouTube Viewers

Here is age data for some viewers:

```
import matplotlib.pyplot as plt
import numpy as np

ages = [
    34, 28, 36, 45, 27, 27, 45, 37, 25, 35,
    25, 25, 32, 10, 12, 24, 19, 33, 20, 15,
    44, 27, 30, 15, 24, 31, 18, 33, 23, 27,
    23, 48, 29, 19, 38, 17, 32, 10, 16, 31,
    37, 31, 28, 26, 15, 22, 25, 40, 33, 12,
    33, 26, 23, 36, 40, 39, 21, 26, 33, 39,
    25, 28, 18, 18, 38, 43, 29, 40, 33, 23,
    33, 45, 29, 45, 3, 38, 30, 27, 30, 10,
    27, 33, 44, 24, 21, 24, 39, 33, 24, 35,
    30, 39, 22, 26, 26, 15, 32, 32, 30, 27
]

bins = [10, 20, 30, 40, 50, 60, 70]

plt.hist(ages, bins=bins, edgecolor='black')
plt.title('Age Distribution of YouTube Viewers')
plt.xlabel('Age Group')
plt.ylabel('Number of Viewers')
plt.grid(True, linestyle='--', alpha=0.5)
plt.show()
```

Adding a Vertical Line: `axvline`

Use `axvline` to mark a specific value like the **average age** or a **threshold**.

```
plt.hist(ages, bins=bins, edgecolor='black')
plt.axvline(np.mean(ages), color='red', linestyle='--', linewidth=2,
label='Average Age')
plt.legend()
plt.title('Age Distribution with Mean Line')
plt.show()
```

Summary

Parameter	Purpose
`bins`	Number or custom edges of bins
`edgecolor`	Color around each bar
`axvline`	Vertical reference line

Scatter Plot in Matplotlib

Scatter plots are used to show the relationship between two variables. Let's assume we are plotting the **study hours vs. exam scores** for a group of students.

1. Basic Scatter Plot

```
import matplotlib.pyplot as plt

# Sample data
study_hours = [1, 2, 3, 4, 5, 6, 7, 8, 9]
exam_scores = [40, 45, 50, 55, 60, 65, 75, 85, 90]

plt.scatter(study_hours, exam_scores)
plt.title('Study Hours vs Exam Score')
plt.xlabel('Study Hours')
plt.ylabel('Exam Score')
plt.grid(True)
plt.show()
```

> This shows a simple scatter plot. You can notice a general upward trend — more study hours lead to better scores.

---

2. Adding Color and Size

```
# Size of points based on score (bigger score -> bigger point)
sizes = [score * 2 for score in exam_scores]
colors = ['red' if score < 60 else 'green' for score in exam_scores]

plt.scatter(study_hours, exam_scores, s=sizes, c=colors)
plt.title('Colored & Sized Scatter Plot')
plt.xlabel('Study Hours')
plt.ylabel('Exam Score')
plt.grid(True)
plt.show()
```

> Points are colored based on performance and scaled in size by score. Red means low score, green means good.

---

3. Using a Colormap

```
import numpy as np

# Also works with Numpy Arrays
scores_normalized = np.array(exam_scores)

plt.scatter(study_hours, exam_scores, c=scores_normalized, cmap='viridis')
plt.colorbar(label='Score')
plt.title('Scatter Plot with Colormap')
plt.xlabel('Study Hours')
plt.ylabel('Exam Score')
plt.grid(True)
plt.show()
```

> Google: https://matplotlib.org/stable/users/explain/colors/colormaps.html
>
> `cmap` adds gradient coloring based on the score. `colorbar` helps understand what the colors represent.

---

4. Adding Annotations

```
plt.scatter(study_hours, exam_scores)

# Add labels
```

```
for i in range(len(study_hours)):
    plt.annotate(f'Student {i+1}', (study_hours[i], exam_scores[i]))

plt.title('Scatter Plot with Annotations')
plt.xlabel('Study Hours')
plt.ylabel('Exam Score')
plt.grid(True)
plt.show()
```

> Annotations help identify individual points, which is useful in small datasets.

---

5. Multiple Groups in One Plot

```
# Assume two groups: Class A and Class B
class_a_hours = [2, 4, 6, 8]
class_a_scores = [45, 55, 65, 85]

class_b_hours = [1, 3, 5, 7, 9]
class_b_scores = [40, 50, 60, 70, 90]

plt.scatter(class_a_hours, class_a_scores, label='Class A', color='blue')
plt.scatter(class_b_hours, class_b_scores, label='Class B', color='orange')

plt.title('Scatter Plot: Class A vs Class B')
plt.xlabel('Study Hours')
plt.ylabel('Exam Score')
plt.legend()
plt.grid(True)
plt.show()
```

> When comparing two datasets, use different colors and a legend for clarity.

Subplots in Matplotlib

Subplots allow you to show **multiple plots in a single figure**, side-by-side or in a grid layout. This is helpful when comparing different datasets or aspects of the same data.

---

1. Basic Subplot (1 row, 2 columns)

```
import matplotlib.pyplot as plt

# Data
x = [1, 2, 3, 4, 5]
y1 = [i * 2 for i in x]
y2 = [i ** 2 for i in x]
```

```
# Create a figure with 1 row and 2 columns
plt.subplot(1, 2, 1)  # (rows, cols, plot_no)
plt.plot(x, y1)
plt.title('Double of x')

plt.subplot(1, 2, 2)
plt.plot(x, y2)
plt.title('Square of x')

plt.tight_layout()
plt.show()
```

> `plt.subplot(1, 2, 1)` means 1 row, 2 columns, and we're plotting in the 1st subplot.

---

2. 2×2 Grid of Subplots

```
# More variations of x
y3 = [i ** 0.5 for i in x]
y4 = [10 - i for i in x]

plt.figure(figsize=(8, 6))  # Optional: make it bigger

plt.subplot(2, 2, 1)
plt.plot(x, y1)
plt.title('x * 2')

plt.subplot(2, 2, 2)
plt.plot(x, y2)
plt.title('x squared')

plt.subplot(2, 2, 3)
plt.plot(x, y3)
plt.title('sqrt(x)')

plt.subplot(2, 2, 4)
plt.plot(x, y4)
plt.title('10 - x')

plt.tight_layout()
plt.show()
```

> This lays out 4 plots in a 2x2 grid. `plt.tight_layout()` avoids overlapping titles and labels.

---

3. Using `plt.subplots()` for Clean Code

```
fig, axs = plt.subplots(1, 2, figsize=(10, 4))

axs[0].plot(x, y1)
axs[0].set_title('x * 2')

axs[1].plot(x, y2)
axs[1].set_title('x squared')

fig.suptitle('Simple Comparison Plots', fontsize=14)
fig.tight_layout()
fig.subplots_adjust(top=0.85)  # So title doesn't overlap
fig.savefig('my_plots.png')    # Save as image

plt.show()
```

So to summarize:

- axs is for working on individual plots
- fig is for settings that apply to the whole figure

> `plt.subplots()` returns a figure and a list/array of axes objects. This is more flexible and cleaner, especially for loops or advanced customizations.

---

4. Looping Over Subplots

```
fig, axs = plt.subplots(2, 2, figsize=(8, 6))
ys = [y1, y2, y3, y4]
titles = ['x * 2', 'x squared', 'sqrt(x)', '10 - x']

for i in range(2):
    for j in range(2):
        idx = i * 2 + j
        axs[i, j].plot(x, ys[idx])
        axs[i, j].set_title(titles[idx])

plt.tight_layout()
plt.show()
```

> This approach works well when dealing with dynamic or repetitive data series.

Introduction to Seaborn

Seaborn is a Python library built on top of Matplotlib that makes it **easier** and **prettier** to create complex, beautiful visualizations.

Why Seaborn?

- Matplotlib is powerful but very **low-level**.
- Seaborn adds **high-level** features like **automatic styling, themes, color palettes**, and **dataframe integration**.
- Seaborn comes with built in Datasets
- Makes complex plots (like **boxplots**, **violin plots**, **heatmaps**, **pairplots**) **very easy**.

In short:

- Less code
- Better-looking graphs
- Easy handling of DataFrames (like from pandas)

Basic Setup

```
# Install Seaborn if you don't have it
!pip install seaborn

# Import Seaborn
import seaborn as sns
import matplotlib.pyplot as plt
```

Seaborn Themes

Seaborn automatically makes your plots look good, but you can even control the overall "theme."

```
sns.set_theme(style="darkgrid")  # Options: whitegrid, dark, white, ticks
```

Example:

```
import numpy as np

x = np.array([0, 2, 3, 4, 5, 6, 7, 8, 9 ,10, 60])
y = np.sin(x)

sns.lineplot(x=x, y=y)
plt.title('Beautiful Line Plot')
plt.show()
```

Summary

- Seaborn makes complex, attractive, and statistical plots simple and ready for professional reports.

- Matplotlib is important to know for fine-tuning or customization, but Seaborn should be your first choice for day-to-day plotting.

- In real-world Data Science projects, Seaborn saves hours of manual work by offering higher-level, smarter defaults.

Basic Plot Types in Seaborn

- **Seaborn** comes with **built-in example datasets**.
- These are small real-world datasets like restaurant tips, flight passenger counts, iris flower measurements, etc.
- They are mainly used for **practice**, **examples**, and **learning** plotting techniques without needing to manually download any data.

You can **load** these datasets directly into a **pandas DataFrame** using `sns.load_dataset()`.

How to See All Available Datasets

```
import seaborn as sns

print(sns.get_dataset_names())
```

This will list dataset names like: `tips`, `flights`, `iris`, `diamonds`, `penguins`, `titanic`, etc.

How to Load a Dataset

```
tips = sns.load_dataset('tips')
print(tips.head())
```

- This loads the **"tips"** dataset (restaurant bills and tips).
- It returns a **pandas DataFrame** ready for analysis or plotting.

Why Seaborn Provides Datasets

- To **quickly test** different types of plots
- To **learn plotting** without needing your own data at first
- To create **examples** and **tutorials** easily
- To show real-world messy data handling (missing values, categorical data, etc.)

Now lets look into some plots we can create usign Seaborn

1. Line Plot

```
tips = sns.load_dataset('tips')

sns.lineplot(x="total_bill", y="tip", data=tips)
plt.title('Line Plot Example')
plt.show()
```

---

2. Scatter Plot

```
sns.scatterplot(x="total_bill", y="tip", data=tips, hue="time")
plt.title('Scatter Plot with Color by Time')
plt.show()
```

---

3. Bar Plot

```
sns.barplot(x="day", y="total_bill", data=tips)
plt.title('Average Bill per Day')
plt.show()
```

---

4. Box Plot

Boxplots show distributions, medians, and outliers in one simple plot.

```
sns.boxplot(x="day", y="total_bill", data=tips)
plt.title('Boxplot of Total Bill per Day')
plt.show()
```

---

5. Heatmap (Correlation Matrix)

```
flights = sns.load_dataset('flights')
pivot_table = flights.pivot("month", "year", "passengers")

sns.heatmap(pivot_table, annot=True, fmt="d", cmap="YlGnBu")
plt.title('Heatmap of Passengers')
plt.show()
```

Working with Pandas DataFrames

One of the biggest strengths of Seaborn:

```
import pandas as pd

df = pd.DataFrame({
    "age": [22, 25, 47, 52, 46, 56, 55, 60, 34, 43],
    "salary": [25000, 27000, 52000, 60000, 58000, 62000, 61000, 65000, 38000,
45000],
    "gender": ["M", "F", "M", "F", "F", "M", "M", "F", "F", "M"]
})

sns.scatterplot(x="age", y="salary", hue="gender", data=df)
plt.title('Salary vs Age Scatter Plot')
plt.show()
```

Summary

Feature	**Matplotlib**	**Seaborn**
Default Styles	Basic	Beautiful
Syntax Level	Low	High
Works with DataFrames	Manual	Easy
Plotting Complex Graphs	Tedious	Very Easy

Final Words

- **Seaborn** makes data visualization **faster, prettier, and smarter**.
- You should still know Matplotlib basics (for fine-tuning plots).
- In real-world Data Science, we usually **start with Seaborn**, and **customize with Matplotlib**.

Data Collection Techniques

In the world of data science, data collection is the cornerstone of all analytical efforts. As the saying goes, "Garbage In, Garbage Out"—the quality of insights directly depends on the quality of data collected. Whether you're building machine learning models, performing statistical analysis, or developing AI systems, your success begins with gathering accurate, reliable data.

Understanding Data Sources

Data can come from a variety of sources:

- **APIs** provide structured access to real-time information, such as weather or stock data.
- **Databases** (both SQL and NoSQL) house large volumes of structured data.
- **Webpages** offer valuable unstructured data from blogs, e-commerce platforms, and more.
- **Files** like CSV, JSON, and Excel sheets serve as local or cloud-based data repositories.
- **Sensors and logs** provide real-time inputs from IoT devices and application events.

Ethical Considerations

Ethical data collection is crucial. Always review a website’s Terms of Service and respect `robots.txt` when scraping. Data must be collected with informed consent and in compliance with regulations such as GDPR and CCPA. Attribution, licensing, and data usage rights should never be overlooked.

Tools and Techniques

A variety of tools support data collection:

- **APIs** (REST or GraphQL) for structured, remote data access.
- **Web Scraping** using tools like BeautifulSoup, Scrapy, or Selenium for HTML parsing.
- **Webhooks** for event-based data (e.g., payment notifications).
- **Manual methods** like forms or surveys for crowd-sourced insights.
- **File handling** with Python libraries such as `pandas`, `csv`, and `json`.

For working with databases, Python libraries like `sqlite3`, `SQLAlchemy`, and `pymongo` are widely used to interact with both relational and NoSQL systems.

Final Takeaway

Data collection is more than just gathering information—it’s about doing so responsibly, efficiently, and with the right tools. Python stands out as a versatile language, offering powerful libraries for virtually every type of data source. Mastering these techniques lays a strong foundation for any data science project.

What is Web Scraping?

- **Web Scraping** is the automated process of extracting information from websites.
- Instead of manually copying data, a scraper **reads** and **parses** the webpage's code to collect the needed data.
- Used widely in data science for collecting:

  - Product prices
  - News articles
  - Job listings
  - Research datasets

Why Use Web Scraping?

- **Automate** repetitive data collection tasks
- Access data that is **not available via APIs**
- Build datasets for **Machine Learning models** and **Analytics**
- Monitor websites for **price changes**, **content updates**, or **news alerts**

When Not to Use Web Scraping

- If the site offers an **official API**, prefer that (it's cleaner and more stable).
- If scraping **violates** the site's **Terms of Service**.
- If the scraping **harms** the website (excessive requests can cause server overload).

HTML Basics for Web Scraping

What is HTML?

- **HTML (HyperText Markup Language)** structures content on the web.
- It's made up of **elements** (tags) like `<div>`, `<p>`, `<h1>`, `<a>`, etc.

Example:

```html
<html>
  <body>
    <h1>Product Title</h1>
    <p class="price">$29.99</p>
    <a href="/buy-now">Buy Now</a>
  </body>
</html>
```

Important HTML Elements for Scraping

Tag	Meaning	Common Use
`<div>`	Division/Container	Group content
`<p>`	Paragraph	Text blocks
`<h1>`, `<h2>`, etc.	Headings	Titles and sections
`<a>`	Anchor (links)	URLs, navigation

Tag	Meaning	Common Use
`<img>`	Image	Pictures and icons
`<table>`	Table	Structured tabular data

Attributes Matter!

- HTML tags often have **attributes** like `id`, `class`, `href`, `src`.
- We use these attributes to **target** the correct elements.

Example:

```
<p class="price">$29.99</p>
```

Here, the `class="price"` attribute helps identify the price on the page.

Quick CSS Basics for Scraping

What is CSS?

- **CSS (Cascading Style Sheets)** controls the style and layout of HTML elements.
- For scraping, we mainly care about **CSS selectors** to **find and extract** data.

Common CSS Selectors

Selector	Meaning	Example
`.class`	Selects elements by class	`.price`, `.title`
`#id`	Selects an element by id	`#main`, `#product-title`
`tag`	Selects all elements of a tag	`h1`, `div`, `p`
`tag.class`	Selects a tag with class	`p.price`, `div.container`

Example: Selecting Elements

HTML:

```
<p class="price">$29.99</p>
```

CSS selector to target this:

```
p.price
```

In Python using BeautifulSoup:

```
soup.select_one("p.price")
```

Tools We'll Use for Web Scraping

- `requests` → To download the HTML content of a webpage.
- `BeautifulSoup` → To parse and extract data from the HTML.
- **(Optional)** `Selenium` → For websites that load data dynamically with JavaScript.

Summary

- Web scraping automates data extraction from websites.
- Understanding basic **HTML structure** and **CSS selectors** is crucial.
- Python provides powerful libraries to make web scraping easy.

What is HTML

HTML (HyperText Markup Language) is the standard language used to structure content on the web. When you load a web page, your browser interprets HTML to display the layout, text, images, and other content.

Basic Structure of an HTML Document

```
<!DOCTYPE html>
<html>
  <head>
    <title>Sample Page</title>
  </head>
  <body>
    <h1>Welcome!</h1>
    <p>This is a sample page.</p>
  </body>
</html>
```

Key Elements

- `<!DOCTYPE html>`: Declares the document type.
- `<html>`: Root element of the page.
- `<head>`: Contains metadata, styles, and scripts.
- `<body>`: Contains visible content.

Common HTML Tags Used in Scraping

Most Frequently Encountered Tags

Tag	Purpose
`<div>`	Section or container for content
`<span>`	Inline container
`<a>`	Anchor tag for hyperlinks
`<img>`	Displays images (uses `src`)
`<ul>`, `<ol>`, `<li>`	Lists and list items
`<table>`, `<tr>`, `<td>`	Tables and cells
`<h1>` to `<h6>`	Headers (various sizes)
`<p>`	Paragraph text
`<form>`, `<input>`, `<button>`	Form elements

Attributes in HTML

HTML tags often include attributes that provide metadata or instructions:

```
<a href="https://example.com" class="nav-link">Visit Site</a>
```

Common Attributes

- `href`: Hyperlink reference
- `src`: Image or media source
- `class`: CSS class (commonly used for scraping)
- `id`: Unique identifier for an element
- `name`: Often used in form elements
- `type`: Used in `<input>` tags

Navigating HTML Structure in Scraping

Scraping tools like BeautifulSoup and Selenium use tag names and attributes to locate elements.

Example Targets

- By tag: `soup.find('div')`
- By class: `soup.find('div', class_='product')`
- By ID: `soup.find(id='header')`
- By attribute: `soup.find('a', {'href': True})`

Understanding Nested Elements

HTML is hierarchical. Tags can contain other tags.

```
<div class="article">
  <h2>Title</h2>
  <p>This is a summary.</p>
</div>
```

To extract both the title and summary, first locate the parent `<div class="article">`, then its children.

Practical Tips

- Use browser DevTools (Right-click > Inspect) to examine HTML structure.
- Target elements with unique `id` or descriptive `class` attributes.
- Use tag nesting logic to extract specific parts of a page.

Useful Python Libraries for HTML Parsing

- `requests`: For sending HTTP requests
- `BeautifulSoup`: For parsing and traversing HTML
- `lxml`: Fast parser for large documents
- `Selenium`: For interacting with JavaScript-rendered pages

Using Requests for Web Scraping

Today we will see how to scrape websites and use requests module to download the raw html of a webpage. In this section we can safely use https://quotes.toscrape.com/ and https://books.toscrape.com/ for scraping demos

1. What is `requests`?

- `requests` is a Python library used to send HTTP requests easily.
- It allows you to fetch the content of a webpage programmatically.
- It is commonly used as the first step before parsing HTML with BeautifulSoup.

---

2. Installing `requests`

To install `requests`, run:

```
pip install requests
```

---

3. Sending a Basic GET Request

Example

```python
import requests

url = "https://example.com"
response = requests.get(url)

# Print the HTML content
print(response.text)
```

Key points:

- `url`: The website you want to fetch.
- `response.text`: The HTML content of the page as a string.

---

4. Checking the Response Status

Always check if the request was successful:

```python
print(response.status_code)
```

Common Status Codes

- `200`: OK (Success)
- `404`: Not Found
- `403`: Forbidden
- `500`: Internal Server Error

Good practice:

```python
if response.status_code == 200:
    print("Page fetched successfully!")
else:
    print("Failed to fetch the page.")
```

---

5. Important Response Properties

Property	Description
`response.text`	HTML content as Unicode text
`response.content`	Raw bytes of the response
`response.status_code`	HTTP status code

Property	Description
`response.headers`	Metadata like content-type, server info

6. Adding Headers to Mimic a Browser

Sometimes websites block automated requests. Adding a `User-Agent` header helps the request look like it is coming from a real browser.

```
headers = {
    "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64)"
}

response = requests.get(url, headers=headers)
```

7. Handling Connection Errors

Wrap your request in a try-except block to handle errors gracefully:

```
try:
    response = requests.get(url, timeout=5)
    response.raise_for_status()  # Raises an HTTPError for bad responses
    print(response.text)
except requests.exceptions.RequestException as e:
    print(f"An error occurred: {e}")
```

8. Best Practices for Fetching Pages

- Always check the HTTP status code.
- Use proper headers to mimic a browser.
- Set a timeout to avoid hanging indefinitely.
- Respect the website by not making too many rapid requests.

9. Summary

- `requests` makes it simple to fetch web pages using Python.
- It is the starting point for most web scraping workflows.
- Combining `requests` with BeautifulSoup allows for powerful data extraction.

Using BeautifulSoup for Web Scraping

1. What is BeautifulSoup?

- **BeautifulSoup** is a Python library used to parse HTML and XML documents.
- It creates a **parse tree** from page content, making it easy to extract data.
- It is often used with `requests` to scrape websites.

---

2. Installing BeautifulSoup

Install both **`beautifulsoup4`** and a parser like **`lxml`**:

```
pip install beautifulsoup4 lxml
```

---

3. Creating a BeautifulSoup Object

Example

```
from bs4 import BeautifulSoup
import requests

url = "https://example.com"
response = requests.get(url)

soup = BeautifulSoup(response.text, "lxml")
```

- `response.text`: HTML content.
- `"lxml"`: A fast and powerful parser (you can also use `"html.parser"`).

---

4. Understanding the HTML Structure

BeautifulSoup treats the page like a tree.
You can search and navigate through **tags**, **classes**, **ids**, and **attributes**.

Example HTML:

```
<html>
  <body>
    <h1>Title</h1>
    <p class="description">This is a paragraph.</p>
    <a href="/page">Read more</a>
  </body>
</html>
```

---

5. Common Methods in BeautifulSoup

5.1 Accessing Elements

- Access the **first occurrence** of a tag:

```
soup.h1
```

- Get the **text** inside a tag:

```
soup.h1.text
```

5.2 `find()` Method

- Finds the **first matching** element:

```
soup.find("p")
```

- Find a tag with specific attributes:

```
soup.find("p", class_="description")
```

5.3 `find_all()` Method

- Finds **all matching** elements:

```
soup.find_all("a")
```

5.4 Using `select()` and `select_one()`

- Select elements using **CSS selectors**.

```
soup.select_one("p.description")
```

```
soup.select("a")
```

---

6. Extracting Attributes

Get the value of an attribute, such as `href` from an `<a>` tag:

```
link = soup.find("a")
print(link["href"])
```

Or using `.get()`:

```
print(link.get("href"))
```

---

7. Traversing the Tree

- Access parent elements:

```
soup.p.parent
```

- Access children elements:

```
list(soup.body.children)
```

- Find the next sibling:

```
soup.h1.find_next_sibling()
```

---

8. Handling Missing Elements Safely

Always check if an element exists before accessing it:

```
title_tag = soup.find("h1")
if title_tag:
    print(title_tag.text)
else:
    print("Title not found")
```

---

9. Summary

- BeautifulSoup helps parse and navigate HTML easily.
- Use `.find()`, `.find_all()`, `.select()`, and `.select_one()` to locate data.
- Always inspect the website's structure before writing scraping logic.

