SmartStock Inventory Optimization for Retail Stores

Smart Stock Inventory System – Overview

Introduction

A Smart Stock Inventory System is a digital solution designed to help retail stores track their products automatically. It replaces manual stock counting with technology that updates inventory in real time, reduces errors, and ensures that important items are never out of stock.

Main Objective

- To monitor stock levels accurately and automatically
- To alert store owners about low stock, overstock, or expired items
- To make daily operations faster, smarter, and more organized

Python basics

1. Python Casting

- Casting = data type conversion
- Common conversions:
    - int() → integer
    - float() → floating number
    - str() → string
- Example: float(5) = 5.0

---

2. Function Initialization (def)

- Syntax:
- def function_name():

- # code
- Function can have parameters and return values**.**

---

3. f-string

- f-string =Used to insert variables directly into a string using {}.
- Fastest & cleanest string formatting.
- name = "Mannat"
- print(f"Hello {name}")

---

4. NumPy

- NumPy = Numerical Python (fast math library).

5. Pandas

- Pandas = data analysis and data cleaning library.
- DataFrame .
- Important operations:
  - df.head()
  - df.describe()

---

6. Matplotlib

- graph/visualization library.
- Graph types:
  - Line → plt.plot()
  - Bar → plt.bar()
  - Pie → plt.pie()
  - Scatter → plt.scatter()
- Basic functions:
  - plt.title(), plt.xlabel(), plt.ylabel(), plt.show()

AI MODELS

AI models are computer programs that **learn patterns from data** and then use that learning to **predict, understand, or generate** things on their own.

Company	Famous Model Family
Google	**Gemini**
OpenAI	**GPT** (Generative Pre-trained Transformer)
Groq	**Llama**

What is an LLM?

LLM = Large Language Model

It is an AI model trained on huge amounts of text to understand and generate human-like language.

Examples of LLMs

These are all LLMs:

OpenAI

- GPT-5
- GPT-4o
- GPT-3.5

Google

- Gemini
- Gemma

Meta

- Llama 3
- Llama 3.1

LLMs are trained using large datasets.

Chat Completion means using an AI model (LLM) to generate the next response in a conversation.
This is how GPT, Gemini, Groq (Llama), Claude, etc. work.

What is Chat Completion?

It is an API style where you send:

- messages (user + assistant history)
- The model replies with the next message

What is PostgreSQL (Postgres)?

PostgreSQL is an **open-source relational database management system (RDBMS)**.

In simple words:
It stores your data in **tables** (rows & columns)
Helps you **query**, **insert**, **update**, **delete**, and **manage data**
Uses **SQL** language

1. Type of Database

- **PostgreSQL** → *Object–Relational Database* (supports advanced data types, complex queries).
- **MySQL** → *Relational Database* (simple structure, very fast for basic queries).

---

2. Speed

- **PostgreSQL** → Slower for simple reads but *faster for complex queries*.
- **MySQL** → Very *fast for simple read-heavy operations*.

Why do we need Vector Databases?

Normal databases (**MySQL, PostgreSQL**) can't search:

- similar meaning sentences
- similar images

- related documents

They only match *exact words*, not *meaning*.

Vector DBs search using **similarity**, like:

- "Find documents similar to this one"
- "Find images that look like this picture"
- "Find product recommendations"

3 Main Types of Machine Learning

Supervised Learning

Learning with labeled data
The model is given **input + correct output**, and it learns the mapping.

Used for:

- Predicting prices
- Classifying emails (spam/not spam)
- Face recognition

Unsupervised Learning

Learning without labeled data
The model finds **patterns and groups** on its own.

Used for:

- Market segmentation
- Anomaly detection
- Recommendation systems

Reinforcement Learning (RL)

Learning by trial and error
The model gets **rewards** for good actions and **punishments** for bad actions.

Semi-Supervised Learning

Some data → labeled
Most data → unlabeled

Used when labeling is expensive.
Examples:

- Google Photos
- Medical images

1. API (Application Programming Interface)

Definition:
An API is a set of rules that allows two software applications to communicate and exchange data.

Simple meaning:
API works like a **messenger** between two programs. One program sends a request, API forwards it, and returns the response.

Real-life example:
A weather app requests today's temperature using a weather API and displays the result.

Benefits:

- Easy communication between software
- Reuse features without rewriting code
- Saves development time
- Secure, fast, structured communication

---

2. API Request Methods

GET Method

- Used to **fetch data** from the server.
- No request body/payload.
- Example: Getting user details.

POST Method

- Used to **send or create data** on the server.
- Contains payload (JSON data).

- Example: Creating a new user.

---

3. Payload

Definition:
Payload is the **actual data** sent inside an API request or response.

Example:
POST Request Payload:

---

4. Endpoint

Definition:
Endpoint is a **specific URL path** of an API where requests are made.

Examples:

- /api/users – GET all users
- /api/users/1 – GET a user by ID
- /api/users – POST to create a user

---

5. FastAPI vs Flask

Flask

- Lightweight web framework
- Beginner-friendly
- Manually handle validation
- Good for small/medium projects

Example:

```
from flask import Flask, jsonify


app = Flask(__name__)


@app.route("/hello")
def hello():
```

```
    return jsonify({"message": "Hello World"})
```

FastAPI

- Modern & high-performance

Example:

```
from fastapi import FastAPI


app = FastAPI()


@app.get("/hello")
def hello():
    return {"message": "Hello World"}
```

---

6. Python :

Create a virtual environment

```
python -m venv venv
```

or

```
python -m venv tenv
```

Activate environment

Windows:

```
venv\Scripts\activate
```

Deactivate

```
deactivate
```

---

7. requirements.txt

Create requirements file

```
requirements.txt
```

Install from requirements

pip install -r requirements.txt

Purpose:
Helps install the same packages on any system.

---

8. Git Commands

git add

- Adds files to staging area.

Add all files:

git add .

Add specific file:

git add filename.py

---

git commit

Saves staged changes:

git commit -m "message"

---

git push

Sends your commits to remote repository:

git push

---

git branch

View branches:

git branch

View all branches (local + remote):

git branch --all

Switch to another branch

git checkout branch_name

Or newer way:

git switch branch_name

Create and switch to a new branch

git checkout -b new_branch

or

git switch -c new_branch

---

git pull

Fetches latest changes and merges them into your branch:

git pull

---

git fetch --all

Downloads all changes from remote **without merging**:

git fetch –all

Database

A **database** is an organized collection of data that allows easy **storage, retrieval, updating, and management** of information. Databases help keep data structured and accessible for applications like banking systems, websites, social media platforms, and school management systems.

Why Do We Use a Database?

