Smart Stock Inventory For Retail Stores

Day1:

Overview of the project

Received an introduction to the internship structure, rules, and expectations. Understood the overall flow of tasks and learning outcomes.

We discussed the main problems in retail inventory management such as overstock and out-of-stock situations.

DAY2:

We discussed and finalized the technical components required for building the Smart Stock Inventory Optimization System.

Frontend: HTML, CSS, JavaScript for creating the user interface and dashboard.

Backend: Python for implementing the core logic, stock calculations, alerts, and prediction models.

Database: MySQL / SQL / MongoDB for storing product details, sales records, supplier info, and stock levels.

NumPy: A Python library used in the backend to perform numerical calculations, mathematical operations, and forecasting formulas efficiently.

Pandas: A data handling library used for loading, cleaning, analyzing, and processing datasets such as product data and sales records.

Python Format Strings: Formatting feature in Python used to print structured, clean, and readable outputs in the backend system.

API Basics: Understanding how different systems communicate APIs such as FastAPI and Flask APIs, used if the project needs frontend-backend data exchange.

DAY3:

Flask API: Flask is a lightweight Python web framework used to create simple APIs.

FastAPI: A modern and very fast Python framework supporting high performance, automatic documentation, and faster request handling.

Difference Between Flask and FastAPI:

- Flask: Simple, lightweight, beginner-friendly.

- FastAPI: Faster, modern, supports automatic documentation.

Postman: A tool used to test APIs by sending requests and checking responses.

DAY4:

GitHub Overview: GitHub is an online platform that stores Git repositories.

Repository (Repo): A folder where your project files and their history are stored.

Local Repository: A Git repository stored on your computer.

Cloning: Copying a GitHub repository to your local system.

Committing: Saving code changes with a message.

Pushing: Sending local commits to the GitHub remote repository.

Pulling: Getting the latest changes from GitHub.

Merge: Combining changes from one branch into another.

Fork: Creating your own copy of another repository.

Pull Request (PR): Request to merge changes into the main project.

Git Commands:

- git init

- git clone

- git status

- git add .

- git commit -m "message"

- git push origin main

- git pull

- git branch

Day5:

Introduction to SQL

SQL (Structured Query Language) is used to interact with databases (CRUD operations).

Create, Read, Update, Delete.

Database: Collection of tables.

Table: Rows and columns.

Primary Key: Unique identifier.

Foreign Key: Links tables.

Artificial Intelligence and Machine Learning basics were also covered.

Day6:

Normalization: Scaling or adjusting data to fit within a standard format.

In databases, normalization reduces redundancy and improves data integrity.
Explained 1NF, 2NF, 3NF, and BCNF.

Day7:

GitHub repository setup and project review conducted.

Day8:

Milestone 1 conducted – Initial module verification and basic functionality testing completed.

Day9:

Milestone 1 continuation – Validation of workflow, correction of minor issues, and preparation for next milestone.

Agent AI Workflow

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

Reactive Agent

A Reactive Agent is an agent that:

1. Responds immediately to the current situation.
2. 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:

1. Acts to achieve a specific goal
2. Can plan its actions instead of just reacting

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

1. Each agent is independent
2. Agents can communicate, cooperate, or compete

How AI Agents Think, Plan, and Work

1. Thinking (Reasoning Phase)

Definition:

AI analyzes the input, understands the user's goal, checks context, and interprets the task.

What happens:

- Identifies the problem
- Understands instructions
- Retrieves relevant knowledge
- Predicts what information is needed

Purpose:

To understand *what needs to be done*.

2. Planning (Decision-Making Phase)

Definition:

AI breaks the task into logical steps and chooses a strategy to solve it.

What happens:

- Generates a step-by-step plan
- Decides which tools or functions to use
- Selects best path to reach the goal

- Organizes actions in sequence

Purpose:

To prepare *how to do the task*.

3. Acting (Execution Phase)

Definition:

AI performs the actions or executes the plan it created.

What happens:

- Calls APIs, tools, or functions
- Generates answers or outputs
- Runs code or performs tasks
- Interacts with user or environment

Purpose:

To *complete the task*.

4. Feedback Loop (Evaluation Phase)

Definition:

AI checks if the task is completed correctly.

What happens:

- Evaluates the result
- Checks for errors

- Decides if another step is required
- Asks follow-up questions if needed

Purpose:

To make the output *accurate and complete*.

Pip install openai

Operations of OpenAI Library

The **OpenAI Python library** provides several important operations that allow developers to use advanced AI models inside applications. After installing the library using `pip install openai`, we can perform the following operations:

1. Text Generation

The model generates human-like text from a given prompt.
It is used for chatbots, story writing, answering questions, and content creation.

```
response = client.chat.completions.create(
    model="gpt-4o-mini",
    messages=[{"role": "user", "content": "Write a story"}]
)
```

2. Chat Completions

This operation enables conversational AI.
The model responds in a dialogue format using message history, making it useful for building assistants and customer-support bots.

```
client.chat.completions.create(
    model="gpt-4o-mini",
    messages=[
```

```
    {"role": "user", "content": "What is AI?"}
  ])
```

3. Image Generation

OpenAI models can create high-quality images from text descriptions.
This is used for design, art creation, and visualization tasks.

```
client.images.generate(
  model="gpt-image-1",
  prompt="sunset over mountains"
)
```

4. Embeddings

Embeddings convert text into numerical vector representations.
They are used in search engines, recommendations, document similarity, and clustering.

```
client.embeddings.create(
  model="text-embedding-3-small",
  input="machine learning"
)
```

5. Audio Operations

OpenAI supports both speech-to-text (transcription) and text-to-speech operations.
These operations help in voice assistants, subtitles, and audio automation.

```
client.audio.transcriptions.create(
  model="gpt-4o-transcribe",
  file=open("audio.mp3", "rb")
)
```

6. File Handling

The library allows uploading, managing, and using files for training or analysis.
This is essential for fine-tuning and advanced applications.

```
client.files.create(file=open("data.jsonl", "rb"), purpose="fine-tune")
```

7. Fine-Tuning

Fine-tuning allows training custom versions of OpenAI models with your own dataset to improve performance on specific tasks.

```
client.fine_tuning.jobs.create(
  model="gpt-4o-mini",
  training_file="file-id"
)
```

8. Function / Tool Calling

The AI can automatically call functions or tools when needed, enabling intelligent automation and agent-like behavior.

```
{
 "type": "function",
 "function": {
   "name": "get_weather",
   "arguments": {"city": "Delhi"}
 }
}
```

Natural language processing:

Natural Language Processing (NLP) is a branch of Artificial Intelligence that helps computers understand, interpret, and generate human languag.