- Combine BeautifulSoup with `requests` for full scraping workflows.

Introduction to Databases

What is a Database?

A **Database** is an organized collection of structured information or data, typically stored electronically in a computer system. Its a way to store data in a format that is easily accessible

> Example: Think of a library where books are organized by topic, author, and title – that's essentially what a database does with data.

Why Do We Need Databases?

- To **store**, **manage**, and **retrieve** large amounts of data efficiently.
- To **prevent data duplication** and maintain **data integrity**.
- To allow **multiple users** to access and manipulate data simultaneously.

What is SQL?

SQL (Structured Query Language) is the standard programming language used to communicate with and manipulate databases.

Common operations using SQL:

- `INSERT` – Add new records (CREATE)
- `SELECT` – Retrieve data (READ)
- `UPDATE` – Modify existing data (UPDATE)
- `DELETE` – Remove records (DELETE)

These operations are usually referred to as CRUD Operations. CRUD stands for Create, Read, Update, and Delete — the four basic operations used to manage data in a database.

Comparison with Excel

Databases and Excel may seem similar at first, but they work differently under the hood.

- In Excel, a **sheet** is like a **table** in a database.
- Each **row** in the sheet is similar to a **record** (or entry) in a database table.
- Each **column** in Excel corresponds to a **field** (or attribute) in a table.
- Excel stores all data in one file, whereas a database can contain multiple related tables.
- In databases, you can define strict data types and rules, which Excel doesn't enforce.
- Unlike Excel, databases allow complex querying, relationships between tables, and secure multi-user access.

> Think of a database as a more powerful, structured, and scalable version of Excel for data management.

Relational vs Non-relational Databases

Relational databases store data in structured tables with predefined schemas and relationships between tables (e.g., MySQL, PostgreSQL). Non-relational databases (NoSQL) use flexible formats like documents, key-value pairs, or graphs, and don't require a fixed schema (e.g., MongoDB, Firebase). Relational is ideal for structured data and complex queries, while non-relational is better for scalability and unstructured data.

Feature	Relational (SQL)	Non-relational (NoSQL)
Structure	Tables (rows & cols)	Documents, Key-Value
Language	SQL	Varies (Mongo Query, etc.)

Feature	Relational (SQL)	Non-relational (NoSQL)
Schema	Fixed schema	Flexible schema
Examples	MySQL, PostgreSQL	MongoDB, Firebase

What is DBMS?

A **Database Management System (DBMS)** is software that interacts with users, applications, and the database itself to capture and analyze data. It allows users to create, read, update, and delete data in a structured way.

- **Examples**: MySQL, PostgreSQL, Oracle Database, SQLite.
- **Functions**: Data storage, retrieval, security, backup, and recovery.

What is MySQL?

MySQL is an open-source relational database management system (RDBMS) that uses SQL.

- Widely used in web development
- High performance and reliability
- Powers platforms like WordPress, Facebook (early days), and YouTube

---

Real-World Use Cases

- **E-commerce websites** to store customer orders and product listings
- **Banking systems** to handle transactions securely
- **Social networks** to manage user data, messages, and posts

---

Summary

- Databases are essential for structured data storage and retrieval.
- SQL is the language used to interact with relational databases.
- MySQL is a popular and powerful SQL-based database system.
- Understanding databases is a must-have skill for any developer or data analyst.

Creating a Database in MySQL

To start working with MySQL, the first step is to create a database.

Syntax

```
CREATE DATABASE database_name;
```

Example

```
CREATE DATABASE student_db;
```

This command creates a new database named `student_db`.

Tips

- Database names should be **unique**.
- Avoid using spaces or special characters.
- Use lowercase and underscores (`_`) for better readability (e.g., `employee_records`).

Viewing All Databases

To see all available databases:

```
SHOW DATABASES;
```

Switching to a Database

Before working with tables, you must select the database:

```
USE student_db;
```

Dropping a Database

To delete an existing database (this action is irreversible):

```
DROP DATABASE database_name;
```

Example

```
DROP DATABASE student_db;
```

> Be very careful! This will permanently delete all data and tables in the database.

Creating a Table in MySQL

Once you have selected a database, you can create tables to organize and store data.

Syntax

```
CREATE TABLE table_name (
  column1 datatype constraints,
  column2 datatype constraints,
  ...
);
```

Example

```
CREATE TABLE students (
  id INT AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(100) NOT NULL DEFAULT 'No Name',
  age INT,
  email VARCHAR(100) UNIQUE,
  admission_date DATE
);
```

Explanation

- `id INT AUTO_INCREMENT PRIMARY KEY` – A unique identifier for each student that auto-increments.
- `name VARCHAR(100) NOT NULL` – Name must be provided.
- `age INT` – Stores numeric values for age.
- `email VARCHAR(100) UNIQUE` – Each email must be different.
- `admission_date DATE` – Stores the date of admission.

Commonly Used Data Types

- `INT` – Whole numbers (e.g., age, quantity)
- `VARCHAR(n)` – Variable-length string (e.g., names, emails)
- `TEXT` – Long text strings (e.g., descriptions)
- `DATE` – Stores date values (YYYY-MM-DD)
- `DATETIME` – Stores date and time values
- `BOOLEAN` – Stores `TRUE` or `FALSE`

Common Constraints

- `PRIMARY KEY` – Uniquely identifies each record
- `NOT NULL` – Ensures the column cannot be left empty
- `UNIQUE` – Ensures all values in a column are different
- `AUTO_INCREMENT` – Automatically increases numeric values
- `DEFAULT` – Sets a default value for the column
- `FOREIGN KEY` – Enforces relationships between tables

View All Tables

```
SHOW TABLES;
```

View Table Structure

```
DESCRIBE students;
```

Viewing Table Data

```
SELECT * FROM students;
```

Tables are the backbone of any relational database. A well-structured table leads to efficient data management and fewer issues later on.

Modifying a Table in MySQL

As your application grows or requirements change, you may need to make changes to existing tables. MySQL provides several `ALTER` and related statements for such modifications.

Renaming a Table

Use the `RENAME TABLE` command to change the name of an existing table.

```
RENAME TABLE old_table_name TO new_table_name;
```

Dropping a Table

To permanently delete a table and all of its data:

```
DROP TABLE table_name;
```

Renaming a Column

To rename a column in an existing table:

```
ALTER TABLE table_name RENAME COLUMN old_column_name TO new_column_name;
```

Dropping a Column

To remove a column from a table:

```
ALTER TABLE table_name DROP COLUMN column_name;
```

Adding a Column

To add a new column to an existing table:

```
ALTER TABLE table_name ADD COLUMN column_name datatype constraints;
```

Example:

```
ALTER TABLE students ADD COLUMN gender VARCHAR(10);
```

Modifying a Column

To change the data type or constraints of an existing column:

```
ALTER TABLE table_name MODIFY COLUMN column_name new_datatype new_constraints;
```

Example:

```
ALTER TABLE students MODIFY COLUMN name VARCHAR(150) NOT NULL;
```

Changing the Order of Columns

To change the order of columns in a table, you can use the `MODIFY` command with the `AFTER` keyword:

```
ALTER TABLE table_name MODIFY COLUMN column_name datatype AFTER
another_column_name;
```

Always review changes on production databases carefully. Use tools like `DESCRIBE table_name` to verify structure before and after modifications.

How to Insert Rows into a Table in MySQL

Inserting data into a MySQL table can be done in two ways: inserting one row at a time or inserting multiple rows at once. Below are the steps to create a new database, create a table, and insert data into it.

1. Create a New Database

```sql
CREATE DATABASE schooldb;
```

2. Select the Database

```sql
USE schooldb;
```

3. Create the `student` Table

```sql
CREATE TABLE student (
    id INT PRIMARY KEY,
    name VARCHAR(100),
    age INT,
    grade VARCHAR(10),
    date_of_birth DATE
);
```

4. Insert Data into the Table

Insert One Row at a Time

```sql
INSERT INTO student (id, name, age, grade, date_of_birth) VALUES (1, 'Ayesha Khan', 16, '10th', '2007-05-15');
INSERT INTO student (id, name, age, grade, date_of_birth) VALUES (2, 'Ravi Sharma', 17, '11th', '2006-03-22');
INSERT INTO student (id, name, age, grade, date_of_birth) VALUES (3, 'Meena Joshi', 15, '9th', NULL);
INSERT INTO student (id, name, age, grade, date_of_birth) VALUES (4, 'Arjun Verma', 18, '12th', NULL);
INSERT INTO student (id, name, age, grade, date_of_birth) VALUES (5, 'Sara Ali', 16, '10th', NULL);
INSERT INTO student (id, name, age, grade, date_of_birth) VALUES (6, 'Karan Mehta', 17, '11th', NULL);
INSERT INTO student (id, name, age, grade, date_of_birth) VALUES (7, 'Tanya Roy', 15, '9th', NULL);
INSERT INTO student (id, name, age, grade, date_of_birth) VALUES (8, 'Vikram Singh', 18, '12th', NULL);
INSERT INTO student (id, name, age, grade, date_of_birth) VALUES (9, 'Anjali Desai', 16, '10th', NULL);
INSERT INTO student (id, name, age, grade, date_of_birth) VALUES (10, 'Farhan Zaidi', 17, '11th', NULL);
```

Insert All Rows at Once

```sql
INSERT INTO student (id, name, age, grade) VALUES
(15, 'Ayesha Khan', 16, '10th'),
(25, 'Ravi Sharma', 17, '11th'),
(35, 'Meena Joshi', 15, '9th'),
(45, 'Arjun Verma', 18, '12th'),
(55, 'Sara Ali', 16, '10th'),
(65, 'Karan Mehta', 17, '11th'),
(75, 'Tanya Roy', 15, '9th'),
(85, 'Vikram Singh', 18, '12th'),
(95, 'Anjali Desai', 16, '10th'),
(105, 'Farhan Zaidi', 17, '11th');
```

5. Verify the Inserted Records

To check all the data in the table:

```
SELECT * FROM student;
```

This will display all the records in the `student` table.

How to Select Data in MySQL

Lets now understand how to query data from a table in MySQL using the `SELECT` statement, filter results using the `WHERE` clause, and apply different comparison operators including how to work with `NULL` values.

---

1. Basic `SELECT` Statement

To retrieve all data from the `student` table:

```
SELECT * FROM student;
```

To retrieve specific columns (e.g., only `name` and `grade`):

```
SELECT name, grade FROM student;
```

---

2. Using the `WHERE` Clause

The `WHERE` clause is used to filter rows based on a condition.

Example: Students in 10th grade

```
SELECT * FROM student WHERE grade = '10th';
```

Example: Students older than 16

```
SELECT * FROM student WHERE age > 16;
```

---

3. Comparison Operators in MySQL

Operator	Description	Example
`=`	Equals	`WHERE age = 16`
`!=`	Not equal to	`WHERE grade != '12th'`
`<>`	Not equal to (alternative)	`WHERE grade <> '12th'`
`>`	Greater than	`WHERE age > 16`
`<`	Less than	`WHERE age < 17`
`>=`	Greater than or equal to	`WHERE age >= 16`
`<=`	Less than or equal to	`WHERE age <= 18`
`BETWEEN`	Within a range (inclusive)	`WHERE age BETWEEN 15 AND 17`
`IN`	Matches any in a list	`WHERE grade IN ('10th', '12th')`
`NOT IN`	Excludes list items	`WHERE grade NOT IN ('9th', '11th')`
`LIKE`	Pattern matching	`WHERE name LIKE 'A%'` (names starting with A)
`NOT LIKE`	Pattern not matching	`WHERE name NOT LIKE '%a'` (names not ending in a)

---

4. Handling `NULL` Values

What is `NULL`?

`NULL` represents missing or unknown values. It is **not** equal to `0`, empty string, or any other value.

Common Mistake (Incorrect):

```sql
-- This will not work as expected
SELECT * FROM student WHERE grade = NULL;
```

Correct Ways to Handle NULL

Condition	Correct Syntax
Is NULL	`WHERE grade IS NULL`
Is NOT NULL	`WHERE grade IS NOT NULL`

Example: Select students with no grade assigned

```sql
SELECT * FROM student WHERE grade IS NULL;
```

Example: Select students who have a grade

```sql
SELECT * FROM student WHERE grade IS NOT NULL;
```

---

5. Combining Conditions

You can use `AND`, `OR`, and parentheses to combine conditions.

Example: Students in 10th grade and older than 16

```
SELECT * FROM student WHERE grade = '10th' AND age > 16;
```

Example: Students in 9th or 12th grade

```
SELECT * FROM student WHERE grade = '9th' OR grade = '12th';
```

Example: Complex conditions

```
SELECT * FROM student
WHERE (grade = '10th' OR grade = '11th') AND age >= 16;
```

---

6. Sorting Results with `ORDER BY`

Sort by age in ascending order:

```
SELECT * FROM student ORDER BY age ASC;
```

Sort by name in descending order:

```
SELECT * FROM student ORDER BY name DESC;
```

---

7. Limiting Results with `LIMIT`

Get only 5 rows:

```
SELECT * FROM student LIMIT 5;
```

Get 5 rows starting from the 3rd (offset 2):

```
SELECT * FROM student LIMIT 2, 5;
```

You're right! The `_` wildcard is very handy for matching **dates**, especially when you're looking for values at a **specific position** (like a specific day or month). Let's expand the section accordingly:

---

8. Using Wildcards with `LIKE`

Wildcards are used with the `LIKE` operator to search for patterns. They're helpful when you're not exactly sure about the full value, or you want to match based on structure or partial content.

There are two wildcards in MySQL:

- `%` – Matches **zero or more characters**
- `_` – Matches **exactly one character**

Example: Names starting with `'A'`

```
SELECT * FROM students
WHERE name LIKE 'A%';
```

This finds any name that **starts with 'A'**, like `Aakash`, `Ananya`, `Aryan`.

Matching Dates with `_` Wildcard

The `_` wildcard is useful for matching specific patterns in **date strings**, especially in `YYYY-MM-DD` format.

Let's say you want to find records from the **5th day of any month**:

```
SELECT * FROM attendance
WHERE date LIKE '____-__-05';
```

Explanation:

- `____` matches any year (4 characters)
- `__` matches any month (2 characters)
- `05` is the 5th day

You’re basically telling MySQL:

> “Give me all rows where the date ends with `-05` — which means the 5th of any month, any year.”

More Date Pattern Examples

Pattern to be Matched	Matches
`'2025-05-%'`	Any day in May 2025
`'2024-12-__'`	All 2-digit days in December 2024
`'____-01-01'`	1st January of any year
`'202_-__-__'`	Any date in the 2020s decade
`'____-__-3_'`	All dates from day 30 to 39 (not valid, but works syntactically)

Quick Recap: `LIKE` Wildcard Matching

Pattern	Meaning
`'A%'`	Starts with A
`'%sh'`	Ends with sh

Pattern	Meaning
`'%ar%'`	Contains “ar”
`'R____'`	5-letter name starting with R
`'____-__-05'`	Dates with day = 05

How to Update Data in a MySQL Table

Today we will learn about how to modify existing data in a table using the `UPDATE` statement. We will see how to use the `SET` keyword and how to use the `WHERE` clause to target specific rows.

1. Basic Syntax

```
UPDATE table_name
SET column1 = value1, column2 = value2, ...
WHERE condition;
```

- `UPDATE` : Specifies the table you want to modify.
- `SET` : Assigns new values to columns.
- `WHERE` : Filters which rows should be updated. **Always include a WHERE clause unless you want to update all rows.**

2. Example Table

Assume the following `student` table:

id	name	age	grade
1	Ayesha Khan	16	10th
2	Ravi Sharma	17	11th
3	Meena Joshi	15	9th

3. Update a Single Row

Example: Change the grade of student with `id = 2` to `12th`

```sql
UPDATE student
SET grade = '12th'
WHERE id = 2;
```

---

4. Update Multiple Columns

Example: Change `age` to 17 and `grade` to ‘10th’ for `id = 3`

```sql
UPDATE student
SET age = 17, grade = '10th'
WHERE id = 3;
```

---

5. Update All Rows

Be careful when updating without a `WHERE` clause.

Example: Set all students to age 18

```sql
UPDATE student
SET age = 18;
```

Warning: This will modify **every row** in the table.

---

6. Conditional Update with Comparison Operators

Example: Promote all students in 9th grade to 10th grade

```
UPDATE student
SET grade = '10th'
WHERE grade = '9th';
```

Example: Increase age by 1 for students younger than 18

```
UPDATE student
SET age = age + 1
WHERE age < 18;
```

---

7. Update Using `IS NULL`

Example: Set default grade to 'Unknown' where grade is NULL

```
UPDATE student
SET grade = 'Unknown'
WHERE grade IS NULL;
```

---

8. Verify the Update

To check the results:

```
SELECT * FROM student;
```

How to Delete Data in a MySQL Table

This guide explains how to remove records from a table using the `DELETE` statement. It covers deleting specific rows using the `WHERE` clause and the consequences of deleting all rows.

---

1. Basic Syntax

```
DELETE FROM table_name
WHERE condition;
```

- `DELETE FROM` : Specifies the table from which to remove rows.
- `WHERE` : Filters which rows should be deleted.

Important: If you omit the `WHERE` clause, **all rows** in the table will be deleted.

---

2. Example Table

Assume the following `student` table:

id	name	age	grade
1	Ayesha Khan	16	10th
2	Ravi Sharma	17	12th
3	Meena Joshi	15	9th

---

3. Delete a Specific Row

Example: Delete student with `id = 2`

```
DELETE FROM student
WHERE id = 2;
```

---

4. Delete Rows Based on a Condition

Example: Delete all students in 9th grade

```
DELETE FROM student
WHERE grade = '9th';
```

---

5. Delete Rows Using Comparison Operators

Example: Delete all students younger than 16

```
DELETE FROM student
WHERE age < 16;
```

---

6. Delete Rows Where a Column is NULL

Example: Delete students with no grade assigned

```
DELETE FROM student
WHERE grade IS NULL;
```

7. Delete All Rows (Use with Caution)

Example: Remove all data from the `student` table

```
DELETE FROM student;
```

This deletes all rows but **retains the table structure**.

8. Completely Remove the Table

To delete the table itself (not just the data), use:

```
DROP TABLE student;
```

This removes both the data and the table structure.

9. Verify After Deletion

Check the contents of the table:

```
SELECT * FROM student;
```

MySQL Tutorial: AUTOCOMMIT, COMMIT, and ROLLBACK

Now, we will explore how MySQL handles transactions using the `AUTOCOMMIT`, `COMMIT`, and `ROLLBACK` statements. Understanding these is essential for maintaining data integrity in your databases, especially in data science workflows where large and complex data operations are common.

What is a Transaction?

A **transaction** is a sequence of one or more SQL statements that are executed as a single unit. A transaction has four key properties, known as ACID:

- **Atomicity**: All or nothing.
- **Consistency**: Valid state before and after.
- **Isolation**: Transactions do not interfere.
- **Durability**: Changes persist after commit.

AUTOCOMMIT

By default, MySQL runs in **autocommit mode**. This means that every SQL statement is treated as a separate transaction and is committed automatically right after it is executed.

Check Autocommit Status

```
SELECT @@autocommit;
```

Disable Autocommit

```
SET autocommit = 0;
```

This allows you to group multiple statements into a transaction manually.

Enable Autocommit

```
SET autocommit = 1;
```

COMMIT

The `COMMIT` statement is used to **permanently save** all the changes made in the current transaction.

Example

```
START TRANSACTION;

UPDATE accounts SET balance = balance - 100 WHERE id = 1;
UPDATE accounts SET balance = balance + 100 WHERE id = 2;

COMMIT;
```

Once committed, the changes are visible to other sessions and are stored permanently in the database.

ROLLBACK

The `ROLLBACK` statement is used to **undo** changes made in the current transaction. It is useful if something goes wrong or a condition is not met.

Example

```
START TRANSACTION;

UPDATE accounts SET balance = balance - 100 WHERE id = 1;

-- An error or condition check fails here
ROLLBACK;
```

After a rollback, all changes since the start of the transaction are discarded.

Summary Table

Statement	Description
AUTOCOMMIT	Automatically commits every query
SET autocommit = 0	Disables autocommit mode
COMMIT	Saves all changes in a transaction
ROLLBACK	Reverts all changes in a transaction

Best Practices

- Always use transactions when performing multiple related operations.
- Disable autocommit when working with critical data updates.
- Rollback if any step in your transaction fails.
- Test your transactions thoroughly before running them on production data.

MySQL Tutorial: Getting Current Date and Time

MySQL provides built-in functions to retrieve the **current date**, **time**, and **timestamp**. These are commonly used in data logging, time-based queries, and tracking records in data science workflows.

---

1. `CURRENT_DATE`

Returns the **current date** in `YYYY-MM-DD` format.

```
SELECT CURRENT_DATE;
```

Example Output:

```
2025-05-02
```

---

2. `CURRENT_TIME`

Returns the **current time** in `HH:MM:SS` format.

```
SELECT CURRENT_TIME;
```

Example Output:

```
14:23:45
```

3. `CURRENT_TIMESTAMP` (or `NOW()`)

Returns the **current date and time**.

```
SELECT CURRENT_TIMESTAMP;
-- or
SELECT NOW();
```

Example Output:

```
2025-05-02 14:23:45
```

This is especially useful for storing creation or update times in records.

4. `LOCALTIME` and `LOCALTIMESTAMP`

These are synonyms for `NOW()` and return the current date and time.

```
SELECT LOCALTIME;
SELECT LOCALTIMESTAMP;
```

These functions return the local date and time of the MySQL server, not the client's time zone.

Important Clarification:

The "local" in LOCALTIME refers to the time zone configured on the MySQL server, not the user's system.

You can check the current server time zone using:

5. Using in Table Inserts

You can use `NOW()` or `CURRENT_TIMESTAMP` to auto-fill date-time columns.

```
INSERT INTO logs (event, created_at)
VALUES ('data_import', NOW());
```

---

6. Date and Time Functions Recap

Function	Returns	Example Output
`CURRENT_DATE`	Date only	`2025-05-02`
`CURRENT_TIME`	Time only	`14:23:45`
`NOW()`	Date and time	`2025-05-02 14:23:45`
`CURRENT_TIMESTAMP`	Date and time	`2025-05-02 14:23:45`
`LOCALTIME`	Date and time	`2025-05-02 14:23:45`

Best Practices

- Use `CURRENT_TIMESTAMP` for record timestamps.
- Use `NOW()` in queries to filter records by current time.
- Avoid relying on system time for business logic; prefer database time for consistency.

SQL Tutorial: Deep Dive into Constraints

Constraints in SQL are rules applied to table columns to enforce data integrity, consistency, and validity. They restrict the type of data that can be inserted into a table and help prevent invalid or duplicate entries.

Why Use Constraints?