- To store large amounts of data safely
- To avoid mistakes and inconsistency
- To access data quickly
- To allow multiple users to work at the same time
- To keep data organized and structured

Types of Databases

1. Relational Databases (SQL)

Data is stored in **tables** with rows and columns. Uses **SQL language**.
Examples: MySQL, PostgreSQL, Oracle.

2. Non-Relational Databases (NoSQL)

Data stored in **documents, key-value pairs, graphs, or wide columns**.
Examples: MongoDB, Firebase, Cassandra.

Important Database Terms

- **Table:** Collection of rows and columns
- **Row (Record):** One entry of data
- **Column (Field):** Attribute of the data
- **Primary Key:** Unique identifier for each record
- **Foreign Key:** Connects two tables
- **Query:** Command to retrieve or update data
- **Index:** Speeds up searching

Normalization

Normalization is a database process used to **organize data, reduce redundancy, and improve data consistency**.
It breaks a large table into smaller linked tables so that data becomes clean and easy to manage.

Why Normalization Is Important

- Removes duplicate data
- Saves storage
- Ensures consistency
- Avoids update/delete anomalies

- Improves performance
- Makes database structure logical

Normal Forms (1NF, 2NF, 3NF)

First Normal Form (1NF)

Rules:

- No repeating columns
- Values must be **atomic** (single values only)
- Each table must have a **primary key**

Meaning:
Break multi-valued and repeating data into separate rows.

Second Normal Form (2NF)

Rules:

- Table must be in **1NF**
- No **partial dependency**
  (A non-key attribute should not depend on part of a composite key)

Meaning:
Non-key attributes must depend on the **whole primary key**.

Third Normal Form (3NF)

Rules:

- Table must be in **2NF**
- No **transitive dependency**
  (Non-key attribute should not depend on another non-key attribute)

Meaning:
Every column must depend **only** on the primary key.

Summary Table

Normal Form	What It Fixes	Meaning
1NF	Repeating / multi-valued data	Only single values per cell
2NF	Partial dependency	Non-key depends on whole primary key
3NF	Transitive dependency	Non-key depends only on primary key

Git vs GitHub

Git	GitHub
Git is a **software/tool**.	GitHub is a **website/platform**.
Used for **version control** (tracks code changes).	Used for **hosting Git repositories online**.
Works **offline** on your computer.	Works **online** with internet.
Helps you **manage project versions**.	Helps you **share, store, and collaborate** on projects.
Only version control.	Provides extra features: issues, pull requests, teams, actions.

git --version is a command used to **check which version of Git is installed** on your computer.

Git Repository

A **Git repository** is a **storage place** where Git keeps all your project files **and all the history of changes**.

It remembers:

- every file
- every update
- every version

- who changed what

Agentic AI Workflow (Simple Explanation)

Agentic AI is an AI system that can **take actions on its own** to achieve goals, not just answer questions.
It works like a **digital agent** that can plan, decide, and act.

Workflow Steps

1. Goal Definition

- The AI is given a **goal or task** to achieve.
  Example: "Book a flight from Delhi to Jaipur."

---

2. Environment Perception

- The AI **observes the environment** (data, APIs, or system states).
- It collects the **information it needs** to act.

Example: Checking flight availability or prices.

---

3. Planning

- The AI **decides a sequence of actions** to reach the goal.
- It may **consider multiple options** and choose the best one.

Example: Compare flights → choose the cheapest → check schedule.

---

4. Action / Execution

- AI **performs the action** in the environment.
- Can be **automatic tasks** like sending emails, booking tickets, or running scripts.

---

5. Monitoring / Feedback

- The AI **checks if the action worked**.
- If not, it **adjusts** its plan and tries again.

Example: Flight not available → pick next cheapest option.

Agent:

1. check stock levels automatically.
2. Predict when items will be out-of-stock
3. send alert to manager
4. auto-generate purchase orders
5. update demand forecasts daily

Reactive Agent

A **Reactive Agent** is an agent that:

- **Responds immediately** to the current situation.
- **Does not use memory** or past experience.

Robot vacuum → Changes direction when it hits an obstacle

Goal-Driven Agent

A **Goal-Driven Agent** is an agent that:

- Acts to **achieve a specific goal**
- Can **plan its actions** instead of just reacting

A **Multi-Agent System** is a system where **multiple agents work together** to achieve goals.

- Each agent is **independent**
- Agents can **communicate, cooperate, or compete**

How AI Agents Plan, Think, and Act

1. Perception (See/Observe)

- Agent **perceives the environment** using sensors or data.

---

2. Thinking / Reasoning

- Agent **analyzes the information** it perceives.
- Uses **logic, rules, or models** to decide what to do next.
- Can **predict outcomes** of different actions.

---

3. Planning

- Agent **makes a plan or sequence of actions** to achieve a goal.
- Considers **multiple options** and chooses the **best path**.

Example: Self-driving car plans the route to reach the destination safely and fast.

---

4. Acting / Execution

- Agent **performs the chosen actions** in the environment.
- The action is designed to **move closer to the goal**.

Example: Robot moves forward, orders items, or sends alerts.

---

5. Monitoring / Feedback

- Agent **monitors the result** of its actions.
- If the goal is not achieved, it **replans** or adjusts actions.

Example: Stock agent sees a product is still low → sends another alert.

AutoGen Agents

AutoGen frameworks often use **multiple agents** to work together. Two important ones are:

---

1. User Proxy Agent

- Acts as a **representative of the user**.
- **Receives instructions** or goals from the user.
- Communicates the **user's intent** to the assistant agent.

---

2. Assistant Agent

- Acts as the **worker or executor**.
- **Plans, reasons, and performs tasks** to achieve the user's goal.
- Can **use tools, APIs, or other agents** to complete the task.

- **pip install openai**

- `pip install openai` is a **command** used to install the **OpenAI Python library** on your computer.

```
resp = client.chat.completions.create(
    model="gpt-4o-mini",  # replace with an available model like "gpt-4o-mini" or "gpt-4o" or "gpt-5" etc.
    messages=[
      {"role": "system", "content": "You are a helpful assistant."},
      {"role": "user", "content": "Write a two-line poem about chai."}
    ],
    max_tokens=120
  )
```

What This Code Does

This code sends a message to an OpenAI model and asks it to generate a reply.

---

1. resp = client.chat.completions.create(...)

This line **creates a chat completion request**.
Means: "Send my messages to the AI model and give me a response."

---

2. model="gpt-4o-mini"

You are choosing which AI model to use.
Example options:

- gpt-4o-mini
- gpt-4o
- gpt-5

Here it means: **Use the gpt-4o-mini model.**

---

3. messages=[...]