Sentiment analysis:

Sentiment Analysis is a Natural Language Processing (NLP) technique used to determine the emotional tone or opinion expressed in a piece of text.It identifies whether the text is Positive, Negative, or Neutral.

Difference between Natural language processing and sentiment analysis:

Definition: A field of AI that focuses on enabling computers to understand and process human language. A sub-task of NLP that identifies the emotional tone or opinion in text.

Purpose: To make machines understand, interpret, and generate human language.To find whether a text is positive, negative, or neutral.

Scope:Very broad (covers many language-related tasks).Narrow (focuses only on emotion or opinion).

Examples : Machine translation, chatbots, speech recognition, NER, text summarization. Customer review analysis, movie ratings, product feedback polarity.

Techniques: Used Tokenization, POS tagging, parsing, embeddings, transformers. Classification models, lexicons, polarity detection.

Output: Structured language understanding (entities, grammar, meaning).
Sentiment score or sentiment label.

Tokenization:

Tokenization is the process of splitting a large text into smaller units called tokens.

These tokens can be words, sentences, or even sub-words.

Simple Definition:

Tokenization means breaking text into small meaningful pieces.

Why Tokenization?

Computers cannot understand raw text directly.

Splitting text into tokens helps in further NLP tasks like stemming, POS tagging, sentiment analysis, etc.

Types of Tokenization

Word Tokenization: Splitting text into individual words.

Example:

Text: "I love Python"

Tokens → ["I", "love", "Python"]

Sentence Tokenization: Splitting text into sentences.

Example:"I love AI. It is interesting."

Sentences → ["I love AI.", "It is interesting."]

Sub-word Tokenization:

Breaking words into smaller parts (used in BERT, GPT).

Example:

"playing" → ["play", "ing"]

Prompt Engineering

What is Prompt Engineering?

- **Prompt engineering** is the process of designing, structuring, and refining the input (prompt) given to an AI model so that it generates accurate, relevant, and high-quality outputs.
- A **prompt** is the text or instruction you give to the model—such as a question, command, or request—that guides the model's response.
- Effective prompt engineering helps the AI understand the task clearly and produce better results.

Related Terms

- **Rate Limit:** The maximum number of requests allowed within a short period of time (e.g., per minute).
- **Quota:** The maximum total usage allowed over a longer period (e.g., per day or per month).

Logistic Regression

1. Definition

- **Logistic Regression** is a **supervised machine learning algorithm** used for **classification** problems.
- It predicts the **probability** of an outcome that can only be in **two categories** (0 or 1), such as:
    - Spam or Not Spam
    - Disease or No Disease
    - Pass or Fail

2. Why It Is Called “Regression”

- Although it is used for **classification**, the name “regression” is used because it predicts a **probability** using a regression-like equation.
- The output value is then converted into a class (0/1).

3. Sigmoid Function (Key Concept)

Logistic Regression uses the **sigmoid function** to convert any numeric output into a probability between **0 and 1**.

Sigmoid Formula:

$\sigma(z)=11+e-z|sigma(z) = |frac\{1\}\{1 + e^\{-z\}\}$σ(z)=1+e−z1

Where:

$z=b0+b1x1+b2x2+...+bnxnz = b\_0 + b\_1x\_1 + b\_2x\_2 + |ldots + b\_nx\_nz$=b0 +b1 x1 +b2 x2 +...+bn xn

4. Decision Boundary

- If predicted probability **≥ 0.5 → Class 1**
- If predicted probability **< 0.5 → Class 0**

5. Types of Logistic Regression

1. **Binary Logistic Regression** – Two classes (0/1)
2. **Multinomial Logistic Regression** – More than two classes
3. **Ordinal Logistic Regression** – Ordered categories

6. Cost Function

Instead of MSE, logistic regression uses **Log Loss**:

$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y_i \log(h_\theta(x_i)) + (1-y_i)\log(1-h_\theta(x_i))]$ $J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y_i \log(h_\theta(x_i)) + (1-y_i)\log(1-h_\theta(x_i))]$ J(θ)=−m1 i=1∑m [yi log(hθ (xi))+(1−yi)log(1−hθ (xi))]

7. Applications

- Email spam detection
- Fraud detection
- Disease diagnosis
- Customer churn prediction

• Decision Tree

Definition

A Decision Tree is a supervised learning algorithm that splits data into branches based on conditions, forming a tree-like structure.

Key Points

- Used for **classification** and **regression**.
- Splits data based on **features** using rules (e.g., age < 30?).
- Nodes represent decisions; leaves represent final outputs.
- Easy to understand and visualize.

How it Works

1. Select the best feature to split the data (using **Gini**, **Entropy**, or **MSE**).
2. Split the dataset.
3. Repeat recursively to build the tree.
4. Stop when leaf purity or depth limit is reached.

Advantages

- Simple to interpret.
- Works with numerical + categorical data.

- No need for feature scaling.

Random Forest

Definition

Random Forest is an ensemble learning algorithm that builds many decision trees and combines their results.

Key Points

- Uses **multiple trees**, not just one.Each tree is trained on a **random sample** of data (bootstrapping).
- Each tree uses a **random subset of features**.