- Ensure **data quality** and **reliability**
- Prevent **invalid**, **duplicate**, or **null** data
- Maintain **business rules** directly in the database layer

---

1. `NOT NULL` Constraint

Ensures that a column cannot contain `NULL` values.

```
CREATE TABLE employees (
    id INT NOT NULL,
    name VARCHAR(100) NOT NULL
);
```

Use Case:

Make sure critical fields like `id`, `name`, or `email` are always filled.

---

2. `UNIQUE` Constraint

Ensures that all values in a column are **distinct** (no duplicates).

```
CREATE TABLE users (
    username VARCHAR(50) UNIQUE,
    email VARCHAR(100) UNIQUE
);
```

Use Case:

Prevent duplicate usernames or email addresses.

> Note: A table can have **multiple UNIQUE constraints**, but only **one PRIMARY KEY**.

---

3. `DEFAULT` Constraint

Sets a default value for a column if none is provided during insert.

```
CREATE TABLE products (
    name VARCHAR(100),
    status VARCHAR(20) DEFAULT 'in_stock'
);
```

Use Case:

Auto-fill common values to reduce data entry effort and prevent missing data.

---

4. `CHECK` Constraint

Validates that values in a column meet a specific condition.

```
CREATE TABLE accounts (
    id INT,
    balance DECIMAL(10,2) CHECK (balance >= 0)
);
```

Use Case:

Enforce business rules such as non-negative balances or valid age ranges.

> Note: MySQL versions before 8.0 parsed `CHECK` but did **not enforce** it. From MySQL 8.0 onwards, `CHECK` constraints are **enforced**.

---

5. Naming Constraints

You can give **explicit names** to constraints. This makes them easier to reference, especially when altering or dropping them later.

```
CREATE TABLE students (
    roll_no INT PRIMARY KEY,
    age INT CONSTRAINT chk_age CHECK (age >= 5),
    email VARCHAR(100) UNIQUE
);
```

Benefits of Named Constraints:

- Improves clarity and debugging
- Useful when using `ALTER TABLE` to drop constraints

---

6. Constraint Recap Table

Constraint	Purpose	Enforced By MySQL	Custom Name Support
`NOT NULL`	Disallow null values	Yes	Yes
`UNIQUE`	Disallow duplicate values	Yes	Yes
`DEFAULT`	Set default value if none given	Yes	No
`CHECK`	Enforce value conditions	Yes (MySQL 8.0+)	Yes

Best Practices

- Use constraints to **enforce critical rules** in the database layer.
- Always name important constraints for easier maintenance.
- Prefer constraints over application-side validation for core rules.
- Test `CHECK` constraints carefully to ensure compatibility and enforcement in your MySQL version.

MySQL Foreign Key Tutorial

This guide demonstrates how to create a database, define tables, and use **foreign keys** to establish relationships between them.

---

1. Create a Database

```
CREATE DATABASE school;
USE school;
```

---

2. Create Tables

We'll create two tables:

- `students`
- `classes`

Each student will belong to a class, creating a **one-to-many** relationship (one class has many students).

Create `classes` Table

```
CREATE TABLE classes (
    class_id INT AUTO_INCREMENT PRIMARY KEY,
    class_name VARCHAR(50) NOT NULL
);
```

Create `students` Table

```
CREATE TABLE students (
    student_id INT AUTO_INCREMENT PRIMARY KEY,
    student_name VARCHAR(100) NOT NULL,
    class_id INT,
    FOREIGN KEY (class_id) REFERENCES classes(class_id)
        ON UPDATE CASCADE
        ON DELETE SET NULL
);
```

---

3. Insert Sample Data

Insert into `classes`

```
INSERT INTO classes (class_name) VALUES ('Mathematics'), ('Science'), ('History');
```

Insert into `students`

```
INSERT INTO students (student_name, class_id) VALUES
('Alice', 1),
('Bob', 2),
('Charlie', 1);
```

---

4. Explanation of Foreign Key Behavior

In the `students` table:

- `class_id` is a **foreign key**.
- It references `class_id` in the `classes` table.

- `ON DELETE SET NULL` : If a class is deleted, the related students will have `class_id` set to `NULL` .
- `ON UPDATE CASCADE` : If a class ID changes, it will update automatically in the `students` table.

---

5. View the Relationships

To check the foreign key constraints:

```
SHOW CREATE TABLE students;
```

To see all foreign keys in the current database:

```
SELECT
    table_name,
    column_name,
    constraint_name,
    referenced_table_name,
    referenced_column_name
FROM
    information_schema.key_column_usage
WHERE
    referenced_table_name IS NOT NULL
    AND table_schema = 'school';
```

Understanding `ON UPDATE CASCADE` and `ON DELETE SET NULL`

When you define a **foreign key** in MySQL, you can specify what should happen to the child table when the parent table is **updated** or **deleted**. These are called **referential actions**.

---

1. `ON UPDATE CASCADE`

Definition: If the value in the parent table (i.e., the referenced column) is **updated**, the corresponding foreign key value in the child table is **automatically updated** to match.

Example: Suppose we update a `class_id` in the `classes` table:

```
UPDATE classes SET class_id = 10 WHERE class_id = 1;
```

Then all students in the `students` table whose `class_id` was `1` will automatically be updated to `10`.

---

2. `ON DELETE SET NULL`

Definition: If a row in the parent table is **deleted**, the foreign key in the child table will be set to **NULL** for all matching rows.

Example: If we delete a class from the `classes` table:

```
DELETE FROM classes WHERE class_id = 2;
```

Then all students in the `students` table who were in class `2` will have their `class_id` set to `NULL`, indicating that they are no longer assigned to a class.

Why Use These Options?

- `ON UPDATE CASCADE` is useful when the primary key of the parent table might change (rare but possible).
- `ON DELETE SET NULL` is helpful when you want to **preserve child records** but indicate that the relationship has been broken.

Alternatives

- `ON DELETE CASCADE` : Deletes the child rows when the parent row is deleted.
- `ON DELETE RESTRICT` : Prevents deletion if any child rows exist.
- `ON DELETE NO ACTION` : Same as RESTRICT in MySQL.
- `ON DELETE SET DEFAULT` : Not supported in MySQL (but available in some other DBMSs).

MySQL Joins – A Simple Guide

When we have data split across multiple tables, we use **joins** to combine that data and get the full picture.

Let's say we have two tables:

`students`

id	name
1	Alice
2	Bob

`marks`

student_id	subject	score
1	Math	95
2	Math	88
2	Science	90

Now let's see the most common types of joins.

---

1. INNER JOIN

We are telling MySQL to include only the rows that have matching values in both tables.

```sql
SELECT students.name, marks.subject, marks.score
FROM students
INNER JOIN marks ON students.id = marks.student_id;
```

This will show only students who have marks recorded. If a student has no marks, they will not appear in the result.

---

2. LEFT JOIN (or LEFT OUTER JOIN)

We are telling MySQL to include **all students**, even if they don't have any marks. If there's no match in the `marks` table, it will show `NULL`.

```
SELECT students.name, marks.subject, marks.score
FROM students
LEFT JOIN marks ON students.id = marks.student_id;
```

This is useful when we want to list all students, and show marks only if available.

---

3. RIGHT JOIN (or RIGHT OUTER JOIN)

We are telling MySQL to include **all rows from the right table (`marks`)**, even if the student is missing from the `students` table.

```
SELECT students.name, marks.subject, marks.score
FROM students
RIGHT JOIN marks ON students.id = marks.student_id;
```

This is rarely used unless we expect some marks that don't have a student record.

---

5. CROSS JOIN

We are telling MySQL to combine every row in the first table with every row in the second table.

```
SELECT students.name, marks.subject
FROM students
CROSS JOIN marks;
```

Use this only when you really want **all combinations** – it can produce a lot of rows.

Summary

Join Type	What it does
INNER JOIN	Only rows with a match in both tables
LEFT JOIN	All rows from the left table, with matched data if any
RIGHT JOIN	All rows from the right table, with matched data if any
CROSS JOIN	All combinations of rows from both tables

Using `UNION` in MySQL

The `UNION` operator is used to **combine the result sets of two or more `SELECT` statements** into a single result.

It helps when:

- You're pulling **similar data** from different tables.
- You want to **merge results** from multiple queries into one list.

When `UNION` Works

1. **Same number of columns** in all `SELECT` statements.
2. **Compatible data types** in corresponding columns.
3. Columns will be matched **by position**, not by name.

```
SELECT name, city FROM customers
UNION
SELECT name, city FROM vendors;
```

This combines names and cities from both tables into a single result.

When `UNION` Doesn't Work

- If the number of columns is different:

```
-- This will throw an error
SELECT name, city FROM customers
```

```
UNION
SELECT name FROM vendors;
```

- If the data types don’t match:

```
-- Error if 'age' is an integer and 'name' is a string
SELECT age FROM users
UNION
SELECT name FROM students;
```

MySQL will complain that the columns can’t be matched due to type mismatch.

`UNION` vs `UNION ALL`

By default, `UNION` removes **duplicate rows**. If you want to **keep duplicates**, use `UNION ALL` :

```
SELECT name FROM students
UNION ALL
SELECT name FROM alumni;
```

Use `UNION` if:

- You want a **clean list** without duplicates.

Use `UNION ALL` if:

- You want **performance** and don’t care about duplicates.
- Or you **expect duplicate values** and want to preserve them.

Practical Use Case Example

Let’s say you have two tables: `students_2023` and `students_2024` .

```
SELECT name, batch FROM students_2023
UNION
SELECT name, batch FROM students_2024;
```

This gives a combined list of all students across both years, without duplicates.

Sorting the Combined Result

You can sort the final output using `ORDER BY` at the end:

```
SELECT name FROM students_2023
UNION
SELECT name FROM students_2024
ORDER BY name;
```

MySQL Functions

Functions in MySQL are like built-in tools that help you work with data — whether you're manipulating text, doing math, or working with dates.

Let's explore some commonly used functions with simple examples.

---

1. `CONCAT()` – Join strings together

We are telling MySQL to combine two or more strings.

```
SELECT CONCAT('Hello', ' ', 'World') AS greeting;
-- Output: Hello World
```

You can also use it with columns:

```
SELECT CONCAT(first_name, ' ', last_name) AS full_name FROM users;
```

---

2. `NOW()` – Get the current date and time

We are telling MySQL to give us the current date and time.

```
SELECT NOW();
-- Output: 2025-05-03 14:20:45 (example)
```

---

3. `LENGTH()` – Find length of a string (in bytes)

```
SELECT LENGTH('Harry');
-- Output: 5
```

Useful for validations or checking string size.

---

4. `ROUND()` – Round numbers to a specific number of decimal places

```
SELECT ROUND(12.6789, 2);
-- Output: 12.68
```

---

5. `DATEDIFF()` – Difference between two dates (in days)

```
SELECT DATEDIFF('2025-06-01', '2025-05-01');
-- Output: 31
```

Comprehensive List of Useful MySQL Functions

Function	Description	Example Usage
`CONCAT()`	Combine multiple strings	`CONCAT('A', 'B')` → `'AB'`
`LENGTH()`	Length of a string (in bytes)	`LENGTH('Hi') → 2`

Function	Description	Example Usage
`CHAR_LENGTH()`	Number of characters in a string	`CHAR_LENGTH('हिंदी')` → `5`
`LOWER()`	Convert string to lowercase	`LOWER('MySQL')` → `mysql`
`UPPER()`	Convert string to uppercase	`UPPER('hello')` → `HELLO`
`REPLACE()`	Replace part of a string	`REPLACE('abc', 'b', 'x')` → `axc`
`TRIM()`	Remove leading/trailing spaces	`TRIM('  hello  ')` → `hello`
`NOW()`	Current date and time	`NOW()`
`CURDATE()`	Current date only	`CURDATE()`
`CURTIME()`	Current time only	`CURTIME()`
`DATE()`	Extract date from datetime	`DATE(NOW())`
`MONTHNAME()`	Get month name from date	`MONTHNAME('2025-05-03')` → `May`
`YEAR()`	Extract year from date	`YEAR(NOW())`
`DAY()`	Extract day of month	`DAY('2025-05-03')` → `3`
`DATEDIFF()`	Days between two dates	`DATEDIFF('2025-06-01', '2025-05-01')`
`ROUND()`	Round to decimal places	`ROUND(5.678, 2)` → `5.68`
`FLOOR()`	Round down to nearest whole number	`FLOOR(5.9)` → `5`
`CEIL()`	Round up to nearest whole number	`CEIL(5.1)` → `6`

Function	Description	Example Usage
`ABS()`	Absolute value	`ABS(-10)` → `10`
`MOD()`	Get remainder	`MOD(10, 3)` → `1`
`RAND()`	Random decimal between 0 and 1	`RAND()`
`IFNULL()`	Replace `NULL` with a default value	`IFNULL(NULL, 'N/A')` → `N/A`
`COALESCE()`	Return first non-NULL value in a list	`COALESCE(NULL, '', 'Hello')` → `''`
`COUNT()`	Count rows	`COUNT(*)`
`AVG()`	Average of a numeric column	`AVG(score)`
`SUM()`	Total sum of values	`SUM(score)`
`MIN()`	Smallest value	`MIN(score)`
`MAX()`	Largest value	`MAX(score)`

MySQL Views

A **View** in MySQL is like a **virtual table**. It doesn't store data by itself but instead **shows data from one or more tables** through a saved SQL query.

You can use a view just like a regular table: `SELECT` from it, filter it, join it, etc.

Why Use Views?

- To **simplify complex queries** by giving them a name.
- To **hide sensitive columns** from users.
- To show **only specific rows/columns** from a table.
- To **reuse** common query logic across your app or reports.

Creating a View

Let's say you have an `employees` table with lots of details, but you only want to show public employee info (name, department, and salary).

```
CREATE VIEW public_employees AS
SELECT name, department, salary
FROM employees;
```

You're telling MySQL:

> "Create a view called `public_employees` that shows only name, department, and salary from the `employees` table."

Using a View

Now you can query it like a normal table:

```
SELECT * FROM public_employees;
```

Or even apply filters:

```
SELECT * FROM public_employees
WHERE department = 'IT';
```

Updating a View

You can **replace** a view like this:

```
CREATE OR REPLACE VIEW public_employees AS
SELECT name, department
FROM employees;
```

Dropping (Deleting) a View

```
DROP VIEW public_employees;
```

This removes the view from the database.

Notes

- Views don’t store data. If the underlying table changes, the view reflects that automatically.

- **Not all views are updatable.** Simple views usually are (like those selecting from one table without grouping or joins), but complex ones may not allow `INSERT`, `UPDATE`, or `DELETE`.
- Views can make your queries **cleaner and easier to maintain**.

Example Use Case

You have this query used 5 times across your app:

```
SELECT customer_id, name, total_orders, status
FROM customers
WHERE status = 'active' AND total_orders > 5;
```

Instead of repeating it, just create a view:

```
CREATE VIEW top_customers AS
SELECT customer_id, name, total_orders, status
FROM customers
WHERE status = 'active' AND total_orders > 5;
```

Now just do:

```
SELECT * FROM top_customers;
```

MySQL Indexes

An **index** in MySQL is a data structure that makes **data retrieval faster**—especially when you're using `WHERE`, `JOIN`, `ORDER BY`, or searching large tables.

Think of an index like the index in a book: instead of reading every page, MySQL uses the index to jump straight to the relevant row(s).

Why Use Indexes?

- **Speed up queries** that search, filter, or sort data.
- **Improve performance** for frequent lookups or joins.
- **Enhance scalability** of your database over time.

How to Create an Index

1. Single Column Index

```
CREATE INDEX idx_email ON users(email);
```

You're telling MySQL:

> "Create a quick lookup structure for the `email` column in the `users` table."

2. Multi-column (Composite) Index

```
CREATE INDEX idx_name_city ON users(name, city);
```

This is useful when your query filters on both `name` and `city` in that specific order.

How to Delete (Drop) an Index

```
DROP INDEX idx_email ON users;
```

You're saying:

> "Remove the index named `idx_email` from the `users` table."

When to Use Indexes

Use indexes when:

- A column is often used in `WHERE`, `JOIN`, or `ORDER BY` clauses.
- You're searching by unique fields like `email`, `username`, or `ID`.
- You're filtering large tables for specific values regularly.
- You want to improve performance of lookups and joins.

When Not to Use Indexes

Avoid adding indexes when:

- The table is **small** (MySQL can scan it quickly anyway).
- The column is rarely used in searches or filtering.
- You're indexing a column with **very few unique values** (like a `gender` field with just `'M'` and `'F'`).
- You're inserting or updating **very frequently**—indexes can slow down writes because they also need to be updated.

Viewing Existing Indexes

To list all indexes on a table:

```
SHOW INDEX FROM users;
```

Summary

Action	Syntax Example
Create index	`CREATE INDEX idx_name ON table(column);`
Delete index	`DROP INDEX idx_name ON table;`
List indexes	`SHOW INDEX FROM table;`

Indexes are essential for performance, but overusing them or indexing the wrong columns can actually hurt performance. Use them wisely based on how your data is queried.

Subqueries in MySQL

A **subquery** is a query nested inside another SQL query.
It helps you perform complex filtering, calculations, or temporary data shaping by breaking down the logic into smaller steps.

You can use subqueries in `SELECT`, `FROM`, or `WHERE` clauses.

What is a Subquery?

A subquery is enclosed in parentheses and returns data to be used by the outer query.
It can return: - A single value (scalar) - A row - A full table

---

Subquery in the `WHERE` Clause

Example: Employees who earn more than average

```
SELECT name, salary
FROM employees
WHERE salary > (
    SELECT AVG(salary)
    FROM employees
);
```

We are telling MySQL: “**First calculate the average salary, then return employees with salaries greater than that.**”

Subquery in the `FROM` Clause

Example: Department-wise average salary above 50,000

```
SELECT department, avg_salary
FROM (
    SELECT department, AVG(salary) AS avg_salary
    FROM employees
    GROUP BY department
) AS dept_avg
WHERE avg_salary > 50000;
```

We are telling MySQL: "**Create a temporary table of average salaries by department, then filter departments where the average is above 50,000.**"

Subquery in the `SELECT` Clause

Example: Count of projects per employee

```
SELECT name,
       (SELECT COUNT(*) FROM projects WHERE projects.employee_id = employees.id) AS
project_count
FROM employees;
```

This gives each employee along with the number of projects they are assigned to.

Correlated Subqueries

A **correlated subquery** depends on the outer query. It runs once for **each row** in the outer query.

Example: Employee earning more than department’s average

```
SELECT name, department, salary
FROM employees e
WHERE salary > (
    SELECT AVG(salary)
    FROM employees
    WHERE department = e.department
);
```

We are telling MySQL: “**For each employee, compare their salary with the average salary of their department.**”

Types of Subqueries

Type	Description
Scalar Subquery	Returns a single value
Row Subquery	Returns one row with multiple columns
Table Subquery	Returns multiple rows and columns
Correlated Subquery	Refers to the outer query inside the subquery

---

When to Use Subqueries

- When your logic depends on **calculated values** (like averages or counts)
- When you need to **filter based on dynamic conditions**
- When you’re **breaking down complex queries** for readability

---

When to Avoid Subqueries

- When the same result can be achieved with a **JOIN**, which is often faster
- When the subquery is being **executed repeatedly** for every row (correlated subqueries on large tables)

Summary

Clause	Use Case
`WHERE`	Filter based on the result of a subquery
`FROM`	Use a subquery as a derived table
`SELECT`	Add related calculations inline

Subqueries are powerful for solving multi-step problems and isolating logic, but be mindful of performance when working with large data sets.

GROUP BY in MySQL

The `GROUP BY` clause is used when you want to **group rows that have the same values in specified columns**.
It's usually combined with **aggregate functions** like `COUNT()`, `SUM()`, `AVG()`, `MAX()`, or `MIN()`.

We are telling MySQL:
"Group these rows together by this column, and then apply an aggregate function to each group."

Example: Count of employees in each department

```
SELECT department, COUNT(*) AS total_employees
FROM employees
GROUP BY department;
```

Here, we're grouping all employees **by their department** and counting how many are in each group.

Example: Average salary per department

```
SELECT department, AVG(salary) AS avg_salary
FROM employees
GROUP BY department;
```

We are telling MySQL: "**Group the data by department, then calculate the average salary for each group.**"

Using `GROUP BY` with Multiple Columns

You can group by more than one column to get more detailed groupings.

Example: Count by department and job title

```
SELECT department, job_title, COUNT(*) AS count
FROM employees
GROUP BY department, job_title;
```

This will count how many employees hold each job title within each department.

The `HAVING` Clause

Once you've grouped data using `GROUP BY`, you might want to **filter the groups themselves** based on the result of an aggregate function. This is where `HAVING` comes in.

> `HAVING` is like `WHERE`, but it works **after** the grouping is done.

Example: Departments with more than 5 employees

```
SELECT department, COUNT(*) AS total
FROM employees
GROUP BY department
HAVING COUNT(*) > 5;
```

We are telling MySQL: "**First group employees by department, then only show those departments where the total number is greater than 5.**"

Difference Between `WHERE` and `HAVING`

Clause	Used For	Example Use
`WHERE`	Filters **rows before** grouping	`WHERE salary > 50000`
`HAVING`	Filters **groups after** grouping	`HAVING AVG(salary) > 60000`

You can also use both together:

```
SELECT department, AVG(salary) AS avg_salary
FROM employees
WHERE status = 'active'
GROUP BY department
HAVING AVG(salary) > 60000;
```

Here's what's happening:

1. `WHERE` filters only the active employees.
2. `GROUP BY` groups them by department.
3. `HAVING` filters out departments with low average salary.

Using `WITH ROLLUP` in MySQL

The `WITH ROLLUP` clause in MySQL is used with `GROUP BY` to add **summary rows** (totals and subtotals) to your result set.

Summary

Keyword	Role
`GROUP BY`	Groups rows with same values into summary rows
`HAVING`	Filters groups based on aggregate results

Use `GROUP BY` when you want to **aggregate** data. Use `HAVING` when you want to **filter those aggregated groups**.

Stored Procedures in MySQL

A **Stored Procedure** is a saved block of SQL code that you can execute later by calling its name.
It allows you to group SQL statements and reuse them—just like a function in programming.

Why Use Stored Procedures?

- To avoid repeating the same SQL logic in multiple places
- To improve performance by reducing network traffic
- To encapsulate complex business logic inside the database

Creating a Stored Procedure

When you create a stored procedure, you need to temporarily change the SQL statement delimiter from `;` to something else like `//` or `$$`.

Why change the `DELIMITER`?

MySQL ends a command at the first `;`.
Since stored procedures contain multiple SQL statements (each ending in `;`), we need to tell MySQL **not to end the procedure too early**.
So we temporarily change the delimiter to something else—then switch it back.

Example: Simple Procedure to List All Employees

```
DELIMITER //

CREATE PROCEDURE list_employees()
BEGIN
    SELECT * FROM employees;
END //

DELIMITER ;
```

This creates a procedure named `list_employees`.

Calling a Stored Procedure

You use the `CALL` statement:

```
CALL list_employees();
```

Stored Procedure with Parameters

You can pass values into procedures using the `IN` keyword.

Example: Get details of an employee by ID

```
DELIMITER //

CREATE PROCEDURE get_employee_by_id(IN emp_id INT)
BEGIN
    SELECT * FROM employees WHERE id = emp_id;
END //
```

```
DELIMITER ;
```

Here, `IN emp_id INT` means:

> "Take an integer input called `emp_id` when this procedure is called."