These are the messages we send to the AI.

a) System message

{"role": "system", "content": "You are a helpful assistant."}

This tells the AI:
“Behave like a helpful assistant.”

b) User message

{"role": "user", "content": "Write a two-line poem about chai."}

This is the actual question we want AI to answer.

---

4. max_tokens=120

This limits how long the AI’s answer can be.
Max 120 tokens (roughly words/parts of words).

from openai import OpenAI mean?

This line is a **Python import statement**.

“From the **openai** library, import the **OpenAI** class.”

client = OpenAI() mean?

This line **creates a connection** between your Python program and the **OpenAI API**.

The word **client** means a program that talks to a server (OpenAI).

pip install google-generativeai

This installs Google's AI library (Gemini models).

```
import google.generativeai as genai

# 1. Configure API KEY

genai.configure(api_key="YOUR_API_KEY")

# 2. Create Model

model = genai.GenerativeModel("gemini-1.5-flash")

# 3. Send user message

response = model.generate_content(

    "Write a short poem about coffee."

)
```

4. Get text output

print(response.text)

LLM Text Processing Workflow

Text → Tokens

- Input text is **split into small pieces called tokens**.
- Tokens can be:
    - Words → "Hello"
    - Word parts → "playing" → "play" + "ing"
    - Punctuation → "."
- Purpose: LLM cannot understand raw text directly, so we break it into manageable units.

Tokens → Embeddings

- Each token is converted into a **vector of numbers** called an embedding.
- Embeddings **capture the meaning of the token** in a numerical form.
- Example: "chai" → [0.12, 0.88, 0.45, ...]

Embeddings → Transformer Layers

- The embeddings are passed through **transformer layers**.
- Transformer layers **analyze the relationships** between all tokens in the sequence.
- They contain **multiple sub-layers** like:
    - Feed-forward layers
    - Normalization layers

Attention Mechanism

- Attention lets the model **focus on important tokens** in the input.
- Example: In "I love chai because it is warm", the model understands "chai" is related to "warm".
- Helps the model **capture context and meaning** across the sentence.

Output Generation

- After transformer processing, the model predicts the **next token**.
- Tokens are generated one by one → converted back to text → final response.

Prompt Engineering

What is Prompt Engineering?

- **Prompt engineering** is the process of **designing and refining the input (prompt) given to an AI model** so that it produces the desired output.
- A **prompt** is the text you give the model, e.g., a question, instruction, or request.
- • **Rate Limit:** Maximum requests per short period (e.g., per minute).
- • **Quota:** Maximum total usage allowed over a long period (e.g., per day/month).

Logistic Regression

- Used for **binary classification**: 0/1, Yes/No, Spam/Not Spam
- Linear equation:

$$z = w_1x_1 + w_2x_2 + ... + b$$

- Sigmoid function converts z to probability:

$$p = \frac{1}{1 + e^{-z}}$$

- **Prediction rule**:
  - If **p > 0.5 → class 1**
  - Else → class 0
- **Uses**: Fraud detection, Medical diagnosis, Customer churn, Email spam
- **Code**:

```
from sklearn.linear_model

import LogisticRegression

model = LogisticRegression()
```

Decision Tree

- Makes decisions like a **flowchart** (if–else questions)
- **Root node** → first question
- **Branches** → answers (Yes/No)
- **Leaf nodes** → final prediction (Class 0/1, Yes/No)

Is it Sunny?

/ \

Yes No → Play = No

|

Is Temperature Hot?

/ \

Yes No

| |

No Yes

Code:

```
from sklearn.tree
 import DecisionTreeClassifier
model = DecisionTreeClassifier()
```

Random Forest

- Ensemble of **many decision trees**
- Uses **random rows + random features** for each tree
- **Classification** → majority vote
- **Regression** → average of outputs
- **Advantages**: Accurate, reduces overfitting, works with large datasets
- **Uses**: Fraud detection, Loan approval, Medical prediction, Stock/Price prediction
- **Code**:

```
from sklearn.ensemble
import RandomForestClassifier
rf = RandomForestClassifier()
```

KNN – K-Nearest Neighbors

- KNN is a **supervised machine learning algorithm**.
- It can be used for **classification** (class labels) or **regression** (numbers).
- It predicts the output of a **new data point** based on **the closest K neighbors** from the training data

How KNN Works

- Choose a value of **K** (number of neighbors).
- Calculate the **distance** between the new point and all training points (common: Euclidean distance).
- Pick the **K nearest points**.
- For **classification**: Take the **majority class** among neighbors.
- For **regression**: Take the **average value** of neighbors.
-

For two points $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

  - $d \rightarrow$ distance between the points
  - $x_1, y_1 \rightarrow$ coordinates of first point
  - $x_2, y_2 \rightarrow$ coordinates of second point
-

K-Means Clustering

It is an **unsupervised machine learning algorithm** that groups data into **K clusters**.

- Each cluster has a **centroid** (mean of points in the cluster).
- The algorithm **minimizes the distance** between points and their assigned cluster centroids.

Steps

1. Initialize **K centroids** ($\mu_1, \mu_2, \ldots, \mu_K$)
2. Assign each point x_i to nearest centroid ($\min ||x_i - \mu_j||^2$)
3. Recalculate centroid:

$$\mu_j = \frac{1}{|C_j|} \sum_{x_i \in C_j} x_i$$

4. Repeat until centroids **don't change**

Linear Regression

- A **supervised machine learning algorithm**.
- Used for **predicting a continuous numerical value** (regression problem).
- Predicts output y from input x using a **linear relationship**.
- Formula:

  Y=mx+c

  m → slope, c → intercept

1. IN / NOT IN

IN

Used to match a value with **multiple values**.

Syntax:

SELECT column_name

FROM table_name

WHERE column_name IN (value1, value2, value3);

Example:

SELECT name

FROM students

WHERE city IN ('Delhi', 'Mumbai', 'Chennai');

Meaning → students from **either** Delhi, Mumbai, or Chennai.

NOT IN

Used to exclude multiple values.

Example:

SELECT name

FROM students

WHERE city NOT IN ('Delhi', 'Mumbai');

Meaning → show students who are **not** from Delhi or Mumbai.

---

2. JOINS

Joins are used to combine rows from two tables based on a common column.

Suppose:

Table 1: Students

std_id name address

Table 2: Subjects

| subject_id | std_id | subject_name |

A. INNER JOIN

Returns only those rows where **matching data exists in both tables**.

Example:

SELECT students.std_id, students.name, subjects.subject_name

FROM students

INNER JOIN subjects

ON students.std_id = subjects.std_id;

Shows only students who have subjects.

B. LEFT JOIN (LEFT OUTER JOIN)

Returns **all rows from the left table** + matching rows from right.
If no match → NULL.