How it Works

1. Create many decision trees.
2. Each tree predicts independently.
3. Final prediction:
    a. **Classification:** Majority voting
    b. **Regression:** Average of outputs

Advantages

- High accuracy.
- Reduces overfitting (compared to a single tree).
- Works well on large datasets.

Disadvantages

- Slower than a single tree.
- Harder to interpret.

K-Means Clustering

Definition

K-Means is an **unsupervised learning** algorithm used to group data into **K clusters** based on similarity.

Key Points

- No labels → algorithm discovers groups itself.
- Uses **distance** (usually Euclidean distance).

How it Works

1. Choose number of clusters **K**.
2. Randomly assign **K centroids**.
3. Assign each point to nearest centroid.
4. Recalculate new centroids.
5. Repeat until centroids stop changing.

Advantages

- Fast and simple.
- Works well when clusters are clearly separated.

Disadvantages

- Need to pre-define K.
- Does not work well with irregular-shaped clusters.
- Sensitive to outliers.

K-Nearest Neighbors (K-NN)

Definition

K-NN is a supervised learning algorithm that predicts labels based on the **K nearest data points**.

Key Points

- Works for **classification** and **regression**.
- Non-parametric (no training phase).
- Uses **distance measure** like Euclidean distance.

How it Works

1. Choose **K** (number of neighbors).
2. Calculate distance between test point and all training points.
3. Pick K closest points.
4. Final output:
    a. **Classification:** Majority vote among K neighbors.
    b. **Regression:** Average of values.

Advantages

- Simple and effective.
- Good for small datasets.
- No need for training.

Disadvantages

- Slow for large datasets.
- Requires scaling of data.
- Sensitive to noisy data.

Linear Regression

Definition

Linear Regression is a **supervised learning algorithm** used to predict a **continuous numerical value** by finding a linear relationship between input (X) and output (Y).

It assumes:

Y = mX + c

Where

m = slope
c = intercept

SQL (Structured Query Language)

Definition

SQL is a **standard language** used to **store, access, manage, and manipulate data** in relational databases.

It allows users to:

- Create databases and tables
- Insert, update, and delete data
- Retrieve data using queries
- Control user access and permissions

SQL Constraints

Used to apply rules on table data.

Constraint	Purpose
PRIMARY KEY	Uniquely identifies each row (not null + unique).
FOREIGN KEY	Links two tables; maintains referential integrity.
UNIQUE	Ensures all values in a column are unique.
NOT NULL	Field cannot be empty.
CHECK	Validates a condition.
DEFAULT	Provides a default value if none is given.

```
CREATE TABLE employees (
    id INT PRIMARY KEY,
    name VARCHAR(50) NOT NULL,
    age INT CHECK (age > 18),
    city VARCHAR(30) DEFAULT 'Hyderabad'
);
```

SQL Joins

Used to combine rows from multiple tables based on a related column.

Join Type	Description
INNER JOIN	Returns matching rows from both tables.
LEFT JOIN	Returns all rows from left table + matches from right.
RIGHT JOIN	Returns all rows from right table + matches from left.
FULL JOIN	Returns all rows from both tables.
CROSS JOIN	Cartesian product (all combinations).

Example

```
SELECT emp.name, dept.dept_name
FROM employees emp
INNER JOIN departments dept
ON emp.dept_id = dept.id;
```

3. SQL Aggregate Functions

Used to perform calculations.

Function	Meaning
COUNT()	Counts rows
SUM()	Adds values
AVG()	Average value
MAX()	Highest value
MIN()	Lowest value

Example

```
SELECT AVG(salary) FROM employees;
```

4. SQL String Functions

Used to manipulate text.

Function	Description
UPPER()	Converts text to uppercase
LOWER()	Converts text to lowercase
CONCAT()	Joins strings
LENGTH()	Number of characters
SUBSTRING()	Extracts part of a string

Example

```
SELECT UPPER(name) FROM students;
```

SQL Date Functions

Used to work with date/time values.

Function	Description
NOW()	Current date & time
CURDATE()	Current date
DATE_ADD()	Adds days/months/years
DATEDIFF()	Difference between dates

Example

```
SELECT DATEDIFF('2025-12-31', CURDATE());
```

File I/O METHODS:

1. open( ) — Opens a File

Used to open a file in a specific mode.

Common Modes

- `"r"` → Read
- `"w"` → Write (overwrites file)
- `"a"` → Append
- `"r+"` → Read + Write

Example

```
file = open("demo.txt", "r")
```

2. `read()` — Reads Entire File

Reads the **complete content** of a file as one string.

Example

```
file = open("demo.txt", "r")
data = file.read()
print(data)
file.close()
```

3. `readline()` — Reads One Line at a Time

Reads **only the next line**, including newline \n.

Example

```
file = open("demo.txt", "r")
line1 = file.readline()
print(line1)
file.close()
```

4. `readlines()` — Reads All Lines into a List

Each line becomes one element of a list.

Example

```
file = open("demo.txt", "r")
lines = file.readlines()
print(lines)
file.close()
```

Output example:

```
['Hello\n', 'How are you?\n', 'Welcome!']
```

5. `write()` — Writes a Single String to File

Used to write text.

Note: In `"w"` mode it **overwrites** the file.

Example

```
file = open("example.txt", "w")
file.write("Hello Pavani!\n")
file.write("Welcome to Python.")
file.close()
```

6. `writelines()` — Writes Multiple Lines

Takes a **list of strings** and writes them to a file.

Example

```
lines = ["Line 1\n", "Line 2\n", "Line 3\n"]

file = open("output.txt", "w")
file.writelines(lines)
file.close()
```

7. `close()` — Closes the File

Used to close a file and free system resources.

Example

```
file = open("demo.txt", "r")
file.close()
```

len() – Returns Length

Gives the number of items in a list, string, tuple, etc.