Call it like this:

```
CALL get_employee_by_id(3);
```

Dropping a Stored Procedure

To delete a stored procedure:

```
DROP PROCEDURE IF EXISTS get_employee_by_id;
```

This ensures it doesn't throw an error if the procedure doesn't exist.

Summary

Task	SQL Command
Create Procedure	`CREATE PROCEDURE`
Change Delimiter	`DELIMITER //` (or any unique symbol)
Call a Procedure	`CALL procedure_name();`
With Input Parameter	`IN param_name data_type`
Drop a Procedure	`DROP PROCEDURE IF EXISTS procedure_name;`

Best Practices

- Always give clear names to procedures.
- Use `IN`, `OUT`, or `INOUT` for flexible parameter handling.
- Keep business logic in the database only if it improves clarity or performance.

Introduction to Probability in Data Science

1. What is Probability?

Probability is a measure of the likelihood that a particular event will occur. It ranges from 0 (impossible) to 1 (certain). Probability is used to model uncertanity with data and make informed decisions based on that uncertainty.

Examples:

- Tossing a coin
- Rolling a die
- Predicting customer churn

2. Role of Probability in Data Science

Probability forms the foundation of statistical inference, machine learning, and decision-making under uncertainty. Data scientists use it to:

- Estimate unknown values from sample data
- Evaluate risks and uncertainties
- Make predictions using probabilistic models

3. Types of Probability

Theoretical Probability

Based on known possible outcomes.

- Example: Probability of getting heads in a fair coin = 0.5

Empirical (Experimental) Probability

Based on observed data.

- Example: Probability that a user clicks an ad based on 10,000 past impressions.

---

4. Key Concepts

- **Sample Space (S):** All possible outcomes of an experiment.
- **Event (E):** A subset of the sample space; one or more outcomes.

Example:

For rolling a die:

- Sample space: {1, 2, 3, 4, 5, 6}
- Event (even number): {2, 4, 6}

---

5. Real-World Business Scenarios

- Predicting customer conversion rate from a marketing campaign
- Estimating product failure rate in manufacturing
- Calculating insurance risks
- Detecting fraud in financial transactions

Calculating Probability

Formula:

P(E) = Number of favorable outcomes / Total number of outcomes

Where:

- P(E) is the probability of event E
- Number of favorable outcomes is the count of outcomes that satisfy event E
- Total number of outcomes is the count of all possible outcomes in the sample space

Example:

If you roll a die, the probability of rolling a 3 is:

P(rolling a 3) = Number of favorable outcomes (1) / Total number of outcomes (6) = 1/6 ≈ 0.1667

Quick Quiz

1. What is the probability of rolling a 5 on a fair six-sided die?

2. Two fair six-sided dice are rolled. What is the probability that at least one of the dice shows a 4?

Homework / Practice

1. List three real-world problems in Data Science where probability is involved.
2. Identify the sample space and possible events for each.
3. Try calculating a simple empirical probability using past data (if available).

Types of Experiments

1. **Deterministic –** Always produces the same outcome when repeated under identical conditions.
2. **Probabilistic / Random –** Outcome is uncertain and may vary even if the experiment is repeated under identical conditions.

Sample Space

The sample space is the set of all possible outcomes of a probabilistic experiment.

Examples:
- **Rolling a Die** → { 1, 2, 3, 4, 5, 6 }
- **Tossing 2 Coins** → { (HH), (HT), (TH), (TT) }

For Coins
- Number of possible outcomes = **2^n**
(where **n** is the number of coins)

For Dice
- Number of possible outcomes = **6^n**
(where **n** is the number of dice)

For Cards
A standard deck contains **52 cards**:

- **26 Red Cards**
  - 13 ♥ Hearts → A, 2–10, J, Q, K
  - 13 ♦ Diamonds → A, 2–10, J, Q, K
- **26 Black Cards**
  - 13 ♠ Spades → A, 2–10, J, Q, K
  - 13 ♣ Clubs → A, 2–10, J, Q, K

Conclusion

Understanding the basic types of experiments and the concept of sample space is essential for solving probability problems. Knowing the number of possible outcomes for coins, dice, and cards helps in calculating probabilities accurately. Mastery of these fundamentals forms the foundation for more advanced topics in probability and statistics.

Questions

1. You roll a fair six-sided die. What is the probability of getting an even number?

---

2. Two coins are tossed. Find the sample space associated with this random experiment.

---

3. Two six-sided dice are rolled. What is the probability that the sum is 7?

---

4. A bag contains 3 red, 2 blue, and 5 green balls. What is the probability of randomly selecting a red ball?

---

5. A card is drawn at random from a deck. Find the probability of:
- Getting a heart
- Getting a face card

Basic Rules of Probability

1. Important Definitions

In probability, understanding the basic definitions is crucial for grasping more complex concepts.

- **Experiment**: An action or process that leads to one or more outcomes (e.g., rolling a die). Deterministic experiments have predictable outcomes, while probabilistic or random experiments have uncertain outcomes.
- **Outcome:** A possible result of an experiment (e.g., rolling a 3 on a die).
- **Probability (P):** A measure of the likelihood that an event will occur, ranging from 0 (impossible) to 1 (certain).
- **P(AUB):** The probability of event A or event B occurring.
- **P(A∩B):** The probability of both events A and B occurring.
- **Sample Space (S):** The set of all possible outcomes of an experiment.
- **Event (E):** A subset of the sample space.
- **Mutually Exclusive Events:** Two events that cannot occur at the same time (e.g., rolling a 2 and rolling a 5 on a die). $P(A \cap B)=0$
- **Independent Events:** Two events where the occurrence of one does not affect the other. $P(A \cap B)=P(A) \cdot P(B)$
- **Certain Event:** An event that is guaranteed to happen, with a probability of 1.
- **Impossible Event:** An event that cannot happen, with a probability of 0.
- **Exhaustive Events:** A set of events that cover the entire sample space, meaning at least one of them must occur.

Example:

Tossing a die:

- Sample space: S = {1, 2, 3, 4, 5, 6}
- Event (E): Rolling an even number = {2, 4, 6}

---

2. The Complement Rule

The complement of an event A is the event that A does not occur.

- Notation: A^c
- Rule: $P(A^c) = 1 - P(A)$

Example:

If the probability of rain today is 0.3, the probability it won't rain is: - P(No Rain) = 1 − 0.3 = 0.7

---

3. The Addition Rule

Used to calculate the probability of the union of two events.

For general events:

P(A or B) = P(A) + P(B) − P(A and B)

For mutually exclusive events:

P(A or B) = P(A) + P(B)

Example:

- P(A) = 0.4, P(B) = 0.5, P(A and B) = 0.2
- P(A or B) = 0.4 + 0.5 − 0.2 = 0.7

4. The Multiplication Rule

Used to find the probability that two events occur together.

For Independent events:

P(A and B) = P(A) × P(B)

Example: A event A is rolling a 3 on a first throw of a die, and event B is rolling an even number on second throw.

- P(A) = 1/6 (rolling a 3)
- P(B) = 3/6 (rolling a 2, 4, or 6)
- P(A and B) = P(A) × P(B) = (1/6) × (3/6) = 1/12

For Dependent events:

P(A and B) = P(A) × P(B | A)

Example: A bag contains 5 red and 3 blue balls. Two balls are drawn **without replacement**. Let:

- Event A be drawing a red ball first.
- Event B be drawing a red ball second.
- P(A) = 5/8 (5 red balls out of 8)
- After removing one red ball, there are 4 red left out of 7 total: P(B/A) = 4/7
- P(A and B) = (5/8) X (4/7)= 20/56 = 5/14

So the probability of drawing two red balls **without replacement** is 5/14

5. Independent vs Dependent Events - More examples

- **Independent:** The outcome of one event does not affect the other.
    - Example: Tossing two coins.
- **Dependent:** One event affects the probability of the other.
    - Example: Drawing two cards without replacement.

---

6. Mutually Exclusive Events

Two events are **mutually exclusive** if they cannot happen at the same time.

Example:

- Event A: Rolling a 2
- Event B: Rolling a 5
- These are mutually exclusive because a die can't show both at once.

Summary

This lesson covered key probability rules and concepts that are foundational for more advanced topics like distributions and statistical inference.

- Complement Rule: $P(A^c) = 1 - P(A)$
- Addition Rule for unions
- Multiplication Rule for intersections
- Understanding independence and exclusivity

Homework / Practice

1. A card is drawn from a standard deck. What is the probability of drawing a red card or a queen?
2. If two dice are rolled, what is the probability that both show even numbers?
3. Think of an example from your daily life where two events are dependent.

Practice Questions

1. A box contains 4 red, 3 green, and 2 blue balls. One ball is picked at random.
   What is the probability that the ball is **not green**?

2. Two coins are tossed. What is the probability of getting **at least one head**?

3. A number is chosen at random from the set {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}.
   What is the probability that the number is a **multiple of 3 or 5**?

4. A bag contains 6 white and 4 black balls. Two balls are drawn **with replacement**.
   What is the probability that **both are white**?

5. A student knows the answer to 70% of the questions on a test.
   If one question is selected at random, what is the probability that the student **doesn't know** the answer?

Conditional Probability - An Intuitive Introduction

What is Conditional Probability?

Conditional probability is the probability of an event **A** occurring, given that another event **B** has already occurred. It helps us update our understanding based on new information.

We denote it as:

P(A/B) = P(A and B) / P(B)

This is read as: “The probability of A given B.”

Law of Total Probability

The **Law of Total Probability** helps you find the probability of an event `A` by breaking it down based on other known events `B1`, `B2`, ..., `Bn`.

Formula:

P(A) = P(A/B1) * P(B1) + P(A/B2) * P(B2) + ... + P(A/Bn) * P(Bn)

Conditions to Use This Law:

1. **Partition of the Sample Space**
   The events `B1`, `B2`, ..., `Bn` must:
   1. Be **mutually exclusive** (no overlap):
      No two events can happen at the same time.

   2. Be **collectively exhaustive**:
      Together, they cover the entire sample space.
2. **Known Probabilities**
   You must know:
   1. P(Bi) for each event in the partition.
   2. P(A/Bi), the conditional probability of A given Bi.
3. **P(Bi) > 0**
   The probability of each Bi must be greater than 0, since you can't condition on an impossible event.

Real-life Analogy

Suppose you have a box containing:

- 5 red balls
- 3 blue balls
- 2 green balls

Now, you're told that the ball picked is **not green**. What is the probability that it's red?

Let:

- A = ball is red
- B = ball is not green

Now, out of the 8 balls that are not green (5 red + 3 blue), 5 are red.

So: **P(Red / Not Green) = 5 / 8**

Formula Breakdown

If we know:

- P(A and B): The probability that both A and B happen
- P(B): The probability that B happens

Then: **P(A / B) = P(A and B) / P(B)**

This means: among all the outcomes where B occurs, how many also include A?

Example Problem

A class has 60% boys and 40% girls. 30% of the boys and 10% of the girls play football. If a randomly selected student is a football player, what is the probability that the student is a boy?

Let:

- B = student is a boy → P(B) = 0.6
- G = student is a girl → P(G) = 0.4
- F = student plays football
  - P(F / B) = 0.3
  - P(F / G) = 0.1

Now calculate the total probability that a student plays football:

P(F) = P(F / B) * P(B) + P(F / G) * P(G)

P(F) = 0.3 * 0.6 + 0.1 * 0.4 = 0.18 + 0.04 = 0.22

Now use conditional probability to find P(B / F):

P(B / F) = (P(F / B) * P(B)) / P(F)

P(B / F) = 0.18 / 0.22 ≈ 0.818

So, there's about an 81.8% chance that a football player is a boy.

Common Mistakes to Avoid

- Confusing P(A / B) with P(B / A)
- Forgetting to divide by P(B)
- Assuming P(A / B) is the same as P(B / A) — they are not

Bayes’ Theorem

Bayes’ Theorem is written as:

P(A / B) = [P(B / A) * P(A)] / P(B)

where:

- **P(A / B)** is the probability of event A occurring given that event B has occurred.
- **P(B / A)** is the probability of event B occurring given that event A has occurred.
- **P(A)** is the prior probability of event A.

This can also be interpreted as: P(A / B) = [P(B / A) * P(A)] / [P(B / A) * P(A) + P(B / No A) * P(No A)]

It calculates the probability of event A given that event B has occurred, using the known conditional probability of B given A.

Proof of Bayes’ Theorem

We start with the definition of conditional probability:

1. **P(A / B) = P(A and B) / P(B)**
2. **P(B / A) = P(A and B) / P(A)**

From equation (2), we can write:

P(A and B) = P(B / A) * P(A)

Now substitute this into equation (1):

P(A / B) = [P(B / A) * P(A)] / P(B)

This is Bayes’ Theorem.

Example: Medical Test Problem

Suppose a disease affects 1% of the population. A test for the disease has the following characteristics:

- If a person **has** the disease, the test is **positive** 99% of the time → P(Positive / Disease) = 0.99
- If a person **does not have** the disease, the test is **positive** 5% of the time → P(Positive / No Disease) = 0.05
- The chance of having the disease → P(Disease) = 0.01
- The chance of not having the disease → P(No Disease) = 0.99

A person takes the test and gets a **positive** result. What is the probability they actually have the disease?

Let:

- A = Disease
- B = Positive test

We apply Bayes’ Theorem:

P(Disease / Positive) = [P(Positive / Disease) * P(Disease)] / P(Positive)

First, calculate P(Positive):

P(Positive) = P(Positive / Disease) * P(Disease) + P(Positive / No Disease) * P(No Disease)

P(Positive) = 0.99 * 0.01 + 0.05 * 0.99

P(Positive) = 0.0099 + 0.0495 = 0.0594

Now calculate P(Disease / Positive):

P(Disease / Positive) = 0.0099 / 0.0594 ≈ 0.1667

Conclusion: Even though the test is 99% accurate, the probability that a person actually has the disease given a positive result is only **16.67%**. This is because the disease is very rare, and false positives affect the result significantly.

Probability Distributions

What is a Probability Distribution?

Imagine you're analyzing real-world data — sales, weather, clicks, heights, etc.
You'll notice:

- Some values are more **likely** than others.
- There's a **pattern** to how data is spread.

That pattern is described by a **probability distribution**.

In short:

> **A probability distribution tells you how likely different outcomes are.**

What is a Probability Distribution?

A **probability distribution** is a **mathematical function** or **table** that:

- Assigns **probabilities** to all possible outcomes of a random process.

There are two types:

1. Discrete Probability Distribution

- For outcomes you can **count** (e.g. number of heads in 3 coin tosses).
- Example: **Binomial Distribution**

> Think: "What's the probability I get exactly 2 heads in 3 coin tosses?"

---

2. Continuous Probability Distribution

- For outcomes that can be **any number in a range** (e.g. height, weight).
- Example: **Normal Distribution**

> Think: “What’s the probability someone’s height is between 165 cm and 170 cm?”

A Simple Analogy

Let’s say you roll a die:

- Outcomes = {1, 2, 3, 4, 5, 6}
- Each has a probability of `1/6`

This is a **uniform distribution** (discrete).

Now imagine measuring people’s **heights**:

- You don’t get fixed values.
- Instead, you get a **curve** — most people around average height, fewer very short or very tall.

That’s a **normal distribution** (continuous).

Why Are Distributions Useful in Data Science?

1. **Model real-world randomness** (user behavior, errors, arrivals, etc.)
2. **Make predictions** (how likely is a customer to buy?)
3. **Run simulations**

Summary:

Type	Example	Used For
Discrete	Binomial, Poisson	Count of events
Continuous	Normal, Uniform	Measuring quantities

Uniform Distribution

What It Is:

> A **uniform distribution** is when **every outcome is equally likely**.

Simple Example: Rolling a Fair Die

- Possible outcomes: {1, 2, 3, 4, 5, 6}
- Each number has a **1/6** chance → That’s a **discrete uniform distribution**

Continuous Version:

Let’s say we randomly pick a number between 0 and 1.

- Every value in that range is **equally likely**.
- That’s a **continuous uniform distribution**.

Graphs to Visualize

1. Discrete Uniform (like a die roll)

```
Outcome:     1    2    3    4    5    6
Probability:|---|---|---|---|---|---|
             1/6 for each → flat bars
```

Discrete Uniform Distribution Formula:

P(X = x) = 1 / n

- Where `n` is the number of possible outcomes (e.g., for a 6-sided die, P(rolling a 4) = 1/6)

2. Continuous Uniform (0 to 1)

- It's just a **flat horizontal line** from x = 0 to x = 1
- The probability density is constant (say 1.0) across that interval

Continuous Uniform Distribution Formula:

f(x) = 1 / (b - a) for values between `a` and `b`

- Outside the range `a` to `b`, the probability is 0

Why It's Useful in Data Science

- It models **pure randomness**
- It's used to **simulate random choices**
- Used in Generating random numbers (`np.random.uniform`)

In Code (Python):

```python
import numpy as np
import matplotlib.pyplot as plt

# Continuous uniform from 0 to 1
samples = np.random.uniform(0, 1, 10000)

plt.hist(samples, bins=50, density=True, alpha=0.6, color='skyblue')
plt.title("Continuous Uniform Distribution (0 to 1)")
plt.xlabel("Value")
plt.ylabel("Probability Density")
plt.grid(True)
plt.show()
```

You'll see a **flat histogram** showing uniform probability.

Summary:

Property	Uniform Distribution
Type	Discrete or Continuous
Shape	Flat
Real-world example	Die roll, random number gen
Python function	`np.random.uniform(a, b)`

What is the Binomial Distribution?

> The **binomial distribution** models the number of **successes** in a fixed number of **independent yes/no experiments**, where each has the **same probability** of success.

Think of this:

- Toss a coin 10 times
- What's the probability of getting **exactly 6 heads**?

That's a binomial problem.

Key Ingredients:

- `n` = number of trials (e.g., 10 tosses)
- `p` = probability of success (e.g., 0.5 for heads)
- `x` = number of successes (e.g., 6 heads)

Plain Text Formula:

P(X = x) = C(n, x) × p^x × (1 - p)^(n - x)

Where:

- `C(n, x)` is "n choose x" = combinations = number of ways to pick `x` successes out of `n`
- `p^x` is the probability of `x` successes
- `(1 - p)^(n - x)` is the probability of the remaining being failures

Example:

10 coin tosses, what's the probability of exactly 6 heads?

- n = 10
- x = 6
- p = 0.5

$P(6heads) = C(10,6) * 0.5^6 * 0.5^4 = 210 * (0.015625) * (0.0625) \approx 0.205$

So there's a ~20.5% chance you'll get exactly 6 heads in 10 tosses.

In Python:

```python
from scipy.stats import binom

# Probability of exactly 6 heads in 10 tosses (p = 0.5)
prob = binom.pmf(k=6, n=10, p=0.5)
print(prob)  # Output: ~0.205
```

When to Use:

- Email campaign: Will 40 out of 100 people click the link?
- Quality check: How many out of 10 products will be defective?
- A/B testing: Will 60 out of 200 visitors convert?

Summary:

Concept	Value
Type	Discrete

Concept	Value
Formula	P(X = x) = C(n, x) * p^x * (1 - p)^(n - x)
Python	`scipy.stats.binom.pmf(x, n, p)`
Used for	Count of successes in repeated trials

Normal Distribution (a.k.a. Gaussian Distribution)

Before we dive into the details, let’s understand two key concepts related to normal distribution, mean, and standard deviation.

1. What is Mean?

The **mean** (or average) of a dataset is the sum of all values divided by the number of values.

Formula:

mean = (x1 + x2 + x3 + ... + xn) / n

It represents the central value of the data.

---

2. What is Standard Deviation?

The **standard deviation** measures how spread out the numbers are from the mean.

Steps to calculate standard deviation:

1. Find the mean
2. Subtract the mean from each value and square the result
3. Take the average of these squared differences (this is the variance)
4. Take the square root of the variance

Formula:

standard deviation (sigma) = sqrt((1/n) * sum((xi - mean)^2))

A smaller standard deviation means the data points are close to the mean. A larger standard deviation means the data is more spread out.

What is Normal Distribution?

> A **normal distribution** is a continuous probability distribution that is **bell-shaped and symmetric** around the mean.

It describes variables where:

- Most values cluster around the **average (mean)**.
- Extreme values (very high or low) are **rare**.

Think of:

- Heights of people
- Test scores
- Measurement errors

All tend to follow a **normal curve**.

Key Properties:

- **Mean (μ):** Center of the distribution
- **Standard Deviation (σ):** Spread of the distribution

Shape of the Curve:

- Bell-shaped
- Symmetrical
- Peaks at the mean
- About **68%** of the data lies within **±1σ**, **95%** within **±2σ**, **99.7%** within **±3σ** → This is the famous **68–95–99.7 Rule**

Formula (Plain Text):

f(x) = (1 / (σ * sqrt(2π))) * e^(-(x - μ)^2 / (2σ^2))

Where:

- `μ` = mean
- `σ` = standard deviation
- `e` = Euler's number (≈ 2.718)
- `π` = pi (≈ 3.14159)

This gives the **probability density** for a given value `x`.

> You **don't need to memorize** this formula — but understanding its shape and behavior is essential.