Example:

SELECT students.std_id, students.name, subjects.subject_name

FROM students

LEFT JOIN subjects

ON students.std_id = subjects.std_id;

Shows *all students*, even those who don't have subjects.

C. RIGHT JOIN (RIGHT OUTER JOIN)

Returns **all rows from the right table** + matching from left.

Example:

SELECT students.name, subjects.subject_name

FROM students

RIGHT JOIN subjects

ON students.std_id = subjects.std_id;

Shows *all subjects*, even if some subjects are not assigned to any student.

D. FULL JOIN (FULL OUTER JOIN)

Returns **all rows from both tables**, matching or not.

Example:

SELECT *

FROM students

FULL OUTER JOIN subjects

ON students.std_id = subjects.std_id;

Every student + every subject, match or no match.

---

3. GROUP BY

Used to group rows that have the **same values**.

Example: Count students in each city

SELECT city, COUNT(*)

FROM students

GROUP BY city;

Output example:

city	count
Delhi	10
Mumbai	8

---

4. HAVING Clause

- WHERE → used before grouping (filters **rows**)
- HAVING → used after grouping (filters **groups**)

Example: Show cities having more than 5 students

SELECT city, COUNT(*)

FROM students

GROUP BY city

HAVING COUNT(*) > 5;

HAVING applies on **aggregate functions** like COUNT(), SUM(), AVG().

YOLO (You Only Look Once)

Introduction

YOLO, which stands for **You Only Look Once**, is one of the fastest and most popular **real-time object detection algorithms**.
The name describes the core idea — the model looks at the image **only once** in a single forward pass and detects all objects instantly.

Because of its high speed and accuracy, YOLO is widely used in:

- CCTV surveillance
- Traffic and vehicle detection
- Self-driving cars
- Robotics and drones
- Medical imaging
- Gaming and AR/VR
- Industrial automation

What is Object Detection?

Object detection is a computer vision task in which an algorithm:

1. **Identifies** which objects are present in an image (cat, car, person, etc.)
2. **Locates** them using bounding boxes

How YOLO Works

YOLO treats object detection as a **single regression problem** — it directly predicts:

- Object class
- Bounding box coordinates
- Confidence score

It divides the input image into an **S × S grid**.
Each grid cell predicts:

- The probability of an object
- The object's class
- The bounding box (x, y, width, height)

Because the entire image is processed at once, YOLO becomes extremely fast.

Key Features of YOLO

1. Real-Time Speed

YOLO is famous for its speed. It can detect objects at **45+ FPS** (frames per second), while the lighter versions (YOLO-tiny) can go above **150 FPS**.

2. End-to-End Single Network

Unlike old models that required multiple networks, YOLO uses just **one neural network** for detection.

3. High Accuracy

YOLO gives very accurate predictions in real time. Later versions like YOLOv5 and YOLOv8 have reached extremely high precision.

4. Generalization

YOLO sees the entire image at once, so it understands context better and makes fewer mistakes like false positives.

YOLO Architecture (Simple Explanation)

A typical YOLO model contains:

1. **Input Image** (e.g., 416×416 or 640×640)
2. **CNN Backbone**
    - Extracts features
    - Examples of backbones: Darknet53, CSPDarknet, MobileNet
3. **Detection Head**
    - Predicts bounding boxes, classes, confidence scores
4. **Output**
    - List of detected objects with box coordinates and labels

---

Advantages of YOLO

- Extremely fast
- Works in real time
- High accuracy
- Simple end-to-end architecture
- Good for embedded devices and mobile
- Can detect multiple objects at the same time

Limitations

- Struggles with very small objects
- Sometimes misses objects that overlap heavily
- Accuracy depends on good training data

Applications of YOLO

✓ Self-Driving Cars

Detects pedestrians, traffic lights, vehicles in real time.

✓ Security & Surveillance

Used in smart CCTV systems to detect breaking rules or suspicious activities.

✓ Medical Imaging

Detects tumors, cells, and abnormalities.

✓ Retail & Industry

Counts products, inspects defects on assembly lines.

✓ Agriculture

Detects plant diseases, animals, crop quality.

✓ Robotics & Drones

Real-time navigation, object tracking, and obstacle detection.

Python Concepts

1. strip()

Removes spaces from both sides of the string.

" hello ".strip() Result → "hello"

2. split()

Breaks the string into parts and returns a list.

"a b c".split()

Result → ['a', 'b', 'c']

---

3. join()

Joins a list of words/letters into one string using a separator.

"-".join(["a", "b", "c"])

Result → "a-b-c"

---

4. replace()

Changes a part of the string with something else.

"hello world".replace("world", "Python")

Result → "hello Python"

---

5. upper()

Changes the whole string to **UPPERCASE**.

"hello".upper()

Result → "HELLO"

---

6. lower()

Changes the whole string to **lowercase**.

"HELLO".lower()

Result → "hello"

---

7. startswith()

Checks if the string **starts** with a word/letter.
Returns **True** or **False**.

"Python".startswith("Py")

Result → True

---

8. endswith()

Checks if the string **ends** with a word/letter.
Returns **True** or **False**.

"hello.txt".endswith(".txt")

Result → True

---

9. find()

Finds the position (index) of a word/letter.
If not found → returns **-1**.

"hello".find("l")

Result → 2 (because index starts from 0)

---

10. isdigit()

Checks if the string has **only numbers**.

"123".isdigit()

Result → True

---

11. isalpha()

Checks if the string has **only letters**.

"abc".isalpha()

Result → True

LIST METHODS

1. append()

Adds **one item** at the **end** of the list.

l = [1, 2, 3]

l.append(4)

print(l)

Result → [1, 2, 3, 4]

---

2. extend()

Adds **multiple items** (another list) at the end.

l = [1, 2, 3]

l.extend([4, 5])

print(l)

Result → [1, 2, 3, 4, 5]

---

3. insert()

Adds an item at a **specific position**.

```
l = [10, 20, 30]

l.insert(1, 15)  # position 1, value 15

print(l)
```

Result → [10, 15, 20, 30]

---

4. remove()

Removes the **first occurrence** of the given item.

```
l = [1, 2, 3, 2]

l.remove(2)

print(l)
```

Result → [1, 3, 2] (first 2 removed)

---

5. pop()

Removes an item by **index** and returns it.
(Default removes last item)

```
l = [10, 20, 30]

l.pop()
```

Result → removes 30

```
l.pop(1)
```

Result → removes item at index 1 (20)

---

6. index()

Gives the **position** of an item.

l = ["a", "b", "c"]

l.index("b")

Result → 1

---

7. count()

Counts **how many times** an item appears.