Example

```
print(len("hello"))
```

output:

```
5
```

range() – Generates a Sequence of Numbers

Mostly used in loops.

Example

```
for i in range(1, 5):
    print(i)
```

Output:

```
1
2
3
4
```

print() – Displays Output

Used to show text or variables.**Example**

```
name = "Pavani"
print("Hello", name)
```

Output: Hello Pavani

type() – Shows Data Type

Tells which datatype the value belongs to.

Example

```
x = 3.14
print(type(x))
```

Output:

```
<class 'float'>
```

id() – Returns Memory Address of Object

Shows where the value is stored in memory (unique ID).

Example

```
a = 10
print(id(a))
```

Output:

```
 A unique number like:

140934839553936
```

sorted() – Returns a Sorted List

Sorts items in ascending (default) order.

Example

```
print(sorted([4, 1, 3, 2]))
```

Output:

```
[1, 2, 3, 4]
```

enumerate() – Adds Index to Items

Returns index + value pairs.

Example

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

for index, fruit in enumerate(fruits):
```

```
    print(index, fruit)
```

Output:

```
0 apple
1 banana
2 cherry
```

zip() – Combines Two (or More) Lists

Pairs items from multiple lists.

Example

```
names = ["Pavani", "Raju", "Vani"]
marks = [90, 85, 88]

for n, m in zip(names, marks):
    print(n, m)
```

Output:

```
Pavani 90
Raju 85
Vani 88
```

Concatenation – Joining Strings or Lists

Used with the + operator.

String Concatenation

```
a = "Hello "
b = "Pavani"
print(a + b)
```

Output:

```
Hello Pavani
```

List Concatenation

```
list1 = [1, 2]
list2 = [3, 4]
print(list1 + list2)
```

Output:

```
[1, 2, 3, 4]
```

Conversion Functions:

1. int() – Converts to Integer

Changes a number or numeric string into an integer (whole number).

Example:

```
x = int("25")
print(x)
```

Output:

```
25
```

2. float() – Converts to Float

Converts numbers or numeric strings into decimal numbers.

Example:

```
y = float("12.5")
print(y)
```

Output:

12.5

3. str() – Converts to String

Converts numbers, lists, etc., into a string.

Example:

```
a = str(100)
print(a)
print(type(a))
```

Output:

```
100
<class 'str'>
```

4. list() – Converts to List

Converts a string, tuple, set, or any iterable to a list.

Example:

```
print(list("hi"))
```

Output:

```
['h', 'i']
```

5. tuple() – Converts to Tuple

Converts lists, strings, or sets into a tuple (immutable).

Example:

```
print(tuple([1, 2, 3]))
```

Output:

```
(1, 2, 3)
```

6. set() – Converts to Set

Creates a set from a list, string, or tuple.
(Removes duplicates automatically)

Example:

```
print(set([1, 2, 2, 3, 3, 4]))
```

Output:

```
{1, 2, 3, 4}
```

7. dict() – Converts to Dictionary

Requires **pairs** (key, value).

Example:

```
pairs = [("name", "Pavani"), ("age", 21)]
print(dict(pairs))
```

Output:

```
{'name': 'Pavani', 'age': 21}
```

8. bool() – Converts to Boolean (True/False)

Everything except 0, "", None, empty list/set/tuple becomes True.

Example:

```
print(bool(5))
print(bool(""))
```

Output:

```
True
False
```

Mathematical Functions :

1. abs() – Absolute Value

Returns the positive value of a number (removes the minus sign).

Example:

```
print(abs(-10))
```

Output: `10`

2. sum() – Adds All Items

Adds all numbers inside a list, tuple, or set.

Example:

```
print(sum([10, 20, 30]))
```

Output:

```
60
```

3. `min()` – Smallest Value

Returns the minimum value from given numbers or a list.

Example:

```
print(min(3, 7, 1, 9))
```

Output: 1

4. `max()` – Largest Value

Returns the maximum value.

Example:

```
print(max(3, 7, 1, 9))
```

Output:

9

5. `pow()` – Power Function

Raises a number to a power.

Syntax:

```
pow(base, exponent)
```

Example:

```
print(pow(2, 3))
```

Output:

8

(Because $2^3 = 8$)

6. round( ) – Rounds a Number

Rounds a number to nearest whole number or to given decimals.

Example 1 (whole number):

```
print(round(3.6))
```

Output:

```
4
```

Example 2 (decimal places):

```
print(round(3.14159, 2))
```

Output:

```
3.14
```

divmod(a, b) – Returns (quotient, remainder)

```
print(divmod(10, 3))
```

Output:

```
(3, 1)
```

math.sqrt() – Square Root

Requires import.

```
import math
print(math.sqrt(16))
```

Output:

```
4.0
```

math.ceil() – Rounds Up

```
import math
print(math.ceil(4.2))
```

Output: 5

`math.floor()` – Rounds Down

```
import math
print(math.floor(4.8))
```

Output: 4

1. `lambda` — Anonymous Function

A **small one-line function** without a name.

Syntax:

```
lambda arguments: expression
```

Example:

```
square = lambda x: x * x
print(square(5))
```

Output: 25

2. `map()` — Applies a Function to Each Item

Used to apply a function to all items in a list.

Example: Double each number

```
nums = [1, 2, 3, 4]
result = list(map(lambda x: x * 2, nums))
print(result)
```

Output:

```
[2, 4, 6, 8]
```

3. `filter()` — Filters Items Based on Condition

Returns only items that satisfy a condition (True).

Example: Filter even numbers

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

Output:

```
[2, 4, 6]
```

4. `reduce()` — Reduces a List to a Single Value

Located in the `functools` module.

Example: Sum of all numbers