In Python:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm

# Generate values
x = np.linspace(-4, 4, 1000)
mean = 0
std_dev = 1

# Get the probability density
y = norm.pdf(x, loc=mean, scale=std_dev)

# Plot
plt.plot(x, y)
plt.title("Standard Normal Distribution (μ=0, σ=1)")
plt.xlabel("x")
plt.ylabel("Probability Density")
plt.grid(True)

plt.show()
```

Summary

Property	Value
Type	Continuous
Shape	Bell curve
Key parameters	Mean (μ), Std. Dev (σ)
Formula	$f(x) = (1 / (\sigma\sqrt{2\pi})) * e\text{^}(-(x - \mu)^2 / 2\sigma^2)$
Python	`scipy.stats.norm.pdf(x, μ, σ)`

Central Limit Theorem Explained

1. What is the Mean?

The **mean** (or average) of a dataset is the sum of all values divided by the number of values.

Formula:

mean = (x1 + x2 + x3 + ... + xn) / n

It represents the central value of the data.

---

2. What is the Standard Deviation?

The **standard deviation** measures how spread out the numbers are from the mean.

Steps to calculate standard deviation:

1. Find the mean
2. Subtract the mean from each value and square the result
3. Take the average of these squared differences (this is the variance)
4. Take the square root of the variance

Formula:

standard deviation (sigma) = sqrt((1/n) * sum((xi - mean)^2))

A smaller standard deviation means the data points are close to the mean. A larger standard deviation means the data is more spread out.

---

3. Central Limit Theorem (CLT)

The **Central Limit Theorem** states:

> If you take many random samples of size n from any population (with finite mean and variance), then the distribution of the sample means will tend to be **approximately normal** as n becomes large — regardless of the shape of the original population.

---

4. Mathematical Expression

Let X1, X2, ..., Xn be n independent, identically distributed (i.i.d) random variables with:

- Mean = mu
- Standard deviation = sigma

Then the **sampling distribution of the sample mean** (denoted as $\bar{X}$) approaches a normal distribution with:

- Mean = mu
- Standard deviation = sigma / sqrt(n)

History of Machine Learning

Imagine it’s 1936. The world is on the edge of chaos. But in a quiet university in England, a 24-year-old genius named Alan Turing is sitting in his room… thinking.

Not about war. Not about politics.

He’s wondering: *“What if I could design a machine… that thinks?”*

This wasn’t science fiction. This was the beginning of a revolution.

The Birth of the Turing Machine

Alan Turing doesn’t build a robot. He builds an **idea**. A hypothetical machine — which we now call the **Turing Machine** — that could simulate any algorithm you can think of, given enough time and tape.

Think of it like this: if you can explain the rules clearly enough, this “machine” can follow them blindly and produce the answer.

It was like inventing the brain… without the neurons.

Fast Forward to 1950: “Can Machines Think?”

In 1950, Turing writes a paper with the boldest question of his time:
“Can machines think?”

To answer that, he proposes a test — later called the **Turing Test** — where if a machine can fool a human into thinking it’s also human, we must admit it has some form of intelligence.

This moment is now seen as the starting pistol for Artificial Intelligence.

Spoiler alert: Machines didn't pass that test then.
But they're getting really close now.

Enter: The First Learning Machine (1957)

Fast forward a few more years.

A psychologist named **Frank Rosenblatt** creates the **Perceptron** — the first algorithm that could *learn* from data.

It was simple, crude, and limited.
But it was a machine that could improve itself.
That was enough to make the scientific world think.

Headlines declared:

> Perceptron could walk, talk, and learn like a baby!

Okay... it was oversold. It flopped.
But it planted a seed. And seeds grow.

1980s: The Quiet Comeback

The world moves on. AI becomes a punchline in computer science. Funding dries up.

But in dusty labs and stubborn minds, the idea of **machine learning** refuses to die.

In 1986, researchers bring back neural networks — inspired by how our brain works — and this time, the models can finally *do something useful*. Like recognizing handwriting. Or predicting data trends.

Still early. Still basic.

But the machines were learning again.

1997: Game Over, Humans

In 1997, IBM's **Deep Blue** defeats world chess champion Garry Kasparov.

Let that sink in:
A computer beat the best human in the world… in chess.
And not by brute force alone — but by learning and optimizing strategies.

Suddenly, "Can machines think?" becomes less of a joke and more of a warning.

We'd entered the Machine Learning era.

But How Do Machines Learn?

That's the million-dollar question, right?

Not all machines learn the same way.

Some learn like students, from labeled examples — we call this **Supervised Learning**.

Some are explorers, figuring things out without answers — **Unsupervised Learning**.

And some are more like gamers, trial-and-error addicts, learning from wins and losses — **Reinforcement Learning**.

That's what we'll explore next.

But remember: It all started with a question in a quiet room in 1936.

A question that hasn't been fully answered yet.

> Can machines think?

Supervised Learning — Explained like you’re 10

Let’s go back to school. Think of your first-grade classroom.

You’re sitting at a desk. Your teacher holds up a flashcard. On it is a photo of an animal.

She says: “This is a cat.”

Then comes another card.

“This is a dog.”

Card after card, she shows you animals and tells you their names.
You see the picture. You get the answer. Over time, your brain starts to figure out the patterns:
Cats have pointy ears. Dogs have longer snouts.
You don’t even realize it — but you’re learning.

That… is supervised learning.

Why “Supervised”?

Because there’s a teacher.
A supervisor.
Someone who tells the machine:
“This is the input, and this is the correct output.”

The machine, like a student, looks at examples and tries to learn the relationship between the two.

It’s not magic. It’s just matching patterns.

A Real-World Example: Spam Emails

You get 100 emails. Some are spam. Some are not.
You (or someone) mark which ones are spam. That's the **label**.

Now you feed these labeled emails to a machine learning model.

It starts to notice:
- Spam emails often say "Congratulations!"
- Or have lots of exclamation marks!!!
- Or ask for money.

Next time, when a new email comes in, the model guesses if it's spam — based on what it learned.

That's supervised learning in action.

Some Famous Supervised Algorithms

You don't need to memorize these. But here's what people often use:

- **Linear Regression** – Predict numbers (like house prices).
- **Logistic Regression** – Predict categories (yes/no, spam/not spam).
- **Decision Trees** – Ask questions like 20 Questions game.
- **Support Vector Machines** – Fancy name, draws a line between things.

They all do the same basic job:
Learn from labeled data → Predict the label of new data.

Why Is It So Popular?

Because labeled data is everywhere.

- A doctor labels an X-ray: "This shows cancer."
- A user labels a product review: "This is positive."
- A bank labels a loan application: "This one defaulted."

Machines love that kind of help.

The more labeled examples they see, the better they get.

When It Fails

But here’s the problem: What if you don’t have labels?

What if no one told you what was a cat or a dog?

Then the machine is on its own. That’s a whole different style of learning — and we’ll cover that in the next lesson.

For now, just remember:

Supervised learning is like a student with a stack of flashcards and a teacher by their side.

Watch. Learn. Practice. Repeat.

That’s how machines — and humans — get smarter.

Unsupervised Learning — Explained like you’re 10

Imagine this.

You’re dropped in a brand new city.
No guide. No map. No translator.

You walk the streets, look at people, try the food, hear the language — and start figuring things out on your own.

You notice some people are always in suits, rushing into glass buildings.
Others wear aprons and hang around food stalls.
Some are in uniforms, directing traffic.

You don’t *know* their job titles.
No one told you.
But your brain starts grouping them: office workers, chefs, police officers.

That… is unsupervised learning.

So, What’s Unsupervised Learning?

It’s learning… without answers.

There’s no teacher. No labels. No one telling the machine what’s right or wrong.

You just give the machine a pile of data and say,
“Go figure it out.”

Find patterns. Find structure. Group things. Compress the chaos.

Why Is It Called “Unsupervised”?

Simple — because there’s no supervision.

No labels. No answers.
Just the data.

The machine is flying blind, using math and curiosity to group things that look similar.

Like organizing a bunch of random photos without knowing who or what is in them.

A Real-World Example: Customer Segmentation

Let’s say you run an online store.

You have thousands of customers. You don’t know much about them — but you have their behavior:

- What they buy
- How often they visit
- How much they spend

Using unsupervised learning, your system might group them like this:

- Bargain hunters
- Big spenders
- Night-time browsers

No one gave the machine these labels.
It *discovered* them.

Now you can offer deals to one group, loyalty perks to another. Smart, right?

The Usual Algorithms (No Memorizing Needed)

Here are some common ones:

- **K-means Clustering** – Groups similar things into "clusters"
- **Hierarchical Clustering** – Builds a family tree of data
- **PCA (Principal Component Analysis)** – Compresses big messy data into simpler form

These are just tools. The idea is the same:
"I don't know what this is. But let's find what's similar."

Sherlock Holmes Was Unsupervised

Think about it.

Sherlock shows up at a crime scene. No one tells him what happened.
He looks at clues, gathers data, and starts seeing patterns others miss.

That's classic unsupervised thinking.

Machines are doing the same thing — but with numbers, not footprints.

When Do We Use It?

- When we don't have labeled data
- When we want to **discover** hidden patterns
- When we want to **explore** before making decisions

It's not always accurate, but it's insanely useful.

Reinforcement Learning — Explained like you’re 10

Picture this.

A small robot is standing in front of a maze.
The goal? Reach the exit.

The robot doesn’t have a map. No one tells it where to go.
So it takes a step forward… hits a wall.
Ouch.

It takes a step to the right… nothing happens.
Takes another step… and the floor lights up.
A reward.

The robot starts smiling (well, in code) — it’s learning.

That’s reinforcement learning.

So, What Is Reinforcement Learning?

It’s learning by doing.
By trying.
By failing.
By getting rewards — or punishments — and figuring out what works.

It’s the kind of learning we all do naturally.

When a kid touches a hot pan — they learn not to do it again.
When a dog gets a treat for sitting — it starts sitting a lot more.

Machines can do the same.

Why “Reinforcement”?

Because every action gets feedback.
Positive or negative.
That feedback is called **reward**.

The machine’s job is to figure out:
“What sequence of actions gets me the most reward over time?”

It doesn’t memorize answers.
It learns a *strategy*.

A Real-World Analogy: Playing a Video Game

Imagine you’re playing Super Mario.
The first time, you fall into holes.
You bump into enemies.
But slowly, you learn when to jump, where to run, how to win.

You’re not memorizing.
You’re adjusting based on results.
That’s reinforcement learning.

Now imagine a bot doing the same thing — but thousands of times faster, and never getting bored.

That’s how machines are now beating humans in games we thought were unbeatable.

Key Terms (Simple Version)

Let’s keep this light:

- **Agent**: The learner (our robot, bot, or AI).
- **Environment**: The world it lives in (the maze, the game, the road).
- **Action**: What the agent does.

- **Reward**: The score it gets — good or bad — after taking an action.
- **Policy**: The strategy it develops over time.

The goal?
Maximize total reward.

The AlphaGo Story (2015)

In 2015, Google DeepMind built **AlphaGo** — a reinforcement learning system that learned how to play the ancient game of Go.

No one thought a computer could beat a Go champion. The game was too complex.

But AlphaGo didn't just study human moves — it *played millions of games against itself*, learning from every win and loss.

And in 2016, it crushed the world champion.

That's reinforcement learning at its best.

Where Is It Used?

- **Self-driving cars**: Learn how to drive safely by trying things in simulation.
- **Robotics**: Teach a robot to walk, pick things up, or open doors.
- **Finance**: Learn when to buy or sell stock based on reward signals.
- **Game AIs**: From chess to Dota, RL bots are everywhere.

Which one to use when?

Alright, we've met all three machine learning styles.

Now imagine this is a reality show.

You're the host.
A problem shows up.
Three contestants - Supervised, Unsupervised, and Reinforcement - raise their hands.

But only one gets to solve it.

So let's figure out:
Who learns what? And when?

Round 1: Recognizing the Animals

Problem: You have 10,000 images of animals. Each one is labeled as "cat" or "dog".

Who's raising their hand?

Supervised Learning stands up.

"Give me the answers, and I'll learn the pattern."

That's what it's built for - labeled data and clear goals.

Round 2: Recognizing the Crowd

Problem: You have customer data, but no labels. You don't know who buys what or why. You just have behavior logs.

Who's up now?

Unsupervised Learning says, “I got this.” “I’ll find patterns in the data without needing labels.”

It starts grouping people based on what they do - even if you didn’t ask it to.

It’s the curious explorer of the group.

Round 3: The Game Player

Problem: You need to train a robot to walk. Or teach a car to drive. Or beat someone in chess.

Who steps up?

Reinforcement Learning smirks and says:
“Let me try. I’ll figure it out after a few thousand mistakes.”

It thrives where actions have consequences - and learning comes from experience.

Cheat Sheet

Learning Type	Learns From	Example Task
Supervised	Labeled data	Email spam filter
Unsupervised	Unlabeled data	Customer segmentation
Reinforcement	Trial and error	Self-driving car, video games

Quick Quiz - Who Would You Call?

1. Predict tomorrow’s weather based on past labeled data?
   → **Supervised**

2. Discover hidden groups in social media users?
   → **Unsupervised**

3. Train a drone to land smoothly on a platform?
   → **Reinforcement**

4. Classify X-rays as healthy or not?
   → **Supervised**

Conclusion

All three learning styles are useful.
None is better than the others.
It depends on the problem.

Think of them like characters in a team:

- Supervised is the **student with notes**
- Unsupervised is the **detective with no clues**
- Reinforcement is the **gamer who never gives up**

Together, they form the core of machine learning.

History of Machine Learning

Imagine it’s 1936. The world is on the edge of chaos. But in a quiet university in England, a 24-year-old genius named Alan Turing is sitting in his room… thinking.

Not about war. Not about politics.

He’s wondering: *“What if I could design a machine… that thinks?”*

This wasn’t science fiction. This was the beginning of a revolution.

The Birth of the Turing Machine

Alan Turing doesn’t build a robot. He builds an **idea**. A hypothetical machine — which we now call the **Turing Machine** — that could simulate any algorithm you can think of, given enough time and tape.

Think of it like this: if you can explain the rules clearly enough, this “machine” can follow them blindly and produce the answer.

It was like inventing the brain… without the neurons.

Fast Forward to 1950: “Can Machines Think?”

In 1950, Turing writes a paper with the boldest question of his time:
“Can machines think?”

To answer that, he proposes a test — later called the **Turing Test** — where if a machine can fool a human into thinking it’s also human, we must admit it has some form of intelligence.

This moment is now seen as the starting pistol for Artificial Intelligence.

Spoiler alert: Machines didn't pass that test then.
But they're getting really close now.

Enter: The First Learning Machine (1957)

Fast forward a few more years.

A psychologist named **Frank Rosenblatt** creates the **Perceptron** — the first algorithm that could *learn* from data.

It was simple, crude, and limited.
But it was a machine that could improve itself.
That was enough to make the scientific world think.

Headlines declared:

> Perceptron could walk, talk, and learn like a baby!

Okay... it was oversold. It flopped.
But it planted a seed. And seeds grow.

1980s: The Quiet Comeback

The world moves on. AI becomes a punchline in computer science. Funding dries up.

But in dusty labs and stubborn minds, the idea of **machine learning** refuses to die.

In 1986, researchers bring back neural networks — inspired by how our brain works — and this time, the models can finally *do something useful*. Like recognizing handwriting. Or predicting data trends.

Still early. Still basic.

But the machines were learning again.

1997: Game Over, Humans

In 1997, IBM's **Deep Blue** defeats world chess champion Garry Kasparov.

Let that sink in:
A computer beat the best human in the world… in chess.
And not by brute force alone — but by learning and optimizing strategies.

Suddenly, "Can machines think?" becomes less of a joke and more of a warning.

We'd entered the Machine Learning era.

But How Do Machines Learn?

That's the million-dollar question, right?

Not all machines learn the same way.

Some learn like students, from labeled examples — we call this **Supervised Learning**.

Some are explorers, figuring things out without answers — **Unsupervised Learning**.

And some are more like gamers, trial-and-error addicts, learning from wins and losses — **Reinforcement Learning**.

That's what we'll explore next.

But remember: It all started with a question in a quiet room in 1936.

A question that hasn't been fully answered yet.

> Can machines think?

Supervised Learning — Explained like you’re 10

Let’s go back to school. Think of your first-grade classroom.

You’re sitting at a desk. Your teacher holds up a flashcard. On it is a photo of an animal.

She says: “This is a cat.”

Then comes another card.

“This is a dog.”

Card after card, she shows you animals and tells you their names.
You see the picture. You get the answer. Over time, your brain starts to figure out the patterns:
Cats have pointy ears. Dogs have longer snouts.
You don’t even realize it — but you’re learning.

That… is supervised learning.

Why “Supervised”?

Because there’s a teacher.
A supervisor.
Someone who tells the machine:
“This is the input, and this is the correct output.”

The machine, like a student, looks at examples and tries to learn the relationship between the two.

It’s not magic. It’s just matching patterns.

A Real-World Example: Spam Emails

You get 100 emails. Some are spam. Some are not.
You (or someone) mark which ones are spam. That's the **label**.

Now you feed these labeled emails to a machine learning model.

It starts to notice:
- Spam emails often say "Congratulations!"
- Or have lots of exclamation marks!!!
- Or ask for money.

Next time, when a new email comes in, the model guesses if it's spam — based on what it learned.

That's supervised learning in action.

Some Famous Supervised Algorithms

You don't need to memorize these. But here's what people often use:

- **Linear Regression** – Predict numbers (like house prices).
- **Logistic Regression** – Predict categories (yes/no, spam/not spam).
- **Decision Trees** – Ask questions like 20 Questions game.
- **Support Vector Machines** – Fancy name, draws a line between things.

They all do the same basic job:
Learn from labeled data → Predict the label of new data.

Why Is It So Popular?

Because labeled data is everywhere.

- A doctor labels an X-ray: "This shows cancer."
- A user labels a product review: "This is positive."
- A bank labels a loan application: "This one defaulted."

Machines love that kind of help.

The more labeled examples they see, the better they get.

When It Fails

But here’s the problem: What if you don’t have labels?

What if no one told you what was a cat or a dog?

Then the machine is on its own. That’s a whole different style of learning — and we’ll cover that in the next lesson.

For now, just remember:

Supervised learning is like a student with a stack of flashcards and a teacher by their side.

Watch. Learn. Practice. Repeat.

That’s how machines — and humans — get smarter.

Unsupervised Learning — Explained like you’re 10

Imagine this.

You’re dropped in a brand new city.
No guide. No map. No translator.

You walk the streets, look at people, try the food, hear the language — and start figuring things out on your own.

You notice some people are always in suits, rushing into glass buildings.
Others wear aprons and hang around food stalls.
Some are in uniforms, directing traffic.

You don’t *know* their job titles.
No one told you.
But your brain starts grouping them: office workers, chefs, police officers.

That… is unsupervised learning.

So, What’s Unsupervised Learning?

It’s learning… without answers.

There’s no teacher. No labels. No one telling the machine what’s right or wrong.

You just give the machine a pile of data and say,
“Go figure it out.”

Find patterns. Find structure. Group things. Compress the chaos.

Why Is It Called “Unsupervised”?

Simple — because there’s no supervision.

No labels. No answers.
Just the data.

The machine is flying blind, using math and curiosity to group things that look similar.

Like organizing a bunch of random photos without knowing who or what is in them.

A Real-World Example: Customer Segmentation

Let’s say you run an online store.

You have thousands of customers. You don’t know much about them — but you have their behavior:

- What they buy
- How often they visit
- How much they spend

Using unsupervised learning, your system might group them like this:

- Bargain hunters
- Big spenders
- Night-time browsers

No one gave the machine these labels.
It *discovered* them.

Now you can offer deals to one group, loyalty perks to another. Smart, right?

The Usual Algorithms (No Memorizing Needed)

Here are some common ones:

- **K-means Clustering** – Groups similar things into “clusters”
- **Hierarchical Clustering** – Builds a family tree of data
- **PCA (Principal Component Analysis)** – Compresses big messy data into simpler form

These are just tools. The idea is the same:
“I don’t know what this is. But let’s find what’s similar.”

Sherlock Holmes Was Unsupervised

Think about it.

Sherlock shows up at a crime scene. No one tells him what happened.
He looks at clues, gathers data, and starts seeing patterns others miss.

That’s classic unsupervised thinking.

Machines are doing the same thing — but with numbers, not footprints.

When Do We Use It?

- When we don’t have labeled data
- When we want to **discover** hidden patterns
- When we want to **explore** before making decisions

It’s not always accurate, but it’s insanely useful.

Reinforcement Learning — Explained like you're 10

Picture this.

A small robot is standing in front of a maze.
The goal? Reach the exit.

The robot doesn't have a map. No one tells it where to go.
So it takes a step forward… hits a wall.
Ouch.

It takes a step to the right… nothing happens.
Takes another step… and the floor lights up.
A reward.

The robot starts smiling (well, in code) — it's learning.

That's reinforcement learning.

So, What Is Reinforcement Learning?

It's learning by doing.
By trying.
By failing.
By getting rewards — or punishments — and figuring out what works.

It's the kind of learning we all do naturally.

When a kid touches a hot pan — they learn not to do it again.
When a dog gets a treat for sitting — it starts sitting a lot more.

Machines can do the same.

Why “Reinforcement”?

Because every action gets feedback.
Positive or negative.
That feedback is called **reward**.

The machine’s job is to figure out:
“What sequence of actions gets me the most reward over time?”

It doesn’t memorize answers.
It learns a *strategy*.

A Real-World Analogy: Playing a Video Game

Imagine you’re playing Super Mario.
The first time, you fall into holes.
You bump into enemies.
But slowly, you learn when to jump, where to run, how to win.

You’re not memorizing.
You’re adjusting based on results.
That’s reinforcement learning.

Now imagine a bot doing the same thing — but thousands of times faster, and never getting bored.

That’s how machines are now beating humans in games we thought were unbeatable.

Key Terms (Simple Version)

Let’s keep this light:

- **Agent**: The learner (our robot, bot, or AI).
- **Environment**: The world it lives in (the maze, the game, the road).
- **Action**: What the agent does.

- **Reward**: The score it gets — good or bad — after taking an action.
- **Policy**: The strategy it develops over time.

The goal?
Maximize total reward.

The AlphaGo Story (2015)

In 2015, Google DeepMind built **AlphaGo** — a reinforcement learning system that learned how to play the ancient game of Go.

No one thought a computer could beat a Go champion. The game was too complex.

But AlphaGo didn't just study human moves — it *played millions of games against itself*, learning from every win and loss.

And in 2016, it crushed the world champion.

That's reinforcement learning at its best.

Where Is It Used?

- **Self-driving cars**: Learn how to drive safely by trying things in simulation.
- **Robotics**: Teach a robot to walk, pick things up, or open doors.
- **Finance**: Learn when to buy or sell stock based on reward signals.
- **Game AIs**: From chess to Dota, RL bots are everywhere.

Which one to use when?

Alright, we've met all three machine learning styles.

Now imagine this is a reality show.

You're the host.
A problem shows up.
Three contestants - Supervised, Unsupervised, and Reinforcement - raise their hands.

But only one gets to solve it.

So let's figure out:
Who learns what? And when?

Round 1: Recognizing the Animals

Problem: You have 10,000 images of animals. Each one is labeled as "cat" or "dog".

Who's raising their hand?

Supervised Learning stands up.

"Give me the answers, and I'll learn the pattern."

That's what it's built for - labeled data and clear goals.

Round 2: Recognizing the Crowd

Problem: You have customer data, but no labels. You don't know who buys what or why. You just have behavior logs.

Who's up now?

Unsupervised Learning says, “I got this.” “I’ll find patterns in the data without needing labels.”

It starts grouping people based on what they do - even if you didn’t ask it to.

It’s the curious explorer of the group.

Round 3: The Game Player

Problem: You need to train a robot to walk. Or teach a car to drive. Or beat someone in chess.

Who steps up?

Reinforcement Learning smirks and says:
“Let me try. I’ll figure it out after a few thousand mistakes.”

It thrives where actions have consequences - and learning comes from experience.

Cheat Sheet

Learning Type	Learns From	Example Task
Supervised	Labeled data	Email spam filter
Unsupervised	Unlabeled data	Customer segmentation
Reinforcement	Trial and error	Self-driving car, video games

Quick Quiz - Who Would You Call?

1. Predict tomorrow’s weather based on past labeled data?
   → **Supervised**

2. Discover hidden groups in social media users?
   → **Unsupervised**

3. Train a drone to land smoothly on a platform?
   → **Reinforcement**

4. Classify X-rays as healthy or not?
   → **Supervised**

Conclusion

All three learning styles are useful.
None is better than the others.
It depends on the problem.

Think of them like characters in a team:

- Supervised is the **student with notes**
- Unsupervised is the **detective with no clues**
- Reinforcement is the **gamer who never gives up**

Together, they form the core of machine learning.

Machine Learning Problem-Solving Steps

In order to solve machine learning problems, we follow a structured approach that helps ensure accuracy, clarity, and effectiveness. Here are the main steps involved:

1. Look at the Big Picture

Understand the overall problem you're solving. Define your objective clearly — what does success look like?

2. Get the Data

Collect relevant and quality data from reliable sources. Without data, there's no machine learning.

3. Explore and Visualize the Data

Analyze and visualize data to uncover patterns, trends, and anomalies. This step helps you understand what you're working with.

4. Prepare the Data

Clean, transform, and format the data. Handle missing values, normalize features, and split the data into training and testing sets.

5. Select a Model and Train It

Choose a suitable machine learning algorithm and train it using your data. This is where your model learns from patterns.

6. Fine-Tune Your Model

Optimize hyperparameters, try different techniques, and improve performance through iteration.

7. Present Your Solution

Explain your model’s results using visuals, metrics, and clear language so stakeholders can understand and make decisions.

8. Launch, Monitor, and Maintain

Deploy the model in the real world, monitor its performance, and update it regularly as new data arrives.

Datasets for Machine Learning

Machine Learning requires quality datasets for training and testing models. Some popular sources include:

- **OpenML.org** – A collaborative platform offering a wide range of datasets with metadata and tools for benchmarking.
- **UCI Machine Learning Repository** – One of the oldest and most widely used sources for machine learning datasets.
- **Kaggle** – A data science community offering large-scale, real-world datasets and competitions.

Quick Training with Scikit-learn (CSV Data)

This guide walks you through training a model on your own CSV dataset using scikit-learn.

1. Import Libraries