l = [1, 2, 2, 3]

l.count(2)

Result → 2

---

8. sort()

Sorts the list in **ascending order**.

l = [3, 1, 2]

l.sort()

print(l)

Result → [1, 2, 3]

---

9. reverse()

Reverses the list order.

l = [1, 2, 3]

l.reverse()

print(l)

Result → [3, 2, 1]

---

1. keys()

Gives all **keys** of the dictionary.

d = {"name": "Avi", "age": 20}

print(d.keys())

Result → dict_keys(['name', 'age'])

---

2. values()

Gives all **values** of the dictionary.

d.values()

Result → dict_values(['Avi', 20])

---

3. items()

Gives **key–value pairs** together.

d.items()

Result → dict_items([('name', 'Avi'), ('age', 20)])

---

4. get()

Returns the **value** of a key.
(No error if key doesn't exist)

d.get("name")

Result → "Avi"

---

5. update()

Adds or changes key–value pairs.

d.update({"city": "Delhi"})

print(d)

Result → {'name': 'Avi', 'age': 20, 'city': 'Delhi'}

---

6. pop()

Removes a key and returns its value.

d.pop("age")

Result → removes "age": 20

---

7. popitem()

Removes the **last added** key–value pair.

d.popitem()

---

8. clear()

Removes **everything** from the dictionary.

d.clear()

print(d)

Result → {} (empty dictionary)

---

9. fromkeys()

Creates a new dictionary using given keys.

keys = ["a", "b", "c"]

new_d = dict.fromkeys(keys, 0)

print(new_d)

Result → {'a': 0, 'b': 0, 'c': 0}

---

10. setdefault()

Returns the value of a key.
If key doesn't exist, it **adds** the key with a default value.

d.setdefault("country", "India")

Result → adds "country": "India"

DICTIONARY METHODS

1. get()

Used to get the value of a key.
If the key doesn't exist → it returns None (no error).

Example:

d = {"name": "Avi", "age": 20}

print(d.get("name"))

Output:

Avi

---

2. keys()

Returns all keys in the dictionary.

Example:

d.keys()

Output:

dict_keys(['name', 'age'])

---

3. values()

Returns all values in the dictionary.

Example:

d.values()

Output:

dict_values(['Avi', 20])

---

4. items()

Returns key–value pairs as tuples.

Example:

d.items()

Output:

dict_items([('name', 'Avi'), ('age', 20)])

---

5. update()

Adds new key–value pairs OR updates existing ones.

Example:

d.update({"city": "Delhi"})

Result:

{'name': 'Avi', 'age': 20, 'city': 'Delhi'}

---

6. pop()

Removes a key and returns its value.

Example:

d.pop("age")

Result:

20

Dictionary now becomes:

{'name': 'Avi'}

---

7. popitem()

Removes the last added key–value pair.

Example:

d.popitem()

If dictionary was:

{'name': 'Avi', 'age': 20}

After popitem():

('age', 20)

Dictionary becomes:

{'name': 'Avi'}

Set Methods:

- add() → Adds an element to the set.
- remove() → Removes a specific element (gives error if element not present).
- discard() → Removes a specific element if it exists (no error if element not present).
- pop() → Removes and returns a random element from the set.
- clear() → Removes all elements from the set.
- union() → Returns a new set with all elements from both sets.
- intersection() → Returns a new set with elements common to both sets.
- difference() → Returns a new set with elements in one set but not in the other.

General-Purpose Functions:

- len() → Returns the number of items in a sequence (like string, list, tuple, etc.).
- range() → Generates a sequence of numbers (often used in loops).
- print() → Displays output on the screen.
- type() → Returns the data type of a variable or value.
- id() → Returns the unique identity (memory address) of an object.
- sorted() → Returns a new sorted list from an iterable.
- enumerate() → Adds a counter to an iterable and returns it as pairs (index, value).
- zip() → Combines multiple iterables element-wise into tuples.

Conversion Functions

Used to convert one data type into another

Function	Description	Example
int()	Converts a value to integer	int("5") → 5
float()	Converts a value to float	float("3.14") → 3.14
str()	Converts a value to string	str(10) → "10"
list()	Converts iterable to list	list("abc") → ['a','b','c']
dict()	Converts sequence of tuples to dictionary	dict([('a',1),('b',2)]) → {'a':1,'b':2}
set()	Converts iterable to set	set([1,2,2,3]) → {1,2,3}
tuple()	Converts iterable to tuple	tuple([1,2,3]) → (1,2,3)

2. Mathematical Functions

Function	Description	Example
abs()	Returns the absolute value of a number.	abs(-5) → 5
sum()	Returns the sum of all items in an iterable.	sum([1,2,3]) → 6
min()	Returns the smallest item from an iterable.	min([4,1,9]) → 1
max()	Returns the largest item from an iterable.	max([4,1,9]) → 9
pow()	Raises a number to the power of another.	pow(2,3) → 8
round()	Rounds a number to the nearest integer or given decimals.	round(3.6) → 4

3. Functional Programming Tools

Function	Description
filter()	Selects items from an iterable that satisfy a condition.
map()	Applies a function to each item of an iterable.
reduce()	Combines all items of an iterable into a single value using a function.
lambda	Creates a small anonymous function in a single line.

4. Input & Output

Function	Description	Example
input()	Reads data from the user as a string.	name = input("Enter your name: ")

5 Class and Object Functions

Function	Description	Example
getattr()	Returns the value of a given attribute of an object.	getattr(obj,'name')
setattr()	Sets a value to an attribute of an object.	setattr(obj,'age',20)
hasattr()	Checks if an object has a specific attribute.	hasattr(obj,'name') → True
delattr()	Deletes an attribute from an object.	delattr(obj,'age')
isinstance()	Checks if an object is an instance of a class.	isinstance(5,int) → True
issubclass()	Checks if a class is a subclass of another class.	issubclass(bool,int) → True

Attribute vs Parameter:

- Attribute is a property of an object (obj.name), whereas a parameter is a value passed to a function (def greet(name):

• 6. Miscellaneous Functions

Function	Description	Example
`globals()`	Returns a dictionary of all global variables.	`globals()['__name__'] → "__main__"`
`locals()`	Returns a dictionary of all local variables.	`locals()`
`callable()`	Checks if an object can be called like a function.	`callable(print) → True`
`exec()`	Executes dynamic Python code.	`exec("a=5")`
`eval()`	Evaluates a Python expression and returns result.	`eval("2+3") → 5`

7. Exception Handling

Concept	Description	Example
try-except	Handles runtime errors and prevents program crash.	try:1/0 except ZeroDivisionError: print("Error")