```
from functools import reduce

nums = [1, 2, 3, 4]
total = reduce(lambda x, y: x + y, nums)
print(total)
```

Output: `10`

When to Use These?

Function	Purpose
lambda	create short functions quickly
map()	modify each element
filter()	select specific elements
reduce()	combine all elements into 1 value

What is a Class?

A **class** is a blueprint or template used to create objects.
It contains **variables (attributes)** and **functions (methods)**.

Example:

```
class Student:
    name = "Pavani"
    age = 21
```

What is an Object?

An **object** is an instance of a class.
Using the class, we create objects.

Example:

```
s1 = Student()   # object created
print(s1.name)
```

Output:

```
Pavani
```

1.getattr(object, attribute, default)

Gets the value of an attribute of an object.
If the attribute doesn't exist, you can give a **default value.**

Example

```
class Student:
    name = "Pavani"
    age = 21

s = Student()

print(getattr(s, "name"))          # Pavani
```

```
print(getattr(s, "grade", "NA"))  # NA (default used)
```

2. setattr(object, attribute, value)

Sets/updates the value of an attribute.

Example

```
class Student:
    pass

s = Student()

setattr(s, "name", "Pavani")
setattr(s, "age", 21)

print(s.name)   # Pavani
print(s.age)    # 21
```

3. hasattr(object, attribute)

Checks whether an object has a given attribute.

Example

```
class Student:
    name = "Pavani"

s = Student()

print(hasattr(s, "name"))   # True
print(hasattr(s, "age"))    # False
```

4. delattr(object, attribute)

Deletes an attribute from an object.

Example

```
class Student:
    name = "Pavani"
    age = 21

s = Student()

delattr(s, "age")
print(hasattr(s, "age"))   # False
```

5. isinstance(object, class)

Checks whether an object belongs to a class.

Example

```
class Student:
    pass

s = Student()

print(isinstance(s, Student))  # True
print(isinstance(s, int))      # False
```

6. issubclass(child_class, parent_class)

Checks whether a class is a **subclass** of another class.

Example

```
class Animal:
    pass

class Dog(Animal):
    pass

print(issubclass(Dog, Animal))    # True
```

```
print(issubclass(Animal, Dog))    # False
```

Table:

Function	Meaning	Example
`getattr(obj, 'attr')`	Get attribute	`getattr(s, "name")`
`setattr(obj, 'attr', value)`	Set attribute	`setattr(s, "age", 21)`
`hasattr(obj, 'attr')`	Check attribute exists	`hasattr(s, "name")`
`delattr(obj, 'attr')`	Delete attribute	`delattr(s, "age")`
`isinstance(obj, Class)`	Object belongs to class?	`isinstance(s, Student)`
`issubclass(A, B)`	A is child of B?	`issubclass(Dog, Animal)`

1. `globals()`

Returns a **dictionary of all global variables** in the program.

```
x = 10
y = "Hello"

print(globals())
```

Output (shortened):

```
{'x': 10, 'y': 'Hello', ...}
```

2. `locals()`

Returns a **dictionary of variables inside the current local scope** (usually inside a function).

Example:

```
def my_fun():
    a = 5
    b = 15
```

```
    print(locals())

my_fun()
```

Output:

```
{'a': 5, 'b': 15}
```

3. `callable(object)`

Checks whether an object can be **called** like a function.

Example:

```
print(callable(len))
print(callable(10))
```

- `len` is callable
- `10` is not callable

4. `eval(expression)`

Evaluates a **string expression** and returns the result.

✓ Example:

```
x = eval("10 + 20")
print(x)
```

Another example:

```
a = 5
print(eval("a * 10"))
```

5. exec(code)

Executes a **block of Python code** given as a string.
Unlike `eval()`, it does not return a result.

Example:

```
code = """
for i in range(3):
    print("Hello", i)
"""
exec(code)
```

Output:

```
Hello 0
Hello 1
Hello 2
```

1. Introduction

Exception handling in Python is a mechanism that allows developers to **detect and manage runtime errors**.
Instead of stopping the program abruptly, Python provides structured blocks to handle such errors gracefully.

2. What is an Exception?

An **exception** is an unexpected event/error that occurs during program execution and disrupts the normal flow.

Common examples:

- Division by zero
- Invalid type conversion
- File not found

- Accessing invalid index
- Missing variables

3. Purpose of Exception Handling

Exception handling helps to:

- Prevent program crashes
- Display user-friendly error messages
- Continue normal program execution
- Clean up resources (files, connections)
- Maintain robust and secure applications

4. Exception Handling Keywords

Python uses the following structured keywords:

Keyword	Description
try	Contains code that may cause an exception
except	Handles the exception
else	Executes if no exception occurs
finally	Executes whether exception occurs or not
raise	Manually raises an exception

5. Syntax Structure

```
try:
    # risky code
except ExceptionType:
    # handling code
else:
```

```
    # executes only if no exception
finally:
    # always executes
```

6. `try` and `except`

Used to catch errors and prevent program termination.

Example

```
try:
    result = 10 / 0
except ZeroDivisionError:
    print("Error: Division by zero is not allowed.")
```

7. Multiple Except Blocks

Used to handle different error types separately.

Example

```
try:
    value = int("abc")
except ValueError:
    print("Invalid value format.")
except TypeError:
    print("Type mismatch detected.")
```

8. Catching All Exceptions

Using the base class `Exception`:

Example

```
try:
    print(10 / 0)
except Exception as error:
    print("An error occurred:", error)
```

9. The `else` Block

Executed only when **no exception** occurs.

Example

```
try:
    num = int(input("Enter a number: "))
except ValueError:
    print("Please enter a valid integer.")
else:
    print("You entered:", num)
```

10. The `finally` Block

Executed in all cases—whether an exception occurs or not.
Mainly used for **cleanup**.

Example

```
try:
    file = open("data.txt")
except FileNotFoundError:
    print("File not found.")
finally:
    print("Execution completed.")
```

11. Raising Exceptions (`raise`)

Used to manually trigger an exception.

Example