```python
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
```

2. Load Your CSV File

```python
data = pd.read_csv('data.csv')  # Replace with your actual CSV file path
```

3. Separate Features and Label

```python
X = data.iloc[:, :-1]  # All columns except the last as features
y = data.iloc[:, -1]   # Last column as label
```

4. Train the Model

```python
model = RandomForestClassifier()
model.fit(X, y)
```

The model is now trained on your complete dataset.

5. Inference

```
predictions = model.predict(X)  # Predict on the same/new data (for demonstration)
print(predictions)  # Display predictions
```

Evaluating Performance of ML Models

When we build a machine learning model, especially a classification model (which predicts categories like “spam” or “not spam”, “dog” or “cat”), it’s important to measure how well the model is performing.

One of the most basic ways to evaluate a classification model is **accuracy**.

What is Accuracy?

In simple terms, **accuracy** tells us how often the model was **right**.

If you gave your model 100 questions to answer, and it got 90 of them correct, its accuracy would be:

```
Accuracy = Correct Predictions / Total Predictions = 90 / 100 = 90%
```

So, **accuracy** is just the fraction of predictions the model got right.

Evaluating Accuracy

> Accuracy = (Correct Predictions) / (Total Predictions)

This is the simplest and most intuitive way to check if the model is doing a good job.

Gurgaon House Price Prediction Model

Why Are We Building This?

Gurgaon, a rapidly growing city in India, has seen a sharp rise in real estate development over the past decade. With its proximity to Delhi, booming IT hubs, and modern infrastructure, Gurgaon has become a major attraction for both homebuyers and investors. However, the real estate market here is highly dynamic and often difficult to assess without proper data-driven tools.

We’re building a Gurgaon house price prediction model to help:

- Understand how factors like location, size, number of rooms, and amenities affect property prices in Gurgaon.
- Assist buyers in identifying fair prices based on historical trends.
- Help sellers estimate an appropriate asking price.
- Empower real estate agents and platforms to improve recommendations and negotiations.

How Will We Build It?

While we don’t have access to a large, clean dataset of house prices in Gurgaon right now, we will use a well-known and cleaned dataset—the California housing dataset—as a proxy. This will allow us to build, test, and evaluate a working model with real-world variables like:

- Median income of the area
- Proximity to the city center
- Number of rooms
- Latitude and longitude
- Population density

We’ll treat this as a simulation: suppose the California data is Gurgaon data, and suppose we are building this model for a neighborhood where both you and I live or work nearby.

Once the model is developed and understood, we can later adapt the same approach to real Gurgaon data when available, using the same techniques and logic.

Revisiting Steps to solve this problem

Understanding the Problem

Before we build the model, we need to understand what kind of machine learning problem we are solving.

First, we'll check if this is a **supervised** or **unsupervised** learning task. Since we have historical data with house prices (our target), and we want to predict the price of a house based on input features, this is a **supervised learning** problem.

Next, we observe that the model is predicting **one continuous label** — the price of a house — based on several input features. This makes it a **univariate regression problem**.

Measuring Errors (RMSE & MAE)

After training our regression model, we need to evaluate how good its predictions are. Two common metrics used for this are **MAE** and **RMSE**.

1. Mean Absolute Error (MAE)

MAE stands for Mean Absolute Error. It calculates the average of the absolute differences between the predicted and actual values.

Formula: MAE = (1/n) × sum of |actual - predicted|

- It treats all errors equally, no matter their size.
- MAE is based on the **Manhattan norm** (also called L1 norm), which measures distance by summing absolute values.

2. Root Mean Squared Error (RMSE)

RMSE stands for Root Mean Squared Error. It calculates the square root of the average of squared differences between predicted and actual values.

Formula: RMSE = sqrt((1/n) × sum of (actual - predicted)^2)

- RMSE gives more weight to larger errors because it squares them.
- RMSE is based on the **Euclidean norm** (also called L2 norm), which measures straight-line distance.

Summary

- **Use MAE** when all errors should be treated equally.
- **Use RMSE** when larger errors should be penalized more.

Analyzing the Data (EDA)

Exploratory Data Analysis (EDA) is the process of examining a dataset to summarize its main characteristics, often using visual methods or quick commands. The goal is to understand the structure of the data, detect patterns, spot anomalies, and get a feel for what kind of preprocessing or modeling might be needed.

For our project, we'll perform EDA on the California housing dataset (which we are treating as if it represents Gurgaon data). Here are some key commands we'll use:

1. `df.head()`

- Displays the first 5 rows of the dataset.
- Useful for getting a quick overview of what the data looks like — column names, data types, and sample values.

2. `df.info()`

- Gives a summary of the dataset.
- Shows the number of entries, column names, data types, and how many non-null values each column has.
- Helps us identify missing values or incorrect data types.

3. `df.describe()`

- Provides statistical summaries for numeric columns.
- Shows:
    - **Count**: Total number of non-null entries
    - **Mean**: Average value
    - **Std**: Standard deviation
    - **Min**: The smallest value (0th percentile or 1st quartile in some contexts)

- **25%**: The 1st quartile (Q1) — 25% of the data is below this value
- **50%**: The median or 2nd quartile (Q2) — half of the data is below this value
- **75%**: The 3rd quartile (Q3) — 75% of the data is below this value
- **Max**: The largest value (often considered the 4th quartile or 100th percentile)

> **Percentiles** divide the data into 100 equal parts. **Quartiles** divide the data into 4 equal parts (Q1 = 25th percentile, Q2 = 50th, Q3 = 75th). So:
>
> - **Min** is the 0th percentile
> - **Max** is the 100th percentile

This helps us understand how the values are spread out and if there are outliers.

4. `df['column_name'].value_counts()`

- Shows the count of each unique value in a specific column.
- Useful for categorical columns to see how values are distributed.

Creating a Test Set

When building machine learning models, one of the most important steps is **splitting your dataset** into training and test sets. This ensures your model is evaluated on data it has never seen before, which is critical for assessing its ability to generalize.

The Problem of Data Snooping Bias

Data snooping bias occurs when information from the test set leaks into the training process. This can lead to overly optimistic performance metrics and models that don't perform well in real-world scenarios.

To avoid this, the test set must be isolated before any data exploration, feature selection, or model training begins.

Random Sampling: A Basic Approach

A simple method to split the data is to randomly shuffle it and then divide it:

```python
import numpy as np

def shuffle_and_split_data(data, test_ratio):
    np.random.seed(42)  # Set the seed for reproducibility
    shuffled_indices = np.random.permutation(len(data))
    test_set_size = int(len(data) * test_ratio)
    test_indices = shuffled_indices[:test_set_size]
    train_indices = shuffled_indices[test_set_size:]
    return data.iloc[train_indices], data.iloc[test_indices]
```

Setting the random seed (e.g., with `np.random.seed(42)`) ensures consistency across runs — this is crucial for debugging and comparing models fairly.

However, pure random sampling might not always be reliable, especially if the dataset contains important patterns that are not evenly distributed.

Stratified Sampling

To ensure that important characteristics of the population are well represented in both the training and test sets, we use **stratified sampling**.

What is a Strata?

A **strata** is a subgroup of the data defined by a specific attribute. Stratified sampling ensures that each of these subgroups is proportionally represented.

For example, in the California housing dataset, **median income** is a strong predictor of house prices. Instead of randomly sampling, we can create strata based on income levels (e.g., binning median income into categories) and ensure the test set maintains the same distribution of income levels as the full dataset.

Creating Income Categories

```
import pandas as pd
# Load the dataset
data = pd.read_csv("housing.csv")
# Create income categories
data["income_cat"] = pd.cut(data["median_income"],
                             bins=[0, 1.5, 3.0, 4.5, 6.0, np.inf],
                             labels=[1, 2, 3, 4, 5])
```

This code creates a new column `income_cat` that categorizes the `median_income` into five bins. Each bin represents a range of income levels, allowing us to stratify our sampling based on these categories.

We can plot these income categories to visualize the distribution:

```
import matplotlib.pyplot as plt
data["income_cat"].value_counts().sort_index().plot.bar(rot=0, grid=True)
plt.title("Income Categories Distribution")
plt.xlabel("Income Category")
plt.ylabel("Number of Instances")
plt.show()
```

Stratified Shuffle Split in Scikit-Learn

Scikit-learn provides a built-in way to perform stratified sampling using `StratifiedShuffleSplit`.

Here’s how you can use it:

```
from sklearn.model_selection import StratifiedShuffleSplit

# Assume income_cat is a column in the dataset created from median_income
split = StratifiedShuffleSplit(n_splits=1, test_size=0.2, random_state=42)

for train_index, test_index in split.split(data, data["income_cat"]):
    strat_train_set = data.loc[train_index]
    strat_test_set = data.loc[test_index]
```

This ensures that the income distribution in both sets is similar to that of the full dataset, reducing sampling bias and making your model evaluation more reliable.

Data Visualization

Before handling missing values or training models, it's important to **visualize the data** to uncover patterns, relationships, and potential issues.

Geographical Scatter Plot

Visualize the geographical distribution of the data:

```
df.plot(kind="scatter", x="longitude", y="latitude", grid=True, alpha=0.2)
plt.show()
```

- `alpha=0.2` makes overlapping points more visible.
- This helps reveal data clusters and high-density areas like coastal regions.

Correlation Matrix

To understand relationships between numerical features, compute the **correlation matrix**:

```
corr_matrix = df.corr()
```

Check how strongly each attribute correlates with the target:

```
corr_matrix["median_house_value"].sort_values(ascending=False)
```

This helps identify useful predictors. For example, `median_income` usually shows a strong positive correlation with house prices.

Scatter Matrix

Plot selected features to see pairwise relationships:

```
from pandas.plotting import scatter_matrix

attributes = ["median_house_value", "median_income", "total_rooms", "housing_median
_age"]
scatter_matrix(df[attributes], figsize=(12, 8))
plt.show()
```

This gives an overview of which features are linearly related and may be good predictors.

Focused Income vs Price Plot

Plot `median_income` vs `median_house_value` directly:

```
df.plot(kind="scatter", x="median_income", y="median_house_value", alpha=0.1,
grid=True)
```

Further Preprocessing & Handling Missing Data

Before feeding your data into a machine learning algorithm, you need to clean and prepare it.

Prepare Data for Training

It's best to write transformation functions instead of applying them manually. This ensures:

- Reproducibility on any dataset
- Reusability across projects
- Compatibility with live systems
- Easier experimentation

Start by creating a clean copy and separating the predictors and labels:

```
housing = strat_train_set.drop("median_house_value", axis=1)
housing_labels = strat_train_set["median_house_value"].copy()
```

Handling Missing Data

Some features, like `total_bedrooms`, contain missing values. You can:

1. Drop rows with missing values
2. Drop the entire column
3. **Impute missing values** (recommended)

We'll use **option 3** using `SimpleImputer` from Scikit-Learn, which allows consistent handling across all datasets (train, test, new data):

```python
from sklearn.impute import SimpleImputer
imputer = SimpleImputer(strategy="median")

housing_num = housing.select_dtypes(include=[np.number])
imputer.fit(housing_num)
```

This computes the median for each numerical column and stores it in `imputer.statistics_`:

```
>>> imputer.statistics_
array([-118.51 ,  34.26 ,  29. , 2125. , 434. , 1167. , 408. , 3.5385])
```

Now apply the learned medians to transform the data:

```python
X = imputer.transform(housing_num)
```

Other available strategies:

- `"mean"` – replaces with mean value
- `"most_frequent"` – for the most common value (can handle categorical)
- `"constant"` – fill with a fixed value using `fill_value=...`

Scikit-Learn Design Principles

Scikit-Learn has a simple and consistent API that makes it easy to use and understand. Below are the key design principles behind it:

---

1. Consistency

All objects follow a standard interface, which makes learning and using different tools in Scikit-Learn easier.

---

2. Estimators

Any object that learns from data is called an **estimator**.

- Use the `.fit()` method to train an estimator.
- In supervised learning, pass both `X` (features) and `y` (labels) to `.fit(X, y)`.
- Hyperparameters (like `strategy='mean'` in `SimpleImputer`) are set when creating the object.

Example:

```
imputer = SimpleImputer(strategy="median")
imputer.fit(data)
```

---

3. Transformers

Some estimators can also transform data. These are called **transformers**.

- Use `.transform()` to apply the transformation after fitting.
- Use `.fit_transform()` to do both in one step.

Example:

```
X_transformed = imputer.fit_transform(data)
```

---

4. Predictors

Models that can make predictions are **predictors**.

- Use `.predict()` to make predictions on new data.
- Use `.score()` to evaluate performance (e.g., accuracy or R^2).

Example:

```
model = LinearRegression()
model.fit(X_train, y_train)
predictions = model.predict(X_test)
score = model.score(X_test, y_test)
```

---

5. Inspection

- Hyperparameters can be accessed directly: `model.param_name`
- Learned parameters are stored with an underscore: `model.coef_`, `imputer.statistics_`

---

6. No Extra Classes

- Inputs and outputs are basic structures like NumPy arrays or Pandas DataFrames.
- No need to learn custom data types.

---

7. Composition

You can combine steps into a **Pipeline**, chaining transformers and a final predictor.

Example:

```
pipeline = Pipeline([
    ("imputer", SimpleImputer(strategy="median")),
    ("model", LinearRegression())
])
pipeline.fit(X, y)
```

---

8. Sensible Defaults

Most tools in Scikit-Learn work well with default settings, so you can get started quickly.

Note on DataFrames

Even if you input a Pandas DataFrame, the output of transformers like `transform()` will be a NumPy array. You can convert it back like this:

```
X = imputer.transform(housing_num)
housing_tr = pd.DataFrame(X, columns=housing_num.columns, index=housing_num.index)
```

Handling Categorical and Text Attributes in Scikit-Learn

Most machine learning algorithms work best with numerical data. But real-world datasets often contain **categorical** or **text attributes**. Let's understand how to handle these in Scikit-Learn using the `ocean_proximity` column from the California housing dataset as an example.

---

1. Categorical Attributes

Text columns like `"ocean_proximity"` are not free-form text but limited to a fixed set of values (e.g., `"NEAR BAY"`, `"INLAND"`). These are known as **categorical attributes**.

Example:

```
housing_cat = housing[["ocean_proximity"]]
housing_cat.head()
```

---

2. Ordinal Encoding

Scikit-Learn's `OrdinalEncoder` can convert categories to numbers:

```
from sklearn.preprocessing import OrdinalEncoder

ordinal_encoder = OrdinalEncoder()
housing_cat_encoded = ordinal_encoder.fit_transform(housing_cat)
```

This will output a 2D NumPy array with numerical category codes.

To see the mapping:

```
ordinal_encoder.categories_
# Output: array(['<1H OCEAN', 'INLAND', 'ISLAND', 'NEAR BAY', 'NEAR OCEAN'])
```

⚠ **Caution**: Ordinal encoding implies an order between categories, which may not be true here. For example, it treats `INLAND (1)` as closer to `<1H OCEAN (0)` than `NEAR OCEAN (4)`, which might not make sense.

---

3. One-Hot Encoding

For unordered categories, **one-hot encoding** is a better choice. It creates one binary column per category.

```
from sklearn.preprocessing import OneHotEncoder

cat_encoder = OneHotEncoder()
housing_cat_1hot = cat_encoder.fit_transform(housing_cat)
```

This gives a **sparse matrix** (efficient storage for mostly zeros).

To convert it to a regular NumPy array:

```
housing_cat_1hot.toarray()
```

Or directly get a dense array:

```
cat_encoder = OneHotEncoder(sparse=False)
housing_cat_1hot = cat_encoder.fit_transform(housing_cat)
```

To check category order:

```
cat_encoder.categories_
```

4. Summary

Method	Use When	Output Type
`OrdinalEncoder`	Categories have an order	2D NumPy array
`OneHotEncoder`	Categories are unordered	Sparse or dense

Using the right encoding ensures your model learns correctly from categorical features.

Feature Scaling and Transformation

Feature scaling is a crucial preprocessing step. Most machine learning algorithms perform poorly when input features have vastly different scales.

In the California housing dataset, for example:

- `total_rooms` ranges from 6 to over 39,000
- `median_income` ranges from 0 to 15

If you don't scale these features, models will give more importance to `total_rooms` simply because it has larger values.

Why Scaling Is Needed

- Many models (like Linear Regression, KNN, SVMs, Gradient Descent-based algorithms) **assume features are on a similar scale**.
- Without scaling, features with larger ranges can **dominate model behavior**.
- Scaling makes training **more stable and faster**.

Min-Max Scaling (Normalization)

This method rescales the data to a specific range, usually `[0, 1]` or `[-1, 1]`.

Formula:

```
scaled_value = (x - min) / (max - min)
```

Use Scikit-Learn's `MinMaxScaler`:

```
from sklearn.preprocessing import MinMaxScaler
```

```
min_max_scaler = MinMaxScaler(feature_range=(-1, 1))
housing_num_min_max_scaled = min_max_scaler.fit_transform(housing_num)
```

- Use `feature_range=(-1, 1)` for models like neural networks.
- Sensitive to outliers — extreme values can distort the scale.

Standardization (Z-score Scaling)

This method centers the data around 0 and scales it based on standard deviation.

Formula:

```
standardized_value = (x - mean) / std
```

Use Scikit-Learn's `StandardScaler`:

```
from sklearn.preprocessing import StandardScaler

std_scaler = StandardScaler()
housing_num_std_scaled = std_scaler.fit_transform(housing_num)
```

- Resulting features have **zero mean and unit variance**
- **Robust to outliers** compared to min-max scaling
- Recommended for most ML algorithms, especially when using gradient descent

Transformation Pipelines

As datasets grow more complex, data preprocessing often involves multiple steps such as imputing missing values, scaling features, encoding categorical variables, etc. These steps must be applied **in the correct order** and consistently across training, validation, test, and future production data.

To streamline this process, **Scikit-Learn** provides the `Pipeline` class — a powerful utility for chaining data transformations.

Building a Numerical Pipeline

A typical pipeline for numerical attributes might include:

1. **Imputation** of missing values (e.g., with median).
2. **Feature scaling** (e.g., with standardization).

```
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler

num_pipeline = Pipeline([
    ("impute", SimpleImputer(strategy="median")),
    ("standardize", StandardScaler()),
])
```

How It Works

- The pipeline takes a list of steps as `(name, transformer)` pairs.
- Names must be unique and should not contain double underscores `__`.
- All intermediate steps must be transformers (i.e., must implement `fit_transform()`).

- The final step can be either a transformer or a predictor.

Using `make_pipeline`

If you don't want to name the steps manually, you can use `make_pipeline()` :

```
from sklearn.pipeline import make_pipeline

num_pipeline = make_pipeline(SimpleImputer(strategy="median"), StandardScaler())
```

- This automatically names the steps using the class names in lowercase.
- If the same class appears multiple times, a number is appended (e.g., `standardscaler-1`).

Applying the Pipeline

Call `fit_transform()` to apply all transformations in sequence:

```
housing_num_prepared = num_pipeline.fit_transform(housing_num)
print(housing_num_prepared[:2].round(2))
```

Example output:

```
array([[-1.42,  1.01,  1.86,  0.31,  1.37,  0.14,  1.39, -0.94],
       [ 0.60, -0.70,  0.91, -0.31, -0.44, -0.69, -0.37,  1.17]])
```

- Each row corresponds to a transformed sample.
- Each column corresponds to a scaled feature.

Retrieving Feature Names

To turn the result back into a DataFrame with feature names:

```
df_housing_num_prepared = pd.DataFrame(
    housing_num_prepared,
    columns=num_pipeline.get_feature_names_out(),
    index=housing_num.index
)
```

Pipeline as a Transformer or Predictor

- If the **last step is a transformer**, the pipeline behaves like a transformer (`fit_transform()`, `transform()`).
- If the **last step is a predictor** (e.g., a model), the pipeline behaves like an estimator (`fit()`, `predict()`).

This flexibility makes `Pipeline` the standard way to handle data preprocessing and modeling in Scikit-Learn projects.

Data Preprocessing - Final Pipeline

In this section, we will **consolidate everything** we've done so far into one final script using Scikit-Learn pipelines. This includes:

1. Creating a stratified test set
2. Handling missing values
3. Encoding categorical variables
4. Scaling numerical features
5. Combining everything using `Pipeline` and `ColumnTransformer`

This will ensure **clean, modular, and reproducible code** — perfect for production and education.

Final Preprocessing Code using Scikit-Learn Pipelines

```python
import pandas as pd
import numpy as np
from sklearn.model_selection import StratifiedShuffleSplit
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
# from sklearn.preprocessing import OrdinalEncoder  # Uncomment if you prefer
ordinal

# 1. Load the data
housing = pd.read_csv("housing.csv")

# 2. Create a stratified test set based on income category
housing["income_cat"] = pd.cut(
    housing["median_income"],
    bins=[0., 1.5, 3.0, 4.5, 6., np.inf],
    labels=[1, 2, 3, 4, 5]
)

split = StratifiedShuffleSplit(n_splits=1, test_size=0.2, random_state=42)
for train_index, test_index in split.split(housing, housing["income_cat"]):
    strat_train_set = housing.loc[train_index].drop("income_cat", axis=1)
    strat_test_set = housing.loc[test_index].drop("income_cat", axis=1)

# Work on a copy of training data
housing = strat_train_set.copy()

# 3. Separate predictors and labels
housing_labels = housing["median_house_value"].copy()
housing = housing.drop("median_house_value", axis=1)

# 4. Separate numerical and categorical columns
num_attribs = housing.drop("ocean_proximity", axis=1).columns.tolist()
cat_attribs = ["ocean_proximity"]

# 5. Pipelines
# Numerical pipeline
num_pipeline = Pipeline([
    ("imputer", SimpleImputer(strategy="median")),
    ("scaler", StandardScaler()),
])

# Categorical pipeline
cat_pipeline = Pipeline([
    # ("ordinal", OrdinalEncoder())  # Use this if you prefer ordinal encoding
    ("onehot", OneHotEncoder(handle_unknown="ignore"))
])

# Full pipeline
full_pipeline = ColumnTransformer([
    ("num", num_pipeline, num_attribs),
    ("cat", cat_pipeline, cat_attribs),
])

# 6. Transform the data
housing_prepared = full_pipeline.fit_transform(housing)
```

```
# housing_prepared is now a NumPy array ready for training
print(housing_prepared.shape)
```

Training and Evaluating ML Models

Now that our data is preprocessed, let's move on to **training machine learning models** and evaluating their performance. We'll start with:

- **Linear Regression**
- **Decision Tree Regressor**
- **Random Forest Regressor**

We'll first test them on the training data and then use **cross-validation** to get a better estimate of their true performance.

---

1. Train and Test Models on the Training Set