Concept	Description	Example
else/elif	else and elif are conditional statements, not for error handling.	if x>5: print("Yes") elif x==5: print("Equal") else: print("No")

8. Memory and Object Management

Function	Description	Example
del()	Deletes a variable or object from memory.	a=5; del a
gc.collect()	Forces garbage collection to free unused memory.	import gc; gc.collect()

9. Iterables

Function	Description	Example
iter()	Creates an iterator from an iterable.	it = iter([1,2,3])
next()	Returns the next element from an iterator.	next(it) → 1

Key Features of Python

Easy to Learn and Use

Interpreted Language

Dynamically Typed

High-Level Language

Object-Oriented Programming (OOP)

Large Standard Library

Strong Community Support

Portable

Python Data Types

Basic Data Types

- **int** – Integer numbers (e.g., 10, -3)
- **float** – Decimal numbers (e.g., 3.14, -2.5)
- **str** – Text/characters (e.g., "Hello")
- **bool** – True or False

Collection Data Types

- **list** – Ordered, changeable collection (e.g., [1, 2, 3])
- **tuple** – Ordered, unchangeable collection (e.g., (1, 2, 3))
- **set** – Unordered, unique items (e.g., {1, 2, 3})
- **dict** – Key-value pairs (e.g., {"name": "Ram", "age": 20})

Advanced Types

- **NoneType** – Represents 'no value' (None)
- **complex** – Complex numbers (e.g., 2+3j)
- **bytes / bytearray** – Binary data
- **range** – Sequence of numbers (e.g., range(1,5))

PEP 8 =

Python's rulebook for writing clean, readable, and standard code.

Mutable Objects

You can modify the object without creating a new one.
You can add, remove, or replace elements.

Example: (list – mutable)

```
a = [1, 2, 3]

a[0] = 10   # changed the list

print(a)     # [10, 2, 3]
```

Immutable Objects

Once created, you **cannot** change them.
If you try to modify, Python creates a **new object**.

Example: (string – immutable)

Indentation in Python

Indentation means giving spaces at the beginning of a line.
Python uses indentation to **define blocks of code** (loops, functions, if statements, etc.).
Instead of { } like other languages, Python uses **indentation level** to show which statements belong together.

How Python Executes a Program

Flow Diagram

Your Code (.py)

↓

Bytecode (.pyc)

↓

Python Virtual Machine (PVM)

↓

Output

Namespace = A container/directory where names are mapped to objects.

How to Make Dictionaries in Python

Python provides **4 common ways** to create a dictionary.

---

1. Using Curly Braces { }

```
student = {
   "name": "Aman",
   "age": 20,
```

```
    "city": "Delhi"
}
```

---

2. Using the dict() Constructor

```
student = dict(name="Aman", age=20, city="Delhi")
```

---

3. Creating an Empty Dictionary and Adding Items

```
student = {}
student["name"] = "Aman"
student["age"] = 20
student["city"] = "Delhi"
```

---

4. Using List of Tuples / Pairs

```
student = dict([("name", "Aman"), ("age", 20), ("city", "Delhi")])
```

Remove duplicates from list

Using set()

```
my_list = [1, 2, 2, 3, 3, 4]
unique_list = list(set(my_list))
print(unique_list)  # [1, 2, 3, 4]
```

List Comprehension

List comprehension is a concise way to **create lists** using a single line of code.
It combines **loops and conditionals** into a **short, readable expression**.

Feature	Shallow Copy	Deep Copy
Definition	Creates a new object, but **nested objects reference the same memory**.	Creates a new object and **recursively copies all nested objects**.

Effect on Original	Changes in nested objects **affect the original**.	Changes in copy **do not affect the original**.

Slicing in Python

Slicing is a technique to **extract a portion of a sequence** (like a list, tuple, or string) using **start, stop, and step** indices.

Syntax

sequence[start:stop:step]

1. Slice a List

numbers = [0, 1, 2, 3, 4, 5]

print(numbers[1:4]) # [1, 2, 3]

Frozenset

A **frozenset** is an **immutable version of a set**.
Once created, its elements **cannot be changed**, but you can perform **set operations** like union, intersection, and difference.

*args and **kwargs

In Python, functions can accept **variable number of arguments** using:

1. *args → **Non-keyword (positional) variable arguments**
2. **kwargs → **Keyword (named) variable arguments**

---

1. *args (Variable Positional Arguments)

- Allows passing **any number of positional arguments** to a function.
- Arguments are received as a **tuple**.

Syntax

```
def func(*args):
    for arg in args:
        print(arg)
```

Example

```
def add(*args):
    return sum(args)

print(add(1, 2, 3))   # 6
print(add(5, 10))     # 15
```

---

2. **kwargs (Variable Keyword Arguments)

- Allows passing **any number of named arguments** to a function.
- Arguments are received as a **dictionary**.

Syntax

```
def func(**kwargs):
    for key, value in kwargs.items():
        print(key, value)
```

Decorators

A **decorator** is a **function that takes another function as input, adds extra functionality to it, and returns it**.
It allows modifying the behavior of a function **without changing its code**.

Closures

A **closure** is a function that **remembers the values from its enclosing scope**, even if the outer function has finished execution.

- **Inner function "closes over" variables of outer function**.
- Useful for **data hiding** and **decorators**.

Key Points

Definition Inner function that remembers outer function's variables

Inheritance

Inheritance is a mechanism in Python where **a class (child/subclass) acquires properties and methods from another class (parent/superclass)**.

- Promotes **code reusability**
- Allows creating **hierarchical relationships** between classes

Multiple vs Multilevel Inheritance

Feature	Multiple Inheritance	Multilevel Inheritance
Definition	A child class inherits from more than one parent class	A class inherits from a child class, forming a chain of inheritance
Number of Parents	Two or more parent classes	Only one parent class per level
Number of Levels	Single level	Multiple levels (grandparent → parent → child)
Syntax	class Child(Parent1, Parent2):	class Grandchild(Child):
Use Case	Combine functionality from multiple classes	Extend functionality gradually through levels

Polymorphism

Polymorphism means **"many forms"**.
In Python, it allows **objects of different classes to be treated in a similar way**, or **methods to behave differently based on context**.

Abstraction

Abstraction is the concept of **hiding the internal implementation details** of a class and showing only the **essential features** to the user.

- Focuses on **what an object does**, not **how it does it**.
- Achieved using **abstract classes** and **abstract methods** in Python.

Monkey Patching

Monkey Patching is the practice of **dynamically modifying or extending a class or module at runtime** without altering the original source code.

- Can be used to **add, change, or override methods or attributes** of existing classes or modules.

- Often used in **testing, debugging, or quick fixes**.