```
age = -1
if age < 0:
    raise ValueError("Age cannot be negative.")
```

12. Custom Exceptions

Python allows creating user-defined exception classes by inheriting `Exception`.

Example

```
class InvalidAgeError(Exception):
    pass

age = -5
if age < 0:
    raise InvalidAgeError("Age must be positive.")
```

13. Common Built-in Exceptions

Exception	Cause
`ZeroDivisionError`	Dividing by zero
`ValueError`	Invalid value format
`TypeError`	Wrong data type
`IndexError`	Invalid list index
`KeyError`	Missing dictionary key
`FileNotFoundError`	File does not exist

`AttributeError`	Attribute not found
`NameError`	Variable not defined

14. Real-World Example

File Handling

```
try:
    f = open("notes.txt", "r")
    content = f.read()
except FileNotFoundError:
    print("The file does not exist.")
finally:
    print("Closing resources...")
```

What is a Decorator?

A **decorator** is a **special function** that **takes another function as input** and **returns a new function** with **added or modified behavior**.

- It allows you to **"wrap" a function** without changing its actual code.
- Decorators are often used for **logging, access control, caching, timing, or validation**.

Example:

```
def decorator(func):
    def wrapper():
        print("Before function")
        func()          # original function runs
        print("After function")
    return wrapper

@decorator
def say_hello():
    print("Hello!")
 say_hello()
```

Output:

Before function

Hello!

After function

Metaprogramming: `getattr`, `setattr`, `hasattr`, `delattr` are **metaprogramming tools**

Metaprogramming is **writing code that can manipulate other code**—like classes, functions, or methods—**at runtime**.

Python supports this because **everything is an object**, including classes and functions.

Examples

a) Dynamic class creation with type()

```
# Create a class dynamically
Student = type("Student", (), {"name": "Alice", "show": lambda self:
print(self.name)})

s = Student()
s.show()
```

Output:

```
Alice
```

b) Dynamic attributes

```
class Person:
    pass

p = Person()
setattr(p, "name", "python")
print(p.name)
```

Output:

```
Python
```

2. `@staticmethod`

- A **static method doesn't access** instance (`self`) or class (`cls`) data.
- Behaves like a **normal function inside a class**.

Example

```
class Math:
    @staticmethod
    def add(a, b):
        return a + b

print(Math.add(5, 3))  # Call without creating instance
```

Output:

```
8
```

- **Use case:** Utility or helper functions inside a class.

3. `@classmethod`

- A **class method** receives **class (`cls`)** as the first argument.
- Can access **class attributes** but **not instance attributes**.
- Often used as **factory methods** or for **class-wide operations**.

Example

```
class Student:
    school = "Loyola"

    @classmethod
    def show_school(cls):
```

```
        print(f"School: {cls.school}")

Student.show_school()
```

Output:

```
School: Loyola
```

What is the Waterfall Model?

The **Waterfall Model** is a **linear and sequential software development process**.

- Each phase **must be completed before the next phase begins**.
- There is **no overlapping** of phases.
- It is called "Waterfall" because the process flows **downwards like a waterfall**.

Phases of the Waterfall Model

1. **Requirement Analysis**
    a. Gather **all user requirements**.
    b. Prepare **SRS (Software Requirement Specification)** document.
    c. Example: Customer specifies the features they want in a software.
2. **System Design**
    a. Design **system architecture** based on requirements.
    b. Decide **hardware, software, modules, database design**.
3. **Implementation (Coding)**
    a. Developers **write code** according to the design.
    b. Each module is implemented separately.
4. **Integration and Testing**
    a. Combine modules into a complete system.
    b. Test the system for **bugs, errors, and requirement compliance**.
5. **Deployment**
    a. Deliver the software to the **customer/environment**.
    b. Install and configure the system for use.
6. **Maintenance**
    a. Fix any **bugs or issues** that appear after deployment.
    b. Implement **minor updates** or changes requested by the customer.

Features of Waterfall Model

- **Simple and easy to understand**.
- **Structured approach** with clear documentation.
- **Best for small projects** where requirements are **well-known**.
- **Works when changes are unlikely** during development.

Advantages

- Easy to manage due to **linear structure**.
- **Documentation is clear**, so new team members can understand easily.
- Each phase has **specific deliverables**.

Disadvantages

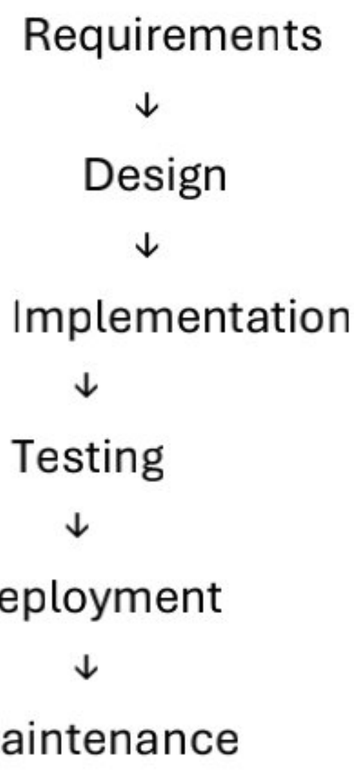
Not flexible: Hard to go back once a phase is completed.

Not suitable for complex projects with changing requirements.

Late discovery of **errors or requirement changes**.

Customer feedback comes **very late**, after deployment.

Waterfall Model Diagram



What is the Agile Model?

The **Agile Model** is an **iterative and incremental software development process**.

- Focuses on **flexibility, customer collaboration, and rapid delivery**.
- Development is divided into **small iterations or sprints**, usually 1–4 weeks.
- Changes can be incorporated **at any stage** of development.

Key Features of Agile

- **Iterative development**: Software is developed in small pieces (iterations).
- **Customer involvement**: Frequent feedback ensures the product meets requirements.
- **Adaptive to change**: Requirements can change during development.
- **Continuous delivery**: Each iteration delivers a **working product**.

Agile Development Cycle

1. **Requirement Gathering**
    a. Collect **user stories** and prioritize them.
    b. Requirements are **dynamic** and can change.
2. **Planning**
    a. Decide the **scope of the iteration/sprint**.
    b. Assign tasks to the development team.
3. **Design and Development**
    a. Design **only what's needed** for the sprint.
    b. Developers write **working code** for the iteration.
4. **Testing**
    a. Test the features developed in the iteration.
    b. Bugs are fixed immediately.
5. **Deployment/Delivery**
    a. Working software is delivered to the customer **at the end of the sprint**.
6. **Feedback & Review**
    a. Customer reviews the iteration.
    b. Feedback is used in the next iteration.

- **Cycle repeats** until the full product is complete.

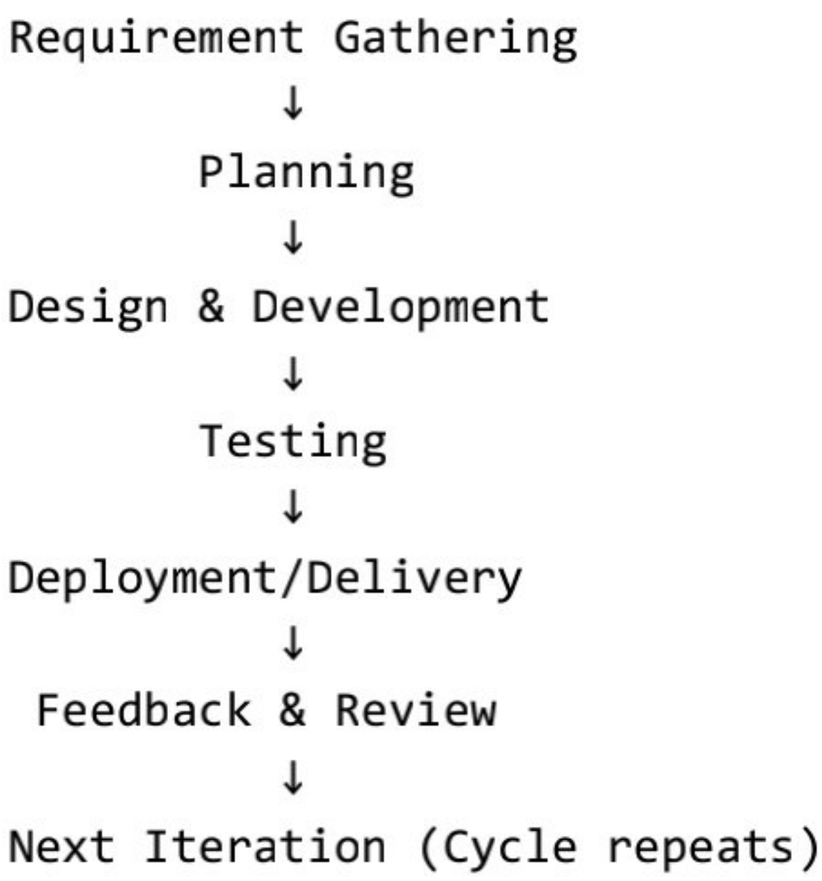
Advantages of Agile

- Flexible to **requirement changes**.
- **Early and continuous delivery** of working software.
- **High customer satisfaction** due to regular feedback.
- Encourages **team collaboration and communication**.

Disadvantages of Agile

- Requires **experienced team members**.
- Can be **chaotic** if requirements change too frequently.
- Hard to predict **time and cost** in advance.
- Not ideal for **large, complex projects** with fixed requirements.

Agile Model Diagram



- Agile is **cyclical**, unlike Waterfall, so **feedback and changes are continuous**.

What is Sorting?

Sorting is the process of arranging data in a **specific order**.

- The order can be **ascending** (smallest to largest) or **descending** (largest to smallest).

- Sorting is very useful because it **makes searching, analyzing, and managing data easier**.

Example

Original list:

```
[5, 2, 8, 1, 9]
```

Sorted in **ascending order**:

```
[1, 2, 5, 8, 9]
```

Sorted in **descending order**:

```
[9, 8, 5, 2, 1]
```

Why Sorting is Important

1. **Faster Searching:** Sorted data allows quick search algorithms like **binary search**.
2. **Data Organization:** Makes data easy to read and analyze.
3. **Required by Other Algorithms:** Many algorithms assume input is sorted (e.g., **merge operations, median calculations**).

What is Bubble Sort?

Bubble Sort is a **simple comparison-based sorting algorithm**.

- It **repeatedly compares adjacent elements** in a list.
- If two elements are in the **wrong order**, it **swaps them**.
- This process continues until the **entire list is sorted**.
- It is called "Bubble Sort" because **larger elements "bubble" to the top** (end of the list) with each pass.

How Bubble Sort Works

Example:

Original list: `[5, 2, 8, 1, 9]`

Goal: Sort in **ascending order**.

Pass 1: Compare and swap adjacent elements

`[5, 2, 8, 1, 9] → [2, 5, 8, 1, 9] → [2, 5, 8, 1, 9] → [2, 5, 1, 8, 9] → [2, 5, 1, 8, 9]`

Pass 2: Repeat comparisons

`[2, 5, 1, 8, 9] → [2, 5, 1, 8, 9] → [2, 1, 5, 8, 9] → [2, 1, 5, 8, 9]`

Pass 3:

`[2, 1, 5, 8, 9] → [1, 2, 5, 8, 9]`

Sorted list: `[1, 2, 5, 8, 9]`

- The largest elements **move to the end** like bubbles in water.

Bubble Sort Algorithm (Steps)

1. Start from the **first element** of the list.
2. Compare the **current element** with the **next element**.
3. If the current element is **greater** than the next, **swap them**.
4. Move to the next pair and repeat steps 2–3 until the **end of the list**.
5. Repeat the whole process **n-1 times** (where n = number of elements).

What is Selection Sort?

Selection Sort is a **simple comparison-based sorting algorithm**.

- It **divides the list into two parts**:
  - **Sorted part** (built from left to right)
  - **Unsorted part** (remaining elements)
- It **repeatedly finds the minimum (or maximum) element** from the unsorted part and **places it in the correct position** in the sorted part.

How Selection Sort Works

Example:
Original list: `[5, 2, 8, 1, 9]`
Goal: Sort in **ascending order**

Step 1: Find the **smallest element** in the list → 1 → swap with first element