```
from sklearn.linear_model import LinearRegression
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error

# Linear Regression
lin_reg = LinearRegression()
lin_reg.fit(housing_prepared, housing_labels)

# Decision Tree
tree_reg = DecisionTreeRegressor(random_state=42)
tree_reg.fit(housing_prepared, housing_labels)

# Random Forest
forest_reg = RandomForestRegressor(random_state=42)
forest_reg.fit(housing_prepared, housing_labels)

# Predict using training data
lin_preds = lin_reg.predict(housing_prepared)
```

```python
tree_preds = tree_reg.predict(housing_prepared)
forest_preds = forest_reg.predict(housing_prepared)

# Calculate RMSE
lin_rmse = mean_squared_error(housing_labels, lin_preds, squared=False)
tree_rmse = mean_squared_error(housing_labels, tree_preds, squared=False)
forest_rmse = mean_squared_error(housing_labels, forest_preds, squared=False)

print("Linear Regression RMSE:", lin_rmse)
print("Decision Tree RMSE:", tree_rmse)
print("Random Forest RMSE:", forest_rmse)
```

A Warning About Training RMSE

Training RMSE **only shows how well the model fits the training data**. It does **not** tell us how well it will perform on unseen data. In fact, the **Decision Tree and Random Forest may overfit**, leading to very low training error but poor generalization.

---

2. Cross-Validation: A Better Evaluation Strategy

Cross-validation helps us evaluate how a model generalizes to new data without needing to touch the test set.

What is Cross-Validation?

Instead of training the model once and evaluating on a holdout set, **k-fold cross-validation** splits the training data into *k* folds (typically 10), trains the model on *k-1* folds, and validates it on the remaining fold. This process repeats *k* times.

We'll use `cross_val_score` from `sklearn.model_selection`.

Cross-Validation on Decision Tree

```python
from sklearn.model_selection import cross_val_score
import pandas as pd

# Evaluate Decision Tree with cross-validation
tree_rmses = -cross_val_score(
    tree_reg,
    housing_prepared,
    housing_labels,
    scoring="neg_root_mean_squared_error",
    cv=10
)

# WARNING: Scikit-Learn’s scoring uses utility functions (higher is better), so
RMSE is returned as negative.
# We use minus (-) to convert it back to positive RMSE.
print("Decision Tree CV RMSEs:", tree_rmses)
print("\nCross-Validation Performance (Decision Tree):")
print(pd.Series(tree_rmses).describe())
```

Model Persistence and Inference with Joblib in a Random Forest Pipeline

Lets now summarize how to **train a Random Forest model on California housing data**, save the model and preprocessing pipeline using `joblib`, and reuse the model later for **inference on new data** (`input.csv`). This approach helps avoid retraining the model every time, improving performance and enabling reproducibility.

Why These Steps?

1. Why Train Once and Save?

- Training models repeatedly is **time-consuming** and **computationally expensive**.
- Saving the model (`model.pkl`) and preprocessing pipeline (`pipeline.pkl`) ensures you can **quickly load and run inference** anytime in the future.

2. Why Use a Preprocessing Pipeline?

- Raw data needs to be cleaned, scaled, and encoded before model training.
- A `Pipeline` automates this transformation and ensures **identical preprocessing** during inference.

3. Why Use Joblib?

- `joblib` efficiently serializes large NumPy arrays (like in sklearn models).
- Faster and more suitable than `pickle` for `scikit-learn` objects.

4. Why the If-Else Logic?

- The program checks if a saved model exists.
  - If **not**, it trains and saves the model.
  - If it **does**, it skips training and **only runs inference**, saving time.

Full Code

```
import os
import pandas as pd
import numpy as np
import joblib

from sklearn.model_selection import StratifiedShuffleSplit
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.ensemble import RandomForestRegressor

MODEL_FILE = "model.pkl"
PIPELINE_FILE = "pipeline.pkl"

def build_pipeline(num_attribs, cat_attribs):
    num_pipeline = Pipeline([
        ("imputer", SimpleImputer(strategy="median")),
        ("scaler", StandardScaler())
    ])
    cat_pipeline = Pipeline([
        ("onehot", OneHotEncoder(handle_unknown="ignore"))
    ])
    full_pipeline = ColumnTransformer([
        ("num", num_pipeline, num_attribs),
        ("cat", cat_pipeline, cat_attribs)
    ])
    return full_pipeline
```

```python
if not os.path.exists(MODEL_FILE):
    # TRAINING PHASE
    housing = pd.read_csv("housing.csv")
    housing['income_cat'] = pd.cut(housing["median_income"],
                                   bins=[0.0, 1.5, 3.0, 4.5, 6.0, np.inf],
                                   labels=[1, 2, 3, 4, 5])
    split = StratifiedShuffleSplit(n_splits=1, test_size=0.2, random_state=42)
    for train_index, _ in split.split(housing, housing['income_cat']):
        housing = housing.loc[train_index].drop("income_cat", axis=1)

    housing_labels = housing["median_house_value"].copy()
    housing_features = housing.drop("median_house_value", axis=1)

    num_attribs = housing_features.drop("ocean_proximity", axis=1).columns.tolist()
    cat_attribs = ["ocean_proximity"]

    pipeline = build_pipeline(num_attribs, cat_attribs)
    housing_prepared = pipeline.fit_transform(housing_features)

    model = RandomForestRegressor(random_state=42)
    model.fit(housing_prepared, housing_labels)

    # Save model and pipeline
    joblib.dump(model, MODEL_FILE)
    joblib.dump(pipeline, PIPELINE_FILE)

    print("Model trained and saved.")

else:
    # INFERENCE PHASE
    model = joblib.load(MODEL_FILE)
    pipeline = joblib.load(PIPELINE_FILE)

    input_data = pd.read_csv("input.csv")
    transformed_input = pipeline.transform(input_data)
    predictions = model.predict(transformed_input)
    input_data["median_house_value"] = predictions
    input_data.to_csv("output.csv", index=False)
    print("Inference complete. Results saved to output.csv")
```

Summary

With this setup, our ML pipeline is:

- **Efficient** – No retraining needed if the model exists.
- **Reproducible** – Same preprocessing logic every time.
- **Production-ready** – Can be deployed or reused across multiple systems.

Conclusion: California Housing Price Prediction Project

In this project, we built a complete machine learning pipeline to **predict California housing prices** using various regression algorithms. We started by:

- **Loading and preprocessing** the dataset (`housing.csv`) with careful treatment of missing values, scaling, and encoding using a custom pipeline.
- **Stratified splitting** was used to maintain income category distribution between train and test sets.
- We **trained and evaluated multiple algorithms** including:
    - Linear Regression
    - Decision Tree Regressor
    - Random Forest Regressor
- Through **cross-validation**, we found that **Random Forest performed the best**, offering the lowest RMSE and most stable results.

Finally, we built a script that:

- Trains the Random Forest model and saves it using `joblib` .
- Uses an **if-else logic** to skip retraining if the model exists.
- Applies the trained model to new data (`input.csv`) to predict `median_house_value` , storing results in `output.csv` .

This pipeline ensures that predictions are **accurate**, **efficient**, and **ready for production deployment**.

Introduction to Neural Networks

Inspiration from Nature

Birds inspired humans to build airplanes. The tiny hooks on burrs sticking to a dog's fur led to the invention of **Velcro**. And just like that, nature has always been humanity's greatest engineer.

So, when it came to making machines that could **think, learn, and solve problems**, where did we look?

To the **human brain**.

That's how **neural networks** were born — machines inspired by **neurons** in our brains, built to recognize patterns, make decisions, and even learn from experience.

What is AI, ML, and DL?

Before we dive into neural networks, let's untangle these buzzwords.

Term	Stands for	Think of it as...
AI	Artificial Intelligence	The big umbrella: making machines "smart"
ML	Machine Learning	A subset of AI: machines that learn from data
DL	Deep Learning	A type of ML: uses neural networks

Let's simplify:

- **AI** is the dream: "Can we make machines intelligent?"
- **ML** is the method: "Let's give machines data and let them learn."
- **DL** is the tool: "Let's use neural networks that learn in layers — like the brain."

So, when we talk about **neural networks**, we're entering the world of **deep learning**, which is a part of **machine learning**, which itself is a part of **AI**.

So, What Are Neural Networks?

Imagine a bunch of simple decision-makers called **neurons**, connected together in layers.

Each neuron:

- Takes some input (like a number)
- Applies a little math (weights + bias)
- Passes the result through a rule (called an activation function)
- Sends the output to the next layer

By connecting many of these neurons, we get a **neural network**.

And what's amazing?

Even though each neuron is simple, when combined, the network becomes powerful — like how a bunch of ants can build a complex colony.

Why Are Neural Networks Useful?

Because they can **learn patterns**, even when we don't fully understand the patterns ourselves.

Examples:

- Recognize cats in photos
- Convert speech to text
- Translate languages
- Predict stock prices
- Generate art
- Power AI like ChatGPT

Structure of a Neural Network

Here’s the basic anatomy of a neural network:

```
Input Layer       →       Hidden Layers        →       Output Layer
  (Data)              (Neurons doing math)             (Prediction)
```

Each **layer** is just a bunch of neurons working together. The more hidden layers, the “**deeper**” the network. Hence: **Deep Learning**.

Wait — Why Not Use Simple Code Instead?

Good question.

Sometimes, a simple formula or rule is enough (like `area = length × width`).

But what about:

- Recognizing handwritten digits?
- Understanding language?
- Diagnosing diseases from X-rays?

There are **no easy formulas** for these. Neural networks **learn the formula by themselves** from lots of examples.

How Do Neural Networks Learn?

Let’s say the network tries to predict `y = x² + x`.

1. It starts with **random guesses** (bad predictions)
2. It checks how wrong it is (loss)
3. It adjusts the internal settings (weights) to be a little better
4. Repeat, repeat, repeat…

Over time, the network **figures out the relationship** between x and y.

This process is called **training** — and it's where the magic happens.

Are They Really Like the Brain?

Kind of — but **very simplified**.

- A biological brain neuron connects to 1000s of others
- It processes chemicals, spikes, timings
- It adapts and rewires itself

A neural network is a **mathematical model** — inspired by the brain, but way simpler. Still, the results are powerful.

Summary

Concept	Meaning
AI	Making machines act smart
ML	Letting machines learn from data
DL	Using multi-layered neural networks to learn complex stuff
Neural Network	A network of artificial neurons that learns from data

Perceptron – The Simplest Neural Network

What is a Perceptron?

The perceptron is the basic building block of a neural network. It’s a simple computational model that takes several inputs, applies weights to them, adds a bias, and produces an output. It’s essentially a decision-making unit.

Real-life Analogy

Imagine you’re trying to decide whether to go outside based on:

- Is it sunny?
- Is it the weekend?
- Are you free today?

You assign importance (weights) to each factor:

- Sunny: 0.6
- Weekend: 0.3
- Free: 0.8

You combine these factors to make a decision: Go or Not Go.

This is what a perceptron does.

Perceptron Formula

A perceptron takes inputs (x1, x2, …, xn), multiplies each by its corresponding weight (w1, w2, …, wn), adds a bias (b), and passes the result through an activation function.

y = f(w1*x1 + w2*x2 + … + wn*xn + b)

Where:

- xi: input features
- wi: weights
- b: bias
- f: activation function (e.g., step function)

Step-by-step Example: Binary Classification

Let’s say we want a perceptron to learn this simple table:

Input (x1, x2)	Output (y)
(0, 0)	0
(0, 1)	0
(1, 0)	0
(1, 1)	1

This is the behavior of a logical AND gate.

We will use:

- Inputs: x1, x2
- Weights: w1, w2
- Bias: b
- Activation Function: Step function

Step function:

```
def step(x):
    return 1 if x >= 0 else 0
```

Code: Simple Perceptron from Scratch

```
def step(x):
    return 1 if x >= 0 else 0

def perceptron(x1, x2, w1, w2, b):
    z = x1 * w1 + x2 * w2 + b
    return step(z)

# Try different weights and bias to match the AND logic
print(perceptron(0, 0, 1, 1, -1.5))  # Expected: 0
print(perceptron(0, 1, 1, 1, -1.5))  # Expected: 0
print(perceptron(1, 0, 1, 1, -1.5))  # Expected: 0
print(perceptron(1, 1, 1, 1, -1.5))  # Expected: 1
```

This matches the AND logic perfectly.

Summary

- A perceptron is the simplest form of a neural network.
- It performs a weighted sum of inputs, adds a bias, and passes the result through an activation function to make a decision.
- It can model simple binary functions like AND, OR, etc.

Common Terms in Deep Learning

Before diving into neural networks, let’s clarify some common terms you’ll encounter:

Perceptron

A **perceptron** is the simplest type of neural network — just one neuron. It takes inputs, multiplies them by weights, adds a bias, and applies an activation function to make a decision (e.g., classify 0 or 1).

Neural Network

A **neural network** is a collection of interconnected layers of perceptrons (neurons). Each layer transforms its inputs using weights, biases, and activation functions. Deep neural networks have multiple hidden layers and can model complex patterns.

Hyperparameters

These are settings we configure **before training** a model. They are not learned from the data. Examples include:

- Learning rate: How much to adjust weights during training
- Number of epochs: How many times the model sees the entire training dataset
- Batch size: How many samples to process before updating weights

- Number of layers or neurons: How many neurons are in each layer of the network

---

Learning Rate (η)

This controls **how much we adjust the weights** after each training step. A learning rate that's too high may overshoot the solution; too low may make training very slow.

Training

This is the process where the model **learns patterns from data** by updating weights based on errors between predicted and actual outputs.

Backpropagation

Backpropagation is the algorithm used to **update weights** in a neural network. It calculates the gradient of the loss function with respect to each weight by applying the chain rule, allowing the model to learn from its mistakes.

Inference

Inference is when the trained model is used to **make predictions** on new, unseen data.

Activation Function

This function adds **non-linearity** to the output of neurons, helping networks model complex patterns.

Common activation functions:

- `ReLU` : Rectified Linear Unit
- `Sigmoid` : squashes output between 0 and 1
- `Tanh` : squashes output between -1 and 1

> Perceptrons typically use a **step function** as the activation, but modern neural nets often use `ReLU` .

Epoch

One **epoch** means one full pass over the entire training dataset. Multiple epochs are used so the model can keep refining its understanding.

Training a Perceptron with scikit-learn

1. Import Libraries

```python
from sklearn.linear_model import Perceptron
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
```

---

2. Create and Split Data

```python
X, y = make_classification(n_samples=1000, n_features=10, n_classes=2,
random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)
```

---

3. Initialize the Perceptron

```python
clf = Perceptron(
    max_iter=1000,     # Maximum number of epochs
    eta0=0.1,          # Learning rate
    random_state=42,   # For reproducibility
    tol=1e-3,          # Stop early if improvement is smaller than this
    shuffle=True       # Shuffle data each epoch
)
```

---

4. Train the Model

```
clf.fit(X_train, y_train)
```

Under the hood, this performs the following steps:

- Loops through the data up to `max_iter` times (epochs)
- Computes predictions
- If a prediction is wrong, updates weights

---

5. Evaluate the Model

```
accuracy = clf.score(X_test, y_test)
print(f"Accuracy: {accuracy:.2f}")
```

Important Hyperparameters Recap:

Hyperparameter	Description
`max_iter`	Number of epochs (passes over training data)
`eta0`	Learning rate
`tol`	Tolerance for stopping early
`shuffle`	Whether to shuffle data between epochs
`random_state`	Seed for reproducibility

TensorFlow vs Keras vs PyTorch

Deep learning has transformed industries—from self-driving cars to language models like ChatGPT. But behind the scenes, there are powerful libraries that make all of this possible: **TensorFlow**, **Keras**, and **PyTorch**. Today, we'll explore:

- Why and how each library was created
- How they differ in philosophy and design
- What a "backend" means in Keras
- Which one might be right for you

A Brief History

1. TensorFlow

- **Released**: 2015 by **Google Brain**
- **Language**: Python (but has C++ core)
- **Goal**: Provide an **efficient**, **production-ready**, and **scalable** library for deep learning.
- TensorFlow is a **computational graph framework**, meaning it represents computations as nodes in a graph.
- Open sourced in 2015, TensorFlow quickly became the go-to library for many companies and researchers.

Fun Fact: TensorFlow was a spiritual successor to Google's earlier tool called **DistBelief**.

---

2. Keras

- **Released**: 2015 by **François Chollet**

- **Goal**: Make deep learning **simple**, **intuitive**, and **user-friendly**.
- Keras was originally just a high-level wrapper over **Theano** and **TensorFlow**, making it easier to build models with fewer lines of code.
- Keras introduced the idea of writing deep learning models like stacking Lego blocks.

Keras was not a full deep learning engine—it needed a "backend" to actually do the math

---

3. PyTorch

- **Released**: 2016 by **Facebook AI Research (FAIR)**
- **Language**: Python-first, with a strong integration to NumPy
- **Goal**: Make deep learning **flexible**, **dynamic**, and **easier for research**.
- PyTorch uses **dynamic computation graphs**, meaning the graph is built **on the fly**, allowing for more intuitive debugging and flexibility.

PyTorch gained massive popularity in academia and research because of its Pythonic nature and simplicity.

In a nut shell...

Feature	TensorFlow	Keras	PyTorch
Developed By	Google	François Chollet (Google)	Facebook (Meta)
Level	Low-level & high-level	High-level only	Low-level (with some high-level APIs)
Computation Graph	Static (TensorFlow 1.x), Hybrid (2.x)	Depends on backend	Dynamic
Ease of Use		Very high	High

Feature	TensorFlow	Keras	PyTorch
	Medium (Better in 2.x)		
Debugging	Harder (in 1.x), Easier in 2.x	Easy	Very easy (Pythonic)
Production-ready	Yes	Yes (via TensorFlow backend)	Gaining ground
Research usage	Moderate	Moderate	Very high

What Is a “Backend” in Keras?

Keras itself **doesn’t perform computations** like matrix multiplications or gradient descent. It’s more like a **user interface** or a **frontend**. It delegates the heavy lifting to a **backend engine**.

Supported Backends Over Time:

- **Theano** (now discontinued)
- **TensorFlow** (default backend now)
- **CNTK** (Microsoft, also discontinued)
- **PlaidML** (experimental support)

You can think of Keras as the *steering wheel*, while TensorFlow or Theano was the *engine* under the hood.

In **TensorFlow 2.0 and later**, Keras is fully integrated as `tf.keras`, eliminating the need to manage separate backends.

TensorFlow vs PyTorch: The Real Battle

Area	TensorFlow	PyTorch
Ease of Deployment	TensorFlow Serving, TFX, TensorFlow Lite	TorchServe, ONNX, some catching up
Mobile Support	Excellent (TF Lite, TF.js)	Improving but limited
Dynamic Graphs	TF 1.x: No, TF 2.x: Yes (via Autograph)	Native support
Community	Large, especially in production	Massive in academia
Performance	Highly optimized	Also strong, especially for GPUs

---

Which One Should You Learn First?

- **Beginner?** → Start with **Keras** via **TensorFlow 2.x** (`tf.keras`). It's simple and production-ready.
- **Researcher or experimenting a lot?** → Use **PyTorch**.
- **Looking for deployment & scalability?** → **TensorFlow** is very robust.

Installing TensorFlow 2.0

Today we will install TensorFlow 2.0, which is a powerful library for machine learning and deep learning tasks. TensorFlow 2.0 simplifies the process of building and training models, making it more user-friendly compared to its predecessor.

Understanding and Visualizing the MNIST Dataset with TensorFlow

The **MNIST** dataset is like the “Hello World” of deep learning. It contains **70,000 grayscale images** of handwritten digits (0–9), each of size **28x28 pixels**. Before we train a neural network, it’s important to *understand what we’re working with*.

Today we’ll:

- Load the MNIST dataset using TensorFlow
- Visualize a few digit samples
- Understand the data format

Step 1: Load the Dataset

TensorFlow provides a built-in method to load MNIST, so no extra setup is needed.

```
import tensorflow as tf
import matplotlib.pyplot as plt

# Load dataset
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()

# Check the shape
print("Training data shape:", x_train.shape)
print("Training labels shape:", y_train.shape)
```

Output:

```
Training data shape: (60000, 28, 28)
Training labels shape: (60000,)
```

- We have 60,000 training images and 10,000 test images.
- Each image is 28x28 pixels.
- Each label is a number from 0 to 9.

Step 2: Visualize Sample Digits

Let's look at a few images to get a feel for the dataset:

```
# Plot first 10 images with their labels
plt.figure(figsize=(10, 2))
for i in range(10):
    plt.subplot(1, 10, i + 1)
    plt.imshow(x_train[i], cmap="gray")
    plt.axis("off")
    plt.title(str(y_train[i]))
plt.tight_layout()
plt.show()
```

This will display the first 10 handwritten digits with their corresponding labels above them.

What You Should Notice

- The digits vary in **writing style**, which makes this dataset great for teaching computers to generalize.
- All images are normalized 28x28 grayscale—**no color channels**.
- The label is **not embedded** in the image; it's provided separately.

Your First Neural Network

TensorFlow is an open-source deep learning library developed by Google. It's widely used in industry and academia for building and training machine learning models. **TensorFlow 2.0** brought significant improvements in ease of use, especially with **eager execution** and tight integration with Keras.

In this tutorial, we'll create a **simple neural network** that learns to classify handwritten digits using the **MNIST dataset**. This dataset contains 28x28 grayscale images of digits from 0 to 9.

Key Features of TensorFlow 2.0

- **Eager execution** by default (no more complex session graphs!)
- **Keras as the official high-level API** (`tf.keras`)
- Better **debugging and simplicity**
- Great for both beginners and professionals

What is a Neural Network?

A **neural network** is a collection of layers that learn to map input data to outputs. Think of layers as filters that extract meaningful patterns. Each layer applies transformations using **weights** and **activation functions**.

Installation

```
pip install tensorflow
```

Importing Required Libraries

```python
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical
```

Loading and Preparing the Data

```python
# Load the data
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Normalize the input data
x_train = x_train / 255.0
x_test = x_test / 255.0

# One-hot encode the labels
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)
```

Building a Simple Neural Network

```python
model = Sequential([
    Flatten(input_shape=(28, 28)),     # 28x28 images to 784 input features
    Dense(128, activation='relu'),     # Hidden layer with 128 neurons
    Dense(10, activation='softmax')    # Output layer for 10 classes
])
```

Compiling the Model

```
model.compile(
    optimizer='adam',
    loss='categorical_crossentropy',
    metrics=['accuracy']
)
```

Training the Model

```
model.fit(x_train, y_train, epochs=5, batch_size=32)
```

Evaluating the Model

```
test_loss, test_acc = model.evaluate(x_test, y_test)
print(f"Test accuracy: {test_acc:.4f}")
```

Web Development for Data Scientists

Why Learn Web Development as a Data Scientist?

Most data science work involves exploring datasets, building models, and analyzing results. However, in many real-world situations, the value of your work increases significantly when others can interact with it. Web development provides a way to make your models and insights accessible through simple user interfaces or APIs.

Even a basic understanding of web technologies allows you to:

- Present your results beyond notebooks
- Create simple forms to collect user input
- Serve your machine learning models via a web application
- Share interactive visualizations or summaries

How Much Web Development is Enough?

As a data scientist, you do not need to become a full-stack web developer. Instead, it’s more practical to focus on a few key skills that complement your existing workflow:

- **HTML** to create and structure web pages
- **CSS** to control the appearance and layout
- **Flask (Python Framework)** to build lightweight web applications and APIs

These skills are sufficient for creating interfaces where users can interact with your models or view results dynamically.

A Practical Approach

This course introduces just enough web development to support your work as a data scientist. The goal is to help you understand how to connect the outputs of your models with simple, user-friendly web interfaces using tools you’re already comfortable with—primarily Python.

We will begin with foundational concepts like HTML and CSS, and then move to using Flask for building basic applications.

HTML for Data Scientists

What is HTML?

HTML (HyperText Markup Language) is the standard language used to define the structure of web pages. It tells the browser how to display content like text, forms, buttons, and tables.

As a data scientist, you can use HTML to:

- Create forms that take input from users
- Display model results on a page
- Structure information clearly

Basic HTML Elements

Page Structure