Syntax Error vs Runtime Error

Feature	Syntax Error	Runtime Error
Definition	Occurs when Python cannot parse the code due to incorrect syntax	Occurs while program is running due to illegal operations
When Detected	Before execution (during parsing)	During execution
Causes	Missing colon, wrong indentation, misspelled keywords	Division by zero, accessing undefined variable, file not found
Program Stops?	Yes, code won't run	Yes, only if exception is not handled
Example	print("Hello" → missing closing parenthesis	x = 5 / 0 → division by zero

What is PIP?

PIP stands for **"Python Package Installer"**.
It is a tool used to **install, upgrade, and manage Python packages and libraries** from the **Python Package Index (PyPI)**.

File Reading Methods in Python

Method	Description	Return Type	Example
read(size)	Reads the entire file or up to size characters	String	f.read() or f.read(10)
readline()	Reads one line from the file at a time	String	f.readline()
readlines()	Reads all lines and returns them as a list of strings	List	f.readlines()

Deleting a File in Python

To delete a file in Python, you can use the os module.

Steps

1. Import os module
2. Use os.remove() or os.unlink() to delete a file

NumPy Array vs Python List

Feature	Python List	NumPy Array
Type of Elements	Can store **different data types**	Stores **elements of the same type**
Performance	Slower for large data	**Faster** due to vectorized operations
Memory	More memory per element	**Less memory**, more efficient
Operations	Element-wise operations require **loops**	Supports **vectorized operations** (a + b, a * b)
Functions	Limited built-in mathematical functions	Rich set of **mathematical, statistical, and linear algebra functions**
Multidimensional	Can create nested lists, but **manual handling needed**	Supports **ndarrays** natively (1D, 2D, 3D, …)
Broadcasting	Not supported	Supported (operate on arrays of different shapes)

Broadcasting

Automatic alignment of arrays of different shapes in NumPy for arithmetic operations is called Broadcasting.

Vectorization in NumPy

Vectorization means performing **operations on entire arrays** (vectors/matrices) **without using explicit loops**.

GIL (Global Interpreter Lock)

GIL stands for **Global Interpreter Lock**.
It is a mutex (mutual exclusion) lock in CPython (the standard Python implementation) that allows only one thread to execute Python bytecode at a time, even on multi-core processors.

One-Hot Encoding

One-Hot Encoding is a technique to **convert categorical variables into numerical form** so that machine learning models can use them.

- Each **category becomes a new column**
- A **1** is placed in the column corresponding to the category, **0** elsewhere

Regex in Python

- **Regular Expressions (Regex)** are **patterns used to match, search, and manipulate strings**.
  In Python, regex operations are performed using the **re module**.

JAVA BASICS

Main Features of Java

- Platform independent (WORA)
- Object Oriented
- Secure
- Robust
- Multithreaded
- High Performance
- Portable

JDK vs JRE vs JVM

- **JVM** → Runs Java bytecode
- **JRE** → JVM + Libraries (to run programs)
- **JDK** → JRE + Development tools (to write programs)

Access Modifiers

- public → accessible everywhere
- protected → same package + subclass
- default → same package only
- private → same class only

Constructor

- Special method to initialize object
- Same name as class
- No return type
- Called automatically

Can Constructor be Private?

- **Yes**
- Used in **Singleton class**
- Prevents object creation outside class

Abstraction vs Encapsulation

- **Abstraction** → hiding implementation (what to do)
- **Encapsulation** → binding data + methods (how to protect)

Primitive Data Types

- byte, short, int, long
- float, double
- char
- boolean

Static vs Non-static Variables

- **static** → class level, shared
- **non-static** → object level, separate copy

---

final vs finally vs finalize

- final → constant / cannot override / cannot inherit
- finally → always executes in exception handling
- finalize() → called by GC before object removal

---

this vs super

- this → current class object
- super → parent class object

String vs StringBuffer vs StringBuilder

- **String** → immutable
- **StringBuffer** → mutable + thread safe
- **StringBuilder** → mutable + fast (not thread safe)

Why String is Immutable

- Security
- Thread safety

Array vs ArrayList

- Array → fixed size
- ArrayList → dynamic size
- Array → faster
- ArrayList → flexible

Collection Framework

- Used to store & manipulate objects
- Interfaces → List, Set, Queue, Map
- Classes → ArrayList, LinkedList, HashSet, HashMap

ArrayList vs LinkedList

- ArrayList → fast access
- LinkedList → fast insertion/deletion
- ArrayList → uses array
- LinkedList → uses nodes

How HashMap Works

- Stores data as **key-value pairs**
- Uses **hashing**
- Allows **one null key**
- No duplicate keys

Checked vs Unchecked Exception

- **Checked** → compile time (IOException)
- **Unchecked** → runtime (NullPointerException)

---

throw vs throws

- throw → throws exception manually
- throws → declares exception in method

Multiple Catch Blocks

- **Yes allowed**
- Order → child first, parent last

🔷 MULTITHREADING

Process vs Thread

- Process → program in execution
- Thread → smallest unit of process
- Thread shares memory

Multithreading

- Multiple threads running simultaneously
- Improves performance
- Better CPU utilization

Deadlock

- Threads waiting for each other forever
- Conditions → Mutual Exclusion, Hold & Wait, No Preemption, Circular Wait

---

volatile Keyword

- Ensures visibility of variable
- Does NOT ensure atomicity
- Used in multithreading

Lambda Expression

- Short way to implement functional interface
- Syntax → (args) -> { body }
- Checked exception must be handled or declared

---

map() vs flatMap()

- map → one-to-one transformation
- flatMap → flattens nested structure

Inner Classes

- Non-static inner
- Static nested
- Local inner
- Anonymous inner

Diamond Problem

- Ambiguity in multiple inheritance
- Java avoids it using **interfaces**
- Must override method

Method Hiding

- Happens with **static methods**
- Reference type decides method
- Compile time binding

Software Development Life Cycle (SDLC)

1. SDLC Overview

- **SDLC** is a structured process to develop software efficiently.
- **Phases:**
    1. Requirement Analysis
    2. System / Functional Design
    3. Development / Implementation
    4. Testing
    5. Deployment
    6. Maintenance

---

2. Roles & Responsibilities

Business / Functional Side

- **Business Analyst / Functional Consultant**
    - Collects business requirements
    - Creates **Functional Specification (FS)**
    - Maps business needs to software functionality
- **Product Manager**
    - Defines product vision, roadmap, and features
    - Focus: *What to build and why*
- **Project Manager**
    - Plans project timelines, resources, and budget
    - Focus: *How & when to deliver*

Development Team

- **Developer / Programmer**
    - Writes code based on **Functional & Technical Specs**