```
[1, 2, 8, 5, 9]
```

Step 2: Find the **next smallest element** in the unsorted part `[2, 8, 5, 9]` → 2 → swap with first unsorted element

```
[1, 2, 8, 5, 9]  # 2 is already in correct position
```

Step 3: Next smallest element in `[8, 5, 9]` → 5 → swap with first unsorted element

```
[1, 2, 5, 8, 9]
```

Step 4: Next smallest in `[8, 9]` → 8 → swap (already in place)

```
[1, 2, 5, 8, 9]
```

Sorted list: `[1, 2, 5, 8, 9]`

Selection Sort Algorithm (Steps)

1. Start from the **first element** of the array.
2. Find the **minimum element** from the unsorted part.
3. Swap it with the **first element of the unsorted part**.
4. Move the boundary of the sorted part **one step to the right**.
5. Repeat until the entire array is sorted.

What is Insertion Sort?

Insertion Sort is a **simple comparison-based sorting algorithm**.

- It **builds the sorted list one element at a time**.
- Each element is **inserted into its correct position** in the already sorted part of the list.
- It is similar to **sorting playing cards in your hands**.

Original List

`[8, 3, 5, 2, 7]`

Goal: Sort in **ascending order**

Step 1: Find the smallest number

- Look at all numbers: `8, 3, 5, 2, 7`
- Smallest = 2 → put it first

List now:

`[2, 8, 3, 5, 7]`

Step 2: Find the next smallest number

- Remaining numbers: `8, 3, 5, 7`
- Next smallest = 3 → put it after 2

List now:

`[2, 3, 8, 5, 7]`

Step 3: Find the next smallest number

- Remaining numbers: `8, 5, 7`
- Next smallest = 5 → put it next

List now:

`[2, 3, 5, 8, 7]`

Step 4: Find the next smallest number

- Remaining numbers: 8, 7
- Next smallest = 7 → put it next

List now:

`[2, 3, 5, 7, 8]`

Step 5: Last number

- Only 8 remains → already in correct position

Sorted List:

`[2, 3, 5, 7, 8]`

What is Quick Sort?

Quick Sort is a **fast, comparison-based, divide-and-conquer sorting algorithm.**

- It works by **choosing a pivot element** from the list.
- **Partition the list** into two parts:
  - Elements **less than or equal to the pivot**
  - Elements **greater than the pivot**
- Recursively **sort both parts**, then combine them.
- It is **faster than Bubble, Selection, and Insertion Sort** for large lists.

How Quick Sort Works (Conceptually)

Example:

Original list: [8, 3, 5, 2, 7]

1. **Choose a pivot** → usually the **first element, last element, or middle element.**
    a. Let’s choose **pivot = 8**
2. **Partition the list** around the pivot:
    a. Elements ≤ 8 → [3, 5, 2, 7]
    b. Pivot → [8]
    c. Elements > 8 → []
3. **Recursively sort the left part [3, 5, 2, 7]**
    a. Choose pivot = 3
    b. Partition: ≤3 → [2, 3], >3 → [5, 7]
    c. Combine: [2, 3, 5, 7]
4. **Combine all parts**: [2, 3, 5, 7] + [8]

Sorted list:

[2, 3, 5, 7, 8]

Quick Sort Algorithm (Steps)

1. Choose a **pivot element** from the list.
2. **Partition** the list:
    a. Elements ≤ pivot → left
    b. Elements > pivot → right
3. **Recursively apply Quick Sort** on left and right sublists.
4. **Combine** the sorted left, pivot, and sorted right.

What is Merge Sort?

Merge Sort is a **divide-and-conquer sorting algorithm.**

- It **divides the list into smaller sublists,** sorts them individually, and then **merges them to form the final sorted list.**
- Very efficient for **large lists.**

How Merge Sort Works (Conceptually)

Example:
Original List

[8, 3, 5, 2, 7]

Goal: Sort in **ascending order**

Step 1: Divide the List

Divide the list into two halves:

Left: [8, 3, 5] Right: [2, 7]

Step 2: Divide Each Half Again

- Left [8, 3, 5] → [8] and [3, 5]
- [3, 5] → [3] and [5]
- Right [2, 7] → [2] and [7]

Now all sublists have **1 element**, which are already sorted:

[8], [3], [5], [2], [7]

Step 3: Merge Sublists

- Merge [3] and [5] → [3, 5]
- Merge [8] and [3, 5] → [3, 5, 8]
- Merge [2] and [7] → [2, 7]

Step 4: Merge Left and Right Halves

- Merge [3, 5, 8] and [2, 7]

Compare elements:

2 < 3 → 2
3 < 7 → 3
5 < 7 → 5
7 < 8 → 7

8 → 8

Resulting sorted list:

[2, 3, 5, 7, 8]

Sorted List:
[2, 3, 5, 7, 8]

Merge Sort Algorithm (Steps)

1. If the list has **1 element**, it is already sorted.
2. Divide the list into **two halves**.
3. Recursively apply **Merge Sort** on each half.
4. **Merge** the two sorted halves into a **single sorted list**.

Key Points

- **Time Complexity:** O(n log n) for best, average, and worst case
- **Space Complexity:** O(n)
- **Stable:** Yes, maintains the relative order of equal elements
- Works well for **large datasets** or **linked lists**

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}")
```