```html
<!DOCTYPE html>
<html>
<head>
  <title>My Web App</title>
</head>
<body>
  <!-- Content goes here -->
</body>
</html>
```

Headings and Paragraphs

```html
<h1>Model Results</h1>
<p>This page shows the output of a machine learning model.</p>
```

Forms and Inputs

```html
<form action="/predict" method="post">
  <label for="feature">Enter a value:</label>
  <input type="text" name="feature" id="feature">
  <button type="submit">Predict</button>
</form>
```

Displaying Tables

```html
<table>
  <tr>
    <th>Feature</th>
    <th>Value</th>
  </tr>
  <tr>
    <td>Age</td>
    <td>29</td>
  </tr>
</table>
```

Summary

You only need to learn a small set of HTML tags to create basic interfaces:

- Structural: `<html>` , `<head>` , `<body>`
- Text: `<h1>` to `<h6>` , `<p>` , `<span>`
- Forms: `<form>` , `<input>` , `<button>`
- Tables: `<table>` , `<tr>` , `<td>` , `<th>`

This is enough to build simple pages that collect input and display results. You will connect these pages with Python using Flask in later lessons.

CSS for Data Scientists

What is CSS?

CSS (Cascading Style Sheets) is used to control the appearance of HTML elements. It allows you to change fonts, colors, spacing, and layout.

For data scientists, CSS helps make basic web interfaces more readable and user-friendly. Even a few lines of CSS can make your pages easier to use.

Ways to Use CSS

Inline CSS (directly in the HTML tag)

```
<p style="color: green;">Prediction Successful</p>
```

Internal CSS (inside a `<style>` block)

```
<head>
  <style>
    body {
      font-family: Arial, sans-serif;
      margin: 40px;
    }
    input {
      padding: 5px;
      margin-bottom: 10px;
    }
    button {
      background-color: #007BFF;
      color: white;
      border: none;
      padding: 8px 12px;
    }
  </style>
</head>
```

Common CSS Properties

Property	What it does
`color`	Text color
`background`	Background color
`margin`	Space outside the element
`padding`	Space inside the element
`font-size`	Size of the text
`border`	Adds a border around elements
`text-align`	Aligns text (left, center, right)

Example

```
<body>
  <h1>Prediction Result</h1>
  <form>
    <input type="text" placeholder="Enter value">
    <button type="submit">Submit</button>
  </form>
</body>
```

With the internal CSS above, this form will have padding and better spacing.

Summary

You don’t need to master CSS as a data scientist. Knowing how to:

- Set padding and margins
- Style buttons and inputs
- Control font and color

is often enough to make your tools more usable. We will now move on to connecting everything using Python and Flask.

Introduction to Flask Framework**

Why Flask?

- Lightweight Python web framework
- Ideal for serving models and exposing APIs

Installation

```
pip install flask
```

Demo App

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def home():
    return 'Hello, Data Scientist!'

if __name__ == '__main__':
    app.run(debug=True)
```

Running the Server

```
python app.py
```

Then go to `http://127.0.0.1:5000` in your browser.

Query Parameters in Flask

What Are Query Parameters?

Query parameters are key-value pairs passed in the URL after a `?`. For example:

```
/predict?x=10\&y=20
```

In this case: - `x` has the value `10` - `y` has the value `20`

They are often used to: - Pass values to your route - Trigger filtering or model predictions without using a form

Accessing Query Parameters in Flask

Use `request.args` to access query parameters:

```
from flask import Flask, request

app = Flask(__name__)

@app.route('/predict')
def predict():
    x = request.args.get('x')
    y = request.args.get('y')
    return f"x = {x}, y = {y}"
```

Example

Visiting this URL:

```
http://localhost:5000/predict?x=12&y=7
```

Will return:

```
x = 12, y = 7
```

Handling Missing Values

You can provide a default value:

```
x = request.args.get('x', default=0)
```

Or use an `if` statement to check if the parameter exists:

```
if 'x' in request.args:
    x = request.args['x']
```

Use Case Example

You might want to use query parameters to trigger a model prediction:

```
@app.route('/predict')
def predict():
    feature = request.args.get('value')
    if not feature:
        return "No value provided"
    # prediction = model.predict([feature])  # pseudocode
    return f"Prediction result for input {feature}"
```

Summary

- Query parameters are passed using `?key=value` in the URL

- Use `request.args.get('key')` to retrieve them
- Useful for triggering actions or passing values without forms

Serving Static Files in Flask

What Are Static Files?

Static files are files that don't change during execution. These typically include:

- CSS files
- JavaScript files
- Images and other media

Flask serves static files by default from a folder named `static`, and they are accessible via the `/static/` URL path.

Default Project Structure

```
my_app/
├── app.py
├── static/
│   └── style.css
├── templates/
│   └── index.html
```

Referencing Static Files in HTML

In your HTML templates (inside the `templates/` folder), use Flask's `url_for()` function to correctly link to static assets:

```
<link rel="stylesheet" href="{{ url_for('static', filename='style.css') }}">
```

This will generate `/static/style.css` by default.

Changing the Static Folder and URL Path

If you want Flask to serve static files from a different folder or under a different URL, use these options when creating your Flask app:

```
from flask import Flask

app = Flask(
    __name__,
    static_folder='assets',          # Physical folder on disk
    static_url_path='/files'         # URL path to access those files
)
```

In this setup:

- Flask will **look for files in the `assets/` folder**
- Users will **access files via `/files/...`** in the browser

Example:

```
<link rel="stylesheet" href="{{ url_for('static', filename='style.css') }}">
```

Generates:

```
/files/style.css
```

Summary

- `static_folder` : changes the folder Flask looks into for static files (default is `static/`)
- `static_url_path` : changes the URL path used to serve those static files (default is `/static`)
- Use `url_for('static', filename='...')` to refer to static files regardless of the folder or path

Handling Forms in Flask

Why Use Forms?

Forms allow users to input values that can be processed by your Flask app. This is useful for: - Passing feature values to a machine learning model - Uploading data - Triggering analysis

Creating a Simple HTML Form

Here is a basic HTML form that accepts a single input:

```html
<form action="/predict" method="post">
  <input type="text" name="feature1" placeholder="Enter a value">
  <button type="submit">Submit</button>
</form>
```

- `action="/predict"` : the form will send data to the `/predict` route
- `method="post"` : the form uses POST request to send data

Handling Form Data in Flask

In your `app.py` , set up a route to receive and process the form data:

```python
from flask import Flask, request, render_template

app = Flask(__name__)

@app.route('/')
def index():
    return render_template('form.html')
```

```
@app.route('/predict', methods=['POST'])
def predict():
    value = request.form['feature1']
    return f"Received input: {value}"
```

Explanation:

- `request.form` is used to access submitted form data
- The key `'feature1'` corresponds to the `name` attribute in the HTML input

Example File Structure

```
my_app/
├── app.py
├── templates/
│   └── form.html
```

form.html Template

```
<!DOCTYPE html>
<html>
<head>
  <title>Prediction Input</title>
</head>
<body>
  <h2>Enter Input</h2>
  <form action="/predict" method="post">
    <input type="text" name="feature1">
    <button type="submit">Submit</button>
  </form>
</body>
</html>
```

Summary

- Use `<form>` in your HTML to collect user input
- Use `request.form['input_name']` in your Flask route to read submitted data
- Forms enable dynamic interaction with your web app, such as sending input to a model or triggering analysis

Jinja2 Templating in Flask

What is Jinja2?

Jinja2 is the templating engine used by Flask. It allows you to embed Python-like logic inside your HTML templates. This is useful when you want to:

- Display variables (like model predictions or user input)
- Loop through lists (like rows in a DataFrame)
- Use conditions (like showing a message only when needed)

Passing Variables from Flask to Templates

In your Flask route, use `render_template()` and pass variables like this:

```python
from flask import Flask, render_template

app = Flask(__name__)

@app.route('/')
def home():
    result = "Prediction: 42"
    return render_template('index.html', prediction=result)
```

Here, `prediction` is a variable you can use inside `index.html`.

Using Variables in HTML

Inside your HTML template (`index.html`), use double curly braces `{{ }}` to display variables:

```html
<!DOCTYPE html>
<html>
<head>
  <title>Result</title>
</head>
<body>
  <h1>Model Output</h1>
  <p>{{ prediction }}</p>
</body>
</html>
```

Control Structures

If Statements

```
{% if prediction %}
  <p>Result: {{ prediction }}</p>
{% else %}
  <p>No result available.</p>
{% endif %}
```

For Loops

You can loop through a list or dictionary:

```
<ul>
  {% for item in items %}
    <li>{{ item }}</li>
  {% endfor %}
</ul>
```

In your Flask route:

```
@app.route('/list')
def show_list():
    return render_template('list.html', items=['Apple', 'Banana', 'Cherry'])
```

Escaping Output

Jinja2 automatically escapes variables to prevent HTML injection. If you trust the variable and want to allow raw HTML, use `|safe`:

```
<p>{{ some_html | safe }}</p>
```

Summary

- Use `render_template()` to pass variables from Flask to HTML
- Use `{{ variable }}` to display data
- Use `{% ... %}` for control structures like loops and conditionals
- Jinja2 makes it easy to combine logic and layout for dynamic pages

Template Inheritance in Flask (Jinja2)

What is Template Inheritance?

When building web apps, many pages share a common layout — for example, a header, footer, or navigation bar. Instead of repeating this code in every file, Jinja2 allows you to define a **base template** that other templates can extend.

This keeps your code cleaner and easier to maintain.

Step 1: Create a Base Template

Create a file called `base.html` inside your `templates/` folder.

```
<!DOCTYPE html>
<html>
<head>
  <title>{% block title %}My App{% endblock %}</title>
</head>
<body>
  <header>
    <h1>My Flask App</h1>
  </header>

  <main>
    {% block content %}{% endblock %}
  </main>
</body>
</html>
```

- `{% block title %}` and `{% block content %}` are placeholders that child templates can fill.

Step 2: Extend the Base Template

Create another HTML file, e.g. `index.html`, that extends the base template:

```
{% extends "base.html" %}

{% block title %}Home Page{% endblock %}

{% block content %}
  <p>Welcome to the homepage.</p>
{% endblock %}
```

Flask Route Example

```
from flask import Flask, render_template

app = Flask(__name__)

@app.route('/')
def home():
    return render_template('index.html')
```

Folder Structure

```
my_app/
├── app.py
├── templates/
│   ├── base.html
│   └── index.html
```

Summary

- Create a `base.html` with `{% block %}` sections

- Use `{% extends "base.html" %}` in child templates
- This avoids repetition and keeps your HTML organized

Template inheritance is useful as your app grows and you start creating multiple pages that share the same layout.

Message Flashing in Flask

What Is Flashing?

Flashing is a way to send temporary messages from the backend (Flask) to the frontend (HTML). These messages are usually used for:

- Status updates (e.g., “Prediction complete”)
- Error messages (e.g., “Invalid input”)
- Notifications (e.g., “File uploaded successfully”)

Flashed messages are stored in the session and automatically cleared after being displayed.

Step 1: Set a Secret Key

Flashing uses Flask’s session, so you must set a secret key:

```
from flask import Flask

app = Flask(__name__)
app.secret_key = 'your_secret_key'
```

Step 2: Flash a Message in Your Route

Use `flash()` to send a message:

```
from flask import flash, redirect, render_template, request

@app.route('/predict', methods=['POST'])
def predict():
    feature = request.form.get('feature1')
    if not feature:
        flash('Please enter a value')
        return redirect('/')

    # process prediction here
    flash('Prediction complete')
    return redirect('/')
```

- `flash('message')` stores the message
- `redirect('/')` sends the user back to a route where the message will be displayed

Step 3: Display Flashed Messages in the Template

In your base or main template (e.g. `index.html` or `base.html`):

```
{% with messages = get_flashed_messages() %}
  {% if messages %}
    <ul>
      {% for msg in messages %}
        <li>{{ msg }}</li>
      {% endfor %}
    </ul>
  {% endif %}
{% endwith %}
```

This block retrieves all flashed messages and displays them in a list. You can style them with CSS as needed.

Summary

- Use `flash('your message')` to send a message
- Use `get_flashed_messages()` in your template to retrieve and display them
- Useful for alerts, validation feedback, and user notifications

Creating an API Using `jsonify` in Flask

What Is an API?

An API (Application Programming Interface) allows other programs (like a frontend, script, or another server) to communicate with your Flask app using structured data—typically JSON.

This is useful when: - You want to expose your ML model as a service - You want other tools to send input and receive predictions

What Is `jsonify`?

Flask's `jsonify()` function converts Python dictionaries or lists into properly formatted JSON responses.

Example: Basic API Endpoint

```python
from flask import Flask, request, jsonify

app = Flask(__name__)

@app.route('/api/predict', methods=['GET'])
def predict():
    value = request.args.get('value')

    if value is None:
        return jsonify({'error': 'Missing input'}), 400

    # Simulated prediction (replace with actual model call)
    result = int(value) * 2
    return jsonify({
        'input': value,
        'prediction': result
    })
```

How to Call It

Visit this in your browser or use a tool like Postman or `curl`:

```
http://localhost:5000/api/predict?value=7
```

Response:

```
{
  "input": "7",
  "prediction": 14
}
```

Returning Error Codes

You can return custom status codes with `jsonify()`:

```
return jsonify({'error': 'Invalid input'}), 400
```

This helps the client understand what went wrong.

Summary

- Use `jsonify()` to return JSON from your Flask routes
- Ideal for serving ML model outputs or exposing data
- Combine with `request.args` (GET) or `request.json` (POST) for flexible input handling

Introduction to LLMs (Large Language Models)

Large Language Models (LLMs) are a breakthrough in artificial intelligence that have revolutionized how machines understand and generate human language. These models are capable of performing a wide range of tasks such as translation, summarization, question answering, and even creative writing — all by learning from massive text datasets.

In this section, we will build a foundational understanding of what LLMs are, why they matter in data science, and how they differ from traditional machine learning models.

What is an LLM?

A **Large Language Model** is a type of AI model that uses deep learning, specifically transformer architectures, to process and generate natural language. These models are "large" because they contain **billions (or even trillions) of parameters** — tunable weights that help the model make predictions.

At their core, LLMs are trained to **predict the next word** in a sentence, given the words that came before. With enough data and training, they learn complex language patterns, world knowledge, and even reasoning skills.

Why are LLMs Important?

- **Versatility**: One LLM can perform dozens of tasks without needing task-specific training.
- **Zero-shot and few-shot learning**: LLMs can handle tasks they've never explicitly seen before, based on prompts or examples.

- **Human-like generation**: They produce text that is often indistinguishable from human writing.
- **Foundation for AI applications**: They power modern tools like ChatGPT, Copilot, Bard, Claude, and more.

How are LLMs Different from Traditional ML Models?

Feature	Traditional ML Models	LLMs
Input	Structured data	Natural language (text)
Training	Task-specific	General pretraining on large text
Parameters	Thousands to millions	Billions to trillions
Adaptability	Limited	Highly adaptable via prompting
Knowledge representation	Feature-engineered	Implicit via word embeddings

---

Where are LLMs Used?

LLMs are widely used across industries:

- **Customer support**: Chatbots and automated help desks
- **Education**: AI tutors, personalized learning
- **Healthcare**: Clinical documentation and patient interaction
- **Software Development**: Code generation and bug detection
- **Creative fields**: Story writing, poetry, music lyrics

History of LLMs

Understanding the history of Large Language Models (LLMs) helps us appreciate how far we’ve come in natural language processing (NLP) and the innovations that made today’s AI systems possible.

This section walks through the key milestones — from early statistical models to the modern transformer revolution.

Early NLP Approaches

Before LLMs, language tasks were handled using:

- **Rule-based systems**: Manually written logic for grammar and syntax.
- **Statistical models**: Such as *n-gram models*, which predicted the next word based on a fixed window of previous words.
- **Bag-of-words** and **TF-IDF**: Used for basic text classification but ignored word order and context.

These models worked for simple tasks, but failed to capture deeper meaning, semantics, or long-range dependencies in language.

The Rise of Neural Networks

With the rise of deep learning, models began learning richer representations:

- **Word Embeddings** like **Word2Vec** (2013) and **GloVe** (2014) mapped words to continuous vector spaces.
- **Recurrent Neural Networks (RNNs)** and **LSTMs** enabled models to process sequences, but they struggled with long texts and parallel processing.

Transformers: The Game Changer

In 2017, Google introduced the **Transformer architecture** in the paper *"Attention is All You Need."*

Key features of transformers:

- **Self-attention** mechanism allows the model to weigh the importance of different words, regardless of their position.
- Enables **parallelization**, making training on massive datasets feasible.

This led to a new generation of LLMs:

Model	Year	Key Contribution
BERT	2018	Bidirectional context understanding
GPT-1	2018	Introduced unidirectional generation
GPT-2	2019	Generated coherent long-form text
T5	2020	Unified text-to-text framework
GPT-3	2020	175B parameters, capable of few-shot learning
ChatGPT / GPT-3.5 / GPT-4	2022–2023	Conversational abilities, better reasoning
Claude, Gemini, LLaMA, Mistral	2023+	Open-source and scalable alternatives

---

Pretraining & Finetuning

The modern LLM pipeline consists of:

1. **Pretraining** on a large corpus of general text (e.g., books, Wikipedia, web pages).
2. **Finetuning** for specific tasks (e.g., summarization, coding help).

3. **Reinforcement Learning from Human Feedback (RLHF)** — used to make models safer and more helpful (e.g., ChatGPT).

Summary

The evolution of LLMs is a story of scale, data, and architecture. The shift from handcrafted rules to deep neural transformers has allowed machines to understand and generate language with remarkable fluency.

How LLMs Work

In this section, we break down the inner workings of Large Language Models (LLMs). While these models seem like magic from the outside, they are grounded in fundamental machine learning and deep learning principles — especially the transformer architecture.

We'll go through how LLMs process text, represent meaning, and generate coherent outputs.

Tokenization: Breaking Text into Units

LLMs do not process text as raw strings. Instead, they break input text into smaller units called **tokens**. Tokens can be:

- Whole words (for simple models)
- Subwords (e.g., “un” + “believ” + “able”)
- Characters (rare for LLMs, used in specific domains)

This process helps reduce the vocabulary size and handle unknown or rare words efficiently.

Popular tokenizers include Byte-Pair Encoding (BPE) and SentencePiece.

Embeddings: Converting Tokens to Vectors

Once text is tokenized, each token is mapped to a high-dimensional vector through an **embedding layer**. These embeddings capture relationships between words based on context.

For example, the words “king” and “queen” will be closer in the embedding space than unrelated words like “banana” or “car”.

The Transformer Architecture

The core of LLMs is the **transformer**, introduced in 2017. It replaced earlier models like RNNs and LSTMs by allowing for better performance and scalability.

Key components of a transformer:

- **Self-Attention Mechanism**: Enables the model to focus on different parts of the input when processing each token. For example, in the sentence “The cat sat on the mat,” the word “sat” may attend more to “cat” and “mat” than to “the”.
- **Multi-Head Attention**: Allows the model to capture different types of relationships simultaneously.
- **Feedforward Networks**: Add depth and complexity to the model.
- **Positional Encoding**: Since transformers process all tokens in parallel, they need a way to encode the order of tokens.

These components are stacked in layers — more layers typically mean more modeling power.

Training LLMs: Predicting the Next Token

LLMs are trained using a simple but powerful objective: **predict the next token given the previous tokens**.

For example:

- Input: “The sun rises in the”
- Output: “east”

The model adjusts its internal weights using a large dataset and **gradient descent** to minimize prediction error. Over billions of examples, the model learns grammar, facts, reasoning patterns, and even basic common sense.

This process is known as **causal language modeling** in models like GPT. Other models like BERT use **masked language modeling**, where random tokens are hidden and the model must predict them.

Generation: Producing Human-like Text

Once trained, the model can generate text by predicting one token at a time:

1. Start with an input prompt.
2. Predict the next token based on context.
3. Append the new token to the prompt.
4. Repeat until a stopping condition is met.

Several sampling strategies control the output:

- **Greedy decoding**: Always choose the most likely next token.
- **Beam search**: Explore multiple token sequences in parallel.
- **Top-k / top-p sampling**: Add randomness for more creative or diverse outputs.

Limitations of LLMs

Despite their capabilities, LLMs have limitations:

- **No true understanding**: They learn patterns, not meaning.
- **Hallucinations**: They can generate plausible but false information.
- **Bias**: Trained on large web corpora, they can inherit societal biases.
- **Compute-intensive**: Training and running LLMs requires significant hardware resources.

Introduction to RAG-based Systems

As Large Language Models (LLMs) become central to modern AI applications, a key limitation remains: **they don’t know anything beyond their training data**. They cannot access up-to-date information or internal company documents unless explicitly provided.

This is where **RAG** (Retrieval-Augmented Generation) systems come in. RAG bridges the gap between language generation and external knowledge, making LLMs more **accurate, dynamic, and context-aware**.

What is a RAG System?

Retrieval-Augmented Generation (RAG) is an AI architecture that combines:

1. **A retriever** – to search a knowledge base for relevant documents or facts.
2. **A generator (LLM)** – to synthesize a response using both the retrieved content and the input question.

Rather than generating answers purely from internal memory (which may be outdated or incomplete), a RAG system fetches real documents and grounds the model’s output in that information.

Why Use RAG?

RAG addresses several key challenges of LLMs:

Problem in LLMs	How RAG Helps
Hallucination (made-up facts)	Anchors generation in real data

Problem in LLMs	How RAG Helps
Outdated knowledge	Uses fresh, external sources like databases or websites
Limited context window	Dynamically injects only the most relevant info
Domain-specific needs	Connects model to private corpora, PDFs, etc.

Basic Workflow of a RAG System

1. **User query** →
2. **Retriever** fetches relevant documents (e.g., from a vector database) →
3. **Documents + query** are passed to the LLM →
4. **LLM generates** a grounded, accurate response.

This loop allows the model to act more like a researcher with access to a searchable library.

Components of a RAG Pipeline

- **Embedding Model**: Converts text into dense vectors to enable similarity search.
- **Vector Store**: A searchable index (e.g., FAISS, Weaviate, Pinecone) where document embeddings are stored.
- **Retriever**: Queries the vector store with the input to find top-k most similar documents.
- **LLM**: Uses the retrieved content and prompt to generate a final response.
- **Optional Reranker**: Improves retrieval quality by reordering results.

Example Use Cases

- **Enterprise chatbots**: Pull answers from internal documents and manuals.
- **Customer support**: Query knowledge bases in real-time.
- **Academic research tools**: Generate summaries grounded in actual papers.
- **Healthcare assistants**: Retrieve clinical guidelines or patient history for personalized advice.