    - Creates **Technical Specification Document (TSD)**
    - Performs **Unit Testing**
- **Documentation**
    - Maintains all project-related documents: BRD, FS, TSD, Test Cases, User Manuals

---

Testing / QA Team

- Creates **Test Cases / Test Scripts** based on FS
- Performs **Testing**: Unit, Integration, System, Regression
- Logs defects and verifies bug fixes

Administration / Operations Team

- Maintains **network, applications, and databases**
- Handles **deployment, configuration, monitoring, and backups**

Security Team

- Ensures **data, network, and application security**
- Performs **security monitoring, access control, and compliance audits**

Support Team

- Provides **post-deployment user support**
- Troubleshoots issues and escalates to developers or admin team
- Maintains **knowledge base / FAQs**

Sales & Marketing Team

- Promotes the product and generates revenue
- Collects **customer feedback**
- Coordinates with Product Manager for improvements

Human Resource (HR) Team

- Manages **recruitment, onboarding, training, payroll, policies, and employee relations**
- Supports project teams by providing skilled resources

Waterfall Model (Software Development)

The **Waterfall Model** is one of the **earliest and simplest SDLC models**. It is **linear and sequential**, meaning each phase must be **completed before the next one starts**.

Key Characteristics

- Sequential flow: Each phase depends on the deliverables of the previous phase
- No overlapping of phases
- Documentation-heavy
- Easy to understand and manage

Phases of Waterfall Model

1. **Requirement Analysis**
    - Gather business and system requirements
    - Create **Requirement Specification Document (SRS)**
2. **System / Software Design**
    - Translate requirements into **architecture and technical design**
    - Prepare design documents
3. **Implementation / Coding**
    - Developers write code based on design documents
    - Unit testing may also be done
4. **Testing**
    - QA team performs **integration and system testing**
    - Identify and fix defects

5. **Deployment / Installation**
    - Software is delivered and installed at the user site
6. **Maintenance**
    - Fix post-deployment issues
    - Provide updates and enhancements

Advantages

- Simple and easy to understand
- Clear milestones and documentation
- Works well for **small projects with fixed requirements**

Disadvantages

- Inflexible to requirement changes
- Late discovery of defects (testing comes after coding)
- Not suitable for **large or complex projects**

Agile Model (Software Development)

The **Agile Model** is an **iterative and incremental approach** to software development. Unlike Waterfall, it **focuses on flexibility, collaboration, and customer feedback**.

Key Characteristics

- Iterative development in **small cycles called sprints**
- Continuous collaboration with **stakeholders and customers**
- Frequent delivery of **working software**
- Emphasis on **responding to change** rather than following a fixed plan

---

Phases / Process of Agile

1. **Requirement Gathering** (High-level)
    - Collect features for the current iteration (sprint)

2. **Planning**
    - Define tasks for the sprint
    - Estimate effort and resources
3. **Design & Development**
    - Develop features in small increments
    - Unit testing by developers
4. **Testing**
    - QA tests each increment
    - Continuous integration ensures working software
5. **Review / Feedback**
    - Present working software to stakeholders
    - Incorporate feedback in next sprint
6. **Deployment**
    - Deliver increments to users frequently
    - Updates done iteratively

Advantages

- Flexible and adaptive to changing requirements
- Early delivery of working software
- Close collaboration with customers
- Reduces risk of project failure

Disadvantages

- Requires active stakeholder involvement
- Less predictable scope and timeline
- Needs skilled and motivated team

A* (A-Star) Search Algorithm

A* is an **informed search algorithm** used to find the **shortest path** between a start node and a goal node. It is widely used in **AI, path-finding, and graph traversal** problems.

Working of A* Algorithm

1. Add the **start node** to the OPEN list.
2. Select the node with **minimum f(n)** from OPEN.
3. If it is the **goal node**, stop.
4. Move the node to the **CLOSED list**.
5. Expand the node and calculate **g, h, and f** for its neighbors.
6. Add unexplored neighbors to OPEN.
7. Repeat until goal is found or OPEN is empty.

Heuristic Function (h(n))

A heuristic must be:

- **Admissible** → never overestimates the true cost
- **Consistent** → satisfies triangle inequality

---

1. Bubble Sort

Idea

Repeatedly compares adjacent elements and swaps them if they are in the wrong order. Largest element moves to the end in each pass.

Algorithm

1. Repeat for all elements:
2. Compare adjacent elements.
3. Swap if left > right.
4. Stop when no swaps occur.

Example

Array: 5 1 4 2
Pass 1 → 1 4 2 5
Pass 2 → 1 2 4 5

2. Selection Sort

Idea

Selects the smallest element and places it at the beginning.

Algorithm

1. Find the minimum element.
2. Swap with the first unsorted position.
3. Repeat for remaining array.

Example

64 25 12 22 11
After 1st pass → 11 25 12 22 64

Complexity

- Best / Average / Worst: **$O(n^2)$**

---

3. Insertion Sort

Idea

Builds sorted list one element at a time by inserting elements in the correct position.

Algorithm

1. Take next element.
2. Shift larger elements to the right.
3. Insert element at correct position.

Example

8 3 5 2
Stepwise → 3 8 5 2 → 3 5 8 2 → 2 3 5 8

Complexity

- Best: **$O(n)$**
- Average: **$O(n^2)$**
- Worst: **$O(n^2)$**

---

4. Merge Sort

Idea

Divide-and-conquer algorithm. Splits array into halves, sorts them, and merges.

Algorithm

1. Divide array into two halves.
2. Recursively sort halves.
3. Merge sorted halves.

Example

38 27 43 3
Split → [38 27] [43 3]
Merge → [3 27 38 43]

Complexity

- Best / Average / Worst: **O(n log n)**

---

5. Quick Sort

Idea

Divide-and-conquer algorithm using a **pivot**. Elements smaller than pivot go left, larger go right.

Algorithm

1. Choose a pivot.
2. Partition array.
3. Recursively apply quick sort.

Example

10 80 30 90 40
Pivot = 40 → [10 30] 40 [80 90]

Complexity

- Best: **O(n log n)**

Code of gc.collect

```
import gc

class Demo:
   def __init__(self, name):
      self.name = name
      self.ref = None

   def __del__(self):
      print(f"Object {self.name} is garbage collected")

# Enable garbage collection
gc.enable()

# Create objects
obj1 = Demo("A")
obj2 = Demo("B")

# Create circular reference
obj1.ref = obj2
obj2.ref = obj1

# Remove references
obj1 = None
obj2 = None

# Force garbage collection
print("Collecting garbage...")
collected_objects = gc.collect()

print(f"Number of unreachable objects collected: {collected_objects}")